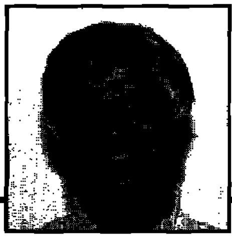
JOURNAL OF
OBJECT-ORIENTED programming

November/December 1991
Vol. 4, No. 7

Features

The Journal of Object-Oriented Programming (ISSN #0896-8438) is published nine times a year, monthly except for Mar/Apr, Jul/Aug, and Nov/Dec. Published by SIGS Publications, Inc., 588 Broadway, Suite 604, New York, New York 10012, (212)274-0640. Please direct advertising inquiries to this address. Second class postage paid at New York, New York, and additional mailing offices. POSTMASTER: Send address changes to JOOP, P.O. Box 3000, Dept. OOP, Denville, NJ 07834. Inquiries and new subscription orders should also be sent to that address.

Manuscripts under review should be typed double spaced (in triplicate). Editorial correspondence and Product News information should be sent to the Editor, Dr. Richard S. Wiener, 2185 Broadmoor Road Circle, Colorado Springs, CO 80906, (719)520-1356.



Editorial

It was nice to meet with so many of our readers and writers at OOPSLA this past week (October 7–11). I found OOPSLA to be an interesting and important conference. The technical papers at the conference focused on experiences with object orientation. The industry now has some real experiences, both successes and failures. Some of the more successful OOP book authors were on hand to discuss their recipes for O-O analysis, design, and programming. Representatives from the entire OOP industry were present. It was evident to me that the move toward making object orientation a mainstream activity is continuing. Of course, it is important to temper this observation by the fact that whenever one is immersed in a sea of advocates of any technology it is easy to believe that the whole world has embraced the technology. In reality, this has not yet happened with object-oriented technology and may take several more years to occur. Many computer science departments are still teaching their students structured analysis, design, and programming techniques exclusively. Some schools have just started offering a few elective courses on object orientation.

It was clear from the vendor area that most of the products exhibited featured C++ or Smalltalk language development or software development tools. These two OOP languages have "won" the language wars in the commercial sector, at least for the time being. Application frameworks and CASE tools for C++ were probably the most popular products on display. The major application areas that are pushing OOP technology into the mainstream are O-O database management and the development of graphical user interfaces.

We at *JOOP* would like to report on experiences with object orientation and therefore plan to produce a special supplement dedicated to this subject in 1992. I would like to solicit contributions now for this special supplement. Please follow the normal *JOOP* guidelines for submission and mail your manuscripts to the editorial office. You are welcome to call me at the editorial office to discuss ideas for such contributions.

This issue contains four feature-length articles.

"Contravariance for the Rest of Us" by Warren Harris discusses a structural weakness of C++ related to overloading. The article suggests areas of needed improvement for C++.

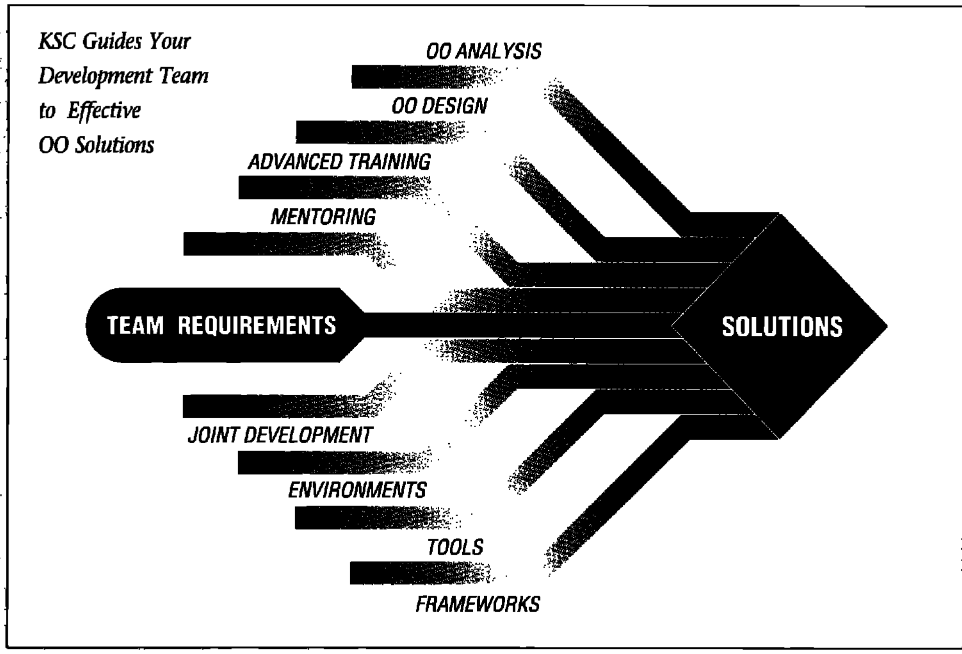
"Multilevel Secure Object-Oriented Data Model: Issues on Noncomposite Objects, Composite Objects, and Versioning" by Bhavani Thuraisingham examines the issues related to maintaining multilevel security of data in an object-oriented environment.

"Delegation in C++" by Ralph Johnson and Jonathan Zweig examines delegation as a language feature that replaces inheritance. The article explores how delegation may be used in C++.

"Real-World Reuse" by Mark Lorenz looks at how application developers work with and view their application classes and how this relates to analysis, design, and the hierarchy of classes used for an application.

Richard S. Wiener

Build Your Smalltalk Team with a World-Class Partner



To build a first rate development team that delivers the maximum benefits of object-oriented technology, you want only the most highly qualified experts to guide you.

Knowledge Systems Corporation offers a cohesive program of object technology training, services, and products. Our unparalleled years of commercial Smalltalk experience enable you to build the most effective advanced development team possible. We give you a powerful strategic advantage by helping your team to:

Increase Productivity
Manage Complexity
Reduce Development Costs

KSC teams with clients to address application areas such as:

- Decision Support Systems
- Manufacturing Information Systems
- Manufacturing Process Modeling
- Financial Transaction Management
- Financial Trading Systems
- Simulation Environments
- Network Management
- Custom GUIs
- SQL Access

KSC's technical experts have helped build in-house development teams and OO/Smalltalk solutions for companies such as:

IBM (Atlanta, Cary, Dallas, Austin)
The Capital Group
Hewlett-Packard
General Electric
Bell Northern Research
Texas Instruments
American Airlines
Boeing Computer Services
Northern Telecom
Texaco
NCR

KSC puts you in the forefront of advanced software development. We provide time-tested solutions to meet your company's specific OO/Smalltalk training and development requirements.

Knowledge Systems Corporation
Partners in Advanced Software Development

114 MacKenan Drive
Cary, NC 27511
(919) 481-4000

Circle 54 on Reader Service Card

Guest Editorial

Code reuse considered harmful

The advantages of object-oriented programming (OOP) do not come just by using inheritance. As the first wave of enthusiasm passes by, I see an awareness of the importance of applying some method to what classes and inheritance are used for. This can be seen both in conference papers and in the quickly growing literature on O-O methods.

The Scandinavian school of OOP can be characterized by its view on these matters. The key word here is "modeling," in short, that programs and class hierarchies should describe concepts and be understandable in the application domain. An implication of this point is that subclassing should be used for modeling specialization of concepts in the same way as Linnaeus used specialization as a method to describe the classification of plants. The Scandinavian school thus has a very firm view on what subclassing and inheritance should be used for and is in contrast to at least two other points of view.

The "type" view concentrates on the signatures of operations and classes, i.e., on parameter types and operation names. Two classes are compatible (have the same type) if they implement operations having the same name and parameters. In the extreme, these types and the relations between them could be calculated automatically. As an example, consider two classes: class Rectangle with operations Move and Draw and class Cowboy with operations Move, Draw, and Shoot. Considering only signatures would lead to the conclusion that Cowboy is a subtype of Rectangle. From a modeling point of view, this is simply nonsense. The effect seems related to the "structure equivalence" approach used in very early Pascal implementations where integers representing numbers of apples and pears would be happily added together in spite of the fact that they were declared as different types by the programmer. This problem was cured by introducing the notion of "name equivalence."

A third point of view is to concentrate on code reuse an construct the class hierarchy to minimize the code volume. I cannot refrain from comparing this with earlier approaches in the history of our science. In the microscopic scale, goto:s were once (a long time ago) used to "reuse" fractions of code with well-known problems of "spaghetti code" as the result. The note "Goto considered harmful" by Dijkstra marks the turning point in the use of structured algorithmic constructs. Interestingly enough, this sometimes leads to some repetition of similar code, which is generally accepted.

The same pattern can be seen in the use of procedures, originally only viewed as a means for saving coding labour — any program fragment that would shorten the program (and possibly reduce the binary code size) would qualify for being turned into a procedure. Singling out one contribution, I select the book *Structured Programming* by Dahl/Dijkstra/Hoare to represent the shift in attitude toward using procedures to model algorithmic abstractions. It was now acceptable to write procedures that were called only once.

Focusing on using inheritance for code reuse leads to the problems as described above for statements and procedures. "Spaghetti" inheritance with artificial relations between classes makes them hard to understand and thus to use. Some inherited methods may not be used and such conventions have to be understood and obeyed. In the view of the Scandinavian school, the use of inheritance for code reuse is bad in the same sense as excessive use of goto:s and code-saving procedures. Here I also must point out that inheritance is not the only way to (re)use code. Aggregation and creation of separate objects to do the job often serve as good alternatives.

Although rarely spelled out in clear, the increasing interest in analysis and design has resulted in a higher awareness of the importance of how class hierarchies are designed. It is not surprising that the Scandinavian school puts emphasis on modeling. The first general-purpose object-oriented programming language, Simula 67, was developed in Norway by Kristen Nygaard and Ole-Johan Dahl. The development of Simula was triggered by the construction of simulation models where modeling of real world concepts and behavior is explicit. Inheritance was thus developed to represent specialization of concepts concepts — no wonder it is for that purpose it works best.

Boris Magnusson
Lund University

[redacted]

Contravariance for the rest of us

by Warren Harris

Software and Systems Laboratory, Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94303

RECENT RESEARCH HAS DEMONSTRATED that subtyping and inheritance are distinct relationships [Cook90]. Primarily, the difference arises because of something called *contravariance* and its effects on object-oriented programming. Contravariance is a phenomenon that occurs as an interaction between subtyping and higher-order functions and has important implications for object-oriented programming. It affects all object-oriented programming languages, including C++, and is usually circumvented by *overloading*. However, overloading does not always have the desired effect, which we will illustrate with actual C++ examples. Finally, we will discuss what a better—more expressive and type safe—language might look like.

WHAT IS CONTRAVARIANCE?

We all have an intuitive notion of what it means for one type to be a subtype of another. We would expect that a value of a subtype can be used anywhere a value of a supertype is expected. Values of a subtype, though, can potentially do more, i.e., support a richer set of operations, than values of the supertype. The difference between the subtype and supertype reflects the increased functionality of the values. In some sense, a subtype is *more specific* than its supertypes. What does it mean to be more specific?

Let us approach this question intuitively. From an implementation standpoint, a data structure is more specific if it has all the fields of its parent but adds additional fields. From an interface standpoint, we would expect a data type to be more specific if it has all the operations of its parent but adds additional operations. However, in object-oriented programming it is often necessary not only to add new operations but also to restrict operations that are inherited. The question then arises: ***what does it mean for one operation to be more specific than another?***

For simplicity's sake, we can think of the operations on objects simply as functions (we will ignore the dispatching aspect of sending a message temporarily). We can now ask what it means for one function to be more specific than another. The type of a function is expressed in terms of the types of its arguments (if any) and the type of its result. We can summarize the subtype relationship between functions as:

> The type of a function is a subtype of the type of another function if (all else being the same) the result type is more specific, or any of the argument types are more general.[1]

Result types are said to be *covariant*— they vary in the same way as the function types. Result types must be more specific for the function type to be more specific. Argument types are said to be *contravariant*— they vary in the opposite way as the function type. Argument types must be more general for the function type to be more specific.

This seems counterintuitive. One would expect an operation defined over employees to be more specific than one defined over all people. The following example will illustrate why this is not true.

EXAMPLE

The whole issue of contravariance comes into play when we manipulate functions from within programs. Functions that manipulate other functions are called ***higher order***. Higher-order functions typically are passed to other functions as arguments and *apply* the functional argument to some values.[2]

When a language involves subtyping, we become concerned about higher-order functions being passed functions that are subtypes of the type required. We would like to check that a function's type is indeed a subtype of the required type and thereby verify that the program will not get runtime errors from being passed and subsequently invoking an inappropriate function.

This is a simple (contrived) example involving some subtypes

[1] This is also true of functions that return no values (void), in which case we simply ignore restrictions on the results, and in functions that return multiple values, in which case each of the results must be either the same or more specific.

[2] Higher-order functions may also obtain a function to apply by other means — either as a piece of literal data or by retrieving one from an external data structure.

and a higher-order function. Let us define a "person" to have a "name," an "employee" to have a "salary" and inherit from person (thereby also having a name), and a "manager" to have someone s/he "manages" (to keep it simple, we will make this a single employee rather than a set) and also inherit from employee (thereby also having a name and salary). We will use C++ classes to specify some structural inheritance (i.e., all the fields from a superclass will also be available in a sub-class):

```
class Person
{
  public:
     char* name;
};

class Employee : public Person
{
  public:
     int salary;
};

class Manager : public Employee
{
  public:
     Employee* manages;
};
```

Now, suppose there exists a collection of functions over these data types. To keep it simple, we will define a set of print functions to print out various fields of the objects. Of course, we could just as well use member functions (methods) but regular functions will be sufficient to illustrate how contravariance works:

```
void print_name(Person* p)
{
  cout << p->name;
};

void print_salary(Employee* e)
{
  cout << e->salary;
};

void print_manages(Manager* m)
{
  cout << m->manages->name;
};
```

Now let us define a higher-order function (a function that takes another function as a parameter and applies it). The higher-order function do-with-banner could take an operation applicable to Employees (such as one of the print functions) and an instance that was at least of type Employee. It would first print some banner, then apply the function:

```
void do_with_banner(void (*action)(Employee*), Employee* employee)
{
  print_banner();
   (*action)(employee);
};
```

Suppose there is a single distinguished Employee instance called employee_of_the_month:

```
Employee* employee_of_the_month;
```

A working example of this simple function is:

```
do_with_banner(print_salary, employee_of_the_month);
```

Now, one would suspect that the following piece of code should signal a compile time error:

```
do_with_banner(print_manages, employee_of_the_month);
```

because we have no way of knowing whether the employee of the month will be a manager or not until runtime (with a specific Employee instance).

Conversely, the following code should work just fine:

```
do_with_banner(print_name, employee_of_the_month);
```

because we know that employee_of_the_month will always at least be an Employee and, therefore, will always have a name (inherited from the Person class).[3]

From this example we can see that functions that are acceptable as arguments to the higher-order function do_with_banner must themselves take arguments of type Employee, or a more *general* type. The arguments to print_name are more general than the arguments to print_salary, therefore, the type of the print_name function is more *specific* than the type of the print_salary function. The print_name function can be used anywhere print_salary can be used. In other words, to be used by do_with_banner, the function must *at least* be defined on Employees (i.e., take Employees or a more specific type as an argument). This is contravariance.

Ultimately, contravariance has ramifications for object-oriented programming. We will examine this in the next section.

HOW IS CONTRAVARIANCE RELEVANT TO OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming's message-passing paradigm *inherently* involves higher-order functions. Even though the user may not write higher-order functions directly, messages act as higher-order functions that invoke individual methods according to the particular object involved.[4] When objects are passed as arguments or returned as values, their methods are actually being passed around, too, just as with higher-order functions.

Let us look at the message dispatch process in detail. When an

3 C++, unfortunately, does not allow this code to pass through the compiler even though it really should work. This is because it does not permit function subtyping at all. Functions must be of exactly the right type to be passed as arguments.

4 Whether or not this method lookup is done at runtime (as with C++ virtual methods) or at compile time (as with its regular methods), the higher-order nature still exists. Contravariance still plays a crucial role in the type checking of methods.

object is sent a message with some arguments, a method that will handle the message is looked up. This method is associated with the particular object and is usually fetched from a table that is accessible from the object. The method is then applied to the arguments and any result returned from the method is also returned from the message dispatcher to the caller. Therefore, sending a message *is* calling a higher-order function.

Since arguments to a message ultimately become arguments to the method and since the method is invoked from within the (higher-order) message dispatcher, *method arguments are subject to contravariance.*

Now, when we type check a method of a subclass that overrides a method of a superclass with the same name we should observe the contravariance rule. This way we can guarantee that the new method will apply to everything that the overridden method applied to and, therefore, the subclass can be used anywhere the superclass can be used. Basically:

> A method of a subclass is more specific than the method it overrides from a superclass if (all else being the same) its result type is more specific, or any of the argument types are more general.

When all the methods of a subclass are equally specific or more specific than the methods of a superclass, the interface of the subclass (the method names and their types) is said to *contain* the interface of the superclass [Canni89a]. *When one interface contains another, instances of that interface can be used wherever instances of the other interface are required.* This notion of containment is *exactly* the same as the notion of subtyping.

This seems simple so far. However, in practice it is not always the case that we want the interface of a subclass to contain the interface of a superclass. What is important is to be able to inherit some methods from the parent class and restrict other methods that must be overridden to make the new class work. One case of this restriction is when arguments to methods must be more specific (be a subtype of the type of the corresponding argument in the parent class) for the new implementation to work properly. Since method arguments are contravariant, making them more specific actually causes subclass interface not to contain the interface of the parent class. In other words, *inheritance is not subtyping,* at least in some cases.

Perhaps the most common occurrence of this phenomenon, where inheriting does not produce subtyping, is when a method must take an argument that is the same type as self (i.e., the type of this in C++).[5] The following example will illustrate:

EXAMPLE

The following example illustrates what we might like to achieve with some code that implements windows and presenters (windows that display an associated object). For convenience, we will write this code in C++ although C++ actually behaves a bit differently. Later, we will describe this difference and what the programmer must do to get around it.

```
class Window
{
  public:
    virtual void insert(Window*);
    ...
};
class Presenter : public Window
{
  public:
    virtual void insert(Presenter*);
    virtual void layout();

    ...
};
```

The intention of this example is that Presenter's insert method override the method inherited from Window while at the same time introducing an additional restriction: Presenters can only have children added to them that are themselves Presenters. One might want to do this because insert will invoke another method (like layout) on each of the inserted children.

A problem arises with this interpretation of the above code in that the interface to Presenter no longer *contains* the interface to Window. This is because all Windows allow other Windows to be inserted as children, whereas Presenters only allow other Presenters. A Presenter cannot be passed to any arbitrary piece of code that expects to receive a Window because it may try to add a child window to it that is a Windows rather than a Presenter:

```
Window* add_a_child(window* w)
{
  Window* child = new Window();
  w->insert(child);
  return w;
};
```

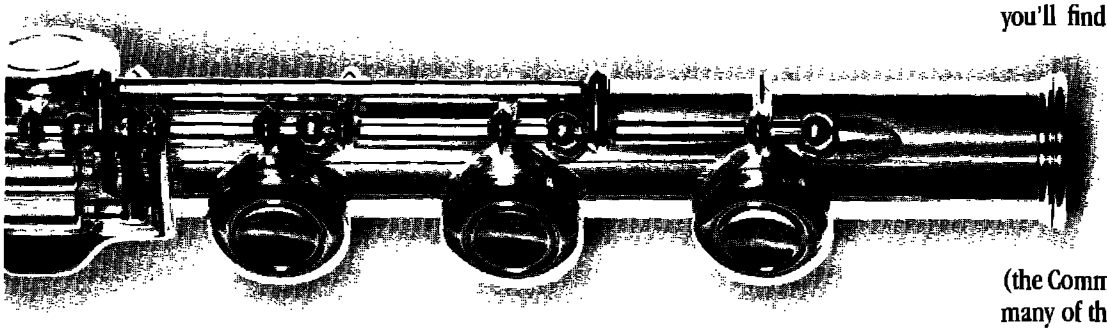
In some sense, the definition of Presenter has *taken away* the insert operation inherited from Window. It is not really a subtype anymore because of this missing operation. It instead includes a more specific operation (also called insert) that only applies to other Presenters.

In actuality, C++ does not take away the inherited operation. Instead, it *overloads* the name "insert" and allows both definitions to exist simultaneously. Even though we read both methods as insert, the compiler treats them as two separate methods. It is in this way that C++ guarantees that subclasses satisfy the interface of the parent.

There is a problem with overloading, however. Even though the code will not get a runtime error because a Window was inserted as a child of a Presenter, what will happen is that the wrong method will be invoked (the inherited insert method). From within add_a_child the Window will indeed be inserted, but the layout method will not be called. Such a maneuver can seriously violate the intended semantics of the program.

[5] In C++, we are not allowed to say "the type of this, however this may have been inherited." The language Eiffel does support this notion via "like Current."

Are you trying to play Carnegie Hall



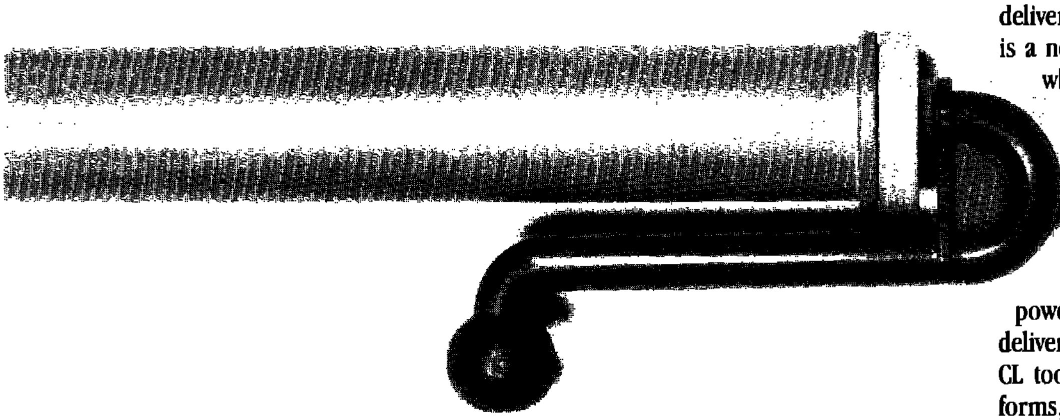
If you're developing a simple application, you'll find that C++ is an adequate object-oriented language. But if you're working on a complicated application, you'll need a development environment that can actually handle the job.

Allegro CL® with CLOS

Allegro Common LISP with CLOS (the Common LISP Object System) automates many of the tasks you'd have to do manually with C++. Allegro CL with CLOS has its roots in the original OOP languages. It has matured over the years into a powerful development environment for complex applications.

	Allegro CL with CLOS	C++
Multiple Inheritance	YES	YES
Polymorphism	YES	YES
Encapsulation	YES	YES
Interactive	YES	NO
Incremental Compilation	YES	NO
Auto. Memory Management	YES	NO
Meta-level Facility	YES	NO
Method Combination	YES	NO
Dynamic Redefinition	YES	NO
Standard Class Library	YES	NO

...with a street corner instrument?



We've also made a recent breakthrough in delivery of LISP applications. Allegro Presto is a new automatic function loading system which reduces the runtime size of each LISP application. Functions required by the application are loaded dynamically; the LISP image is then "frozen" for delivery.

Now you can get a fully-functional OOP environment, the prototyping power of LISP, and an elegant new way to deliver your application. Of course, Allegro CL tools work on all popular UNIX™ platforms, from the 386™ to Cray computers.

True OOP virtuosity can only be obtained with the help of the best LISP instruments—and the support of a company devoted to the success of your application.

FRANZ INC.

1995 University Avenue, Berkeley, CA 94704
TEL: 415-548-3600 FAX: 415-548-8253

Sometimes it is the case that we really do want to override a method and restrict its usage. In these cases, the new class is not really a subtype of the parent.

In such cases, the compiler should not allow subclasses to be used wherever the superclass is specified. In the above example, the correctness of the program does in fact depend on Windows *not* being inserted as children of Presenters.

WHAT DO C++ PROGRAMMERS REALLY DO?

There are five ways in which C++ programmers typically circumvent the problem of subclasses not being subtypes and overloading not performing what is actually desired:

1. Often in C++, we are unfortunately inclined to loosen type restrictions. In this case, we change the argument to Presenter's insert method so that the Presenter class becomes:

```
class Presenter : public Window
{
  public:
    virtual void insert(Window*);
    virtual void layout();
    ...
};
```

   The programmer must assume that at runtime insert will indeed be called with a Presenter rather than a Window. Then, if Presenter operations are to be performed on the w parameter, "casts" must be used to short-circuit the type checker. As a result, the type checker performs the role of verifying that the programmer indeed declared what operations s/he was interested in (via casts to classes that support those operations) rather than verifying that the entire program hangs together as a consistent whole. This really nullifies much of the benefit of type checking.[6]

2. A cleaner solution in this case would be to define a third class from which Window and Presenter both inherit. This class, SimpleWindow, could provide everything Window provided except the insert method. Window and Presenter would then be disjoint classes, each with their own version of insert, and the compiler would be able to detect that one is an unacceptable data type to a routine that expected the other.

   This solution is infeasible when we consider that classes like Window are often contained in libraries and that it is not possible to repartition its set of methods so that we could inherit some and override others. A completely usable *and* type-correct library would have to consist of a large number of classes each containing a single method. These classes would then be combined together with multiple inheritance to form the desired classes. This is highly impractical, and defeats the primary benefit of object-oriented programming — ease of programming through inheritance.

3. Rather than trying to split the Window class into two portions so that we can inherit from the part we need, we could instead use *private* inheritance:

```
class Presenter : private Window
{
  public:
  virtual void insert(Presenter*);
  virtual void layout();
  ...
}:
```

   Private inheritance allows the implementation of Window to be used inside the implementation of Presenter, but does no allow the Window methods to be available to clients of Presenter. Effectively, this makes Presenter inherit from Window but not be a subtype of it. This is exactly what we want in this case—with one exception. Although clients of Presenter are completely protected from inadvertently invoking Window's insert methods, the Presenter implementation itself is not. If inside one of Presenter's methods the insert method is invoked, the problem arises again. This is because Window's insert method is still privately available. Programs can thereby type check but produce the wrong behavior at runtime.

4. Another solution that is often used is the encoding of runtime "type" information into objects. Routines like Presenter's insert would first check some sort of tag field within the object before proceeding to assume the object actually is a Presenter, even though the compile-time type information declared the object to be only a Window. Such solutions not only are time-consuming to implement and decrease the performance of the running system but they also introduce the question of how to recover from type errors at runtime.

5. Perhaps the solution used most often is to further overload methods to keep unwarranted methods from applying. In the Presenter example, we would define yet another insert method:

```
class Presenter : public Window
{
  public:
    virtual void insert(Window*);
    virtual void insert(Presenter*);
    virtual void layout();
    ...
};
```

   The first insert method, insert(Window*), would simply prevent the Window class's insert(Window*)from being used. This method would either ignore the attempt to insert or signal some form of runtime error. The second insert method, insert(Presenter*), would actually implement the desired semantics.

[6] In fact, several large C++ applications have been forced into this style of coding where all variables in the system are basically of the most general type (e.g., the NIH Class Library of Smalltalk-like classes). The safety of such applications leaves much to be desired.

This solution seems unsatisfying in that these dummy methods must be around at runtime simply because the compiler could not catch at compile time the cases where they would be invoked. A correct application should never call them. This solution also has problems in that the choice of whether to use the insert(Window*) method or the insert(Presenter*) is determined at compile time. This choice is based on the declared type of arguments at the call sites of insert rather than the actual type of the arguments at runtime. Since C++ preserves no type information at runtime, the programmer is forced into one of the previously mentioned solutions.

WHAT ELSE CAN BE DONE?

Some of the problems with C++'s overloading mechanism stems from the fact that only the object can be used to discriminate methods at runtime (i.e., virtual methods). The types of all other arguments are factored away at compile time when the overloaded names are resolved. Single argument dispatch allows a simple table to be used for the method lookup process.

The language CLOS [Bobro88] allows any number of arguments to be used in the runtime method lookup process and terms these *multimethods*. Multimethods also eliminate the problem with contravariance (i.e., that subclasses may not be subtypes) because, like C++, they overload message names. Multimethods defer the entire lookup process until runtime, not just the lookup associated with the "first" argument, and therefore permit many correct method invocations that C++ would reject.

Although multimethods are more general, they carry along with them all the same problems with overloading found in C++. Basically, if a more general method is not found that corresponds to the types of the actual parameters (obeying contravariance), a method from a superclass that is not a supertype may be used instead. As we have already seen, in most cases this method will not be able to preserve the intended semantics of an application and, in general, is always the incorrect method to call. However, rather than immediately generating a "no applicable method" error, subsequent errors will arise that are much removed from the actual problem (e.g., sending a Window a layout message rather than disallowing the call to insert a Window into a Presenter in the first place).

With each CLOS method invocation, there must always be some method in the system with every formal parameter at least as general as each actual parameter in the invocation. Without a type checker, it is possible to have some actual parameters be more specific while others are too general and consequently no method will be found at runtime. Programmers are left to visualize the crossproduct of all possible parameter types, both to ensure that some method will exist and to determine exactly which method will apply in a given situation. The simple conceptual model of inheriting methods from a class lattice can no longer be used.

WHY HAS CONTRAVARIANCE NOT BEEN A PROBLEM BEFORE?

For one thing, contravariance only arises when subtyping is involved. Since languages like C do not have subtypes (i.e., the arrangement of types into a generalization/specialization hierarchy), contravariance does not come up as a problem.Languages like Smalltalk [Goldb83] and CLOS do indeed exhibit contravariant behavior but types are not checked statistically. At runtime, it is possible to get a type error because the wrong type of function was passed as an argument. This may not seem to happen in most working programs, but it is not possible to guarantee that it will not happen in general without, essentially, type checking. Sometimes certain bugs are not encountered for months or years, simply because the right combination of data has not been encountered that would cause a certain portion of code or method body to be executed. When the faulty code is finally executed, a type error that could have been caught statistically finally occurs. Also, as a program becomes larger it becomes increasingly difficult to ensure that portions of it (possibly written by different programmers) will work together reliably.

WHAT CAN BE DONE TO MAKE PROGRAMMING TYPE SAFE?

Research underway at Hewlett-Packard is striving to make object-oriented programming type safe without being too restrictive as are C++ and Simula. In other words, we want to guarantee that a pice of code will not break at runtime because it was handed a piece of data of the wrong type. To do this, we are careful to make a distinction between *classes* (which specify implementations) and *types* (which specify interfaces). By observing the rules of contravariance (and a few other), we can statistically determine when a class is an acceptable implementation for a piece of code that expects a certain type.

Checking that certain pieces of code are type safe is only half the problem, though. We also desire that the language be expressive enough to concisely encode the problem we are trying to solve. This includes allowing generic code to be stored in libraries and reused. This is accomplished in two ways. The first is by allowing implementation (class) inheritance to be independent from interface inheritance (subtyping). The second is through property of *parametric polymorphism.* Parametric polymorphism is the ability to parameterize a piece of code over the types that it can potentially handle. In some sense, it establishes constraints between the types in a piece of code. Parametric polymorphism can further be broken down into simple (*unquantified*) parametric polymorphism, *bounded quantification*, and *f-bounded quantification.* We will examine each of these features in turn.

Let us reconsider the Window and Presenter types to show how we can separate the subtype and subclass notions:

```
interface Window
{
  methods:
    insert(Self) returns Void
    ...
};
interface Presenter
{
  inherits: Window
  methods:
    layout() returns Void
};
```

These interface definitions define the operations available on the types Window and Presenter respectively. Window defines an insert operation (method) that takes another Window as a parameter and returns nothing. Self indicates that the *same* type as this interface is required. If the Window interface is inherited, the type Self will change to reflect the inheritance. In the case of Presenter, insert will be available but will require another Presenter as an argument.

At first, the dissention between this and C++ may seem nominal but it allows the type checker to ensure that both the call to insert one Window into another and to insert one Presenter to another will succeed, whereas attempting to mix the two types will be caught at compile time. This is because of the contravariant use of Self in a method signature.

Moreover, the type checker will catch an inadvertent mixing of the two types even if a Presenter class (a specific implementation of the Presenter interface) inherits most of the code from a Window class. The type checker can also determine that the programmer will have to supply a new insert method for Presenters because of the contravariant use of Self.

Here is what some working examples of insert might look like:

```
w1 : Window = make_Simple_Window(...);
w2 : Window = make_Bordered_Window(...);

w1.insert(w2);

p1 : Presenter = make_Column_Presenter(...);
p2 : Presenter = make_Graph_Presenter(graph1, ...);

p1.insert(p2);
```

As previously mentioned, parametric polymorphism can be used to parameterize a piece of code over the types that it can potentially handle. Using parametric polymorphism, we can rewrite the do_with_banner function as:

```
function do_with_banner[T : TYPE](fn : T -> Void, arg : T) returns Void
{
  print_banner();
  fn(arg);
};
```

This polymorphic function establishes a constraint that the type of the parameter to the fn argument must be the same as the type of arg. The square brackets specify the type parameter T, which is evaluated at compile time. We could use this function as follows:

```
do_with_banner[Employee](print_name, employee_of_the_month);
```

or, if we knew that in a certain section of code employee_of_the_month was bound to a Manager:[7]

```
do_with_banner[Manager](print_manages, employee_of_the_month);
```

However, when writing reusable routines it is often necessary not only to specify that two arguments must be the same type but also to specify that that type must support *at least* a certain interface. This is because we know that the argument will be used in a certain way such as being sent a specific message. The object had better be able to support that message. This can be done by what we call ***bounded quantification.*** Bounded quantification is just a way of saying that an object must be at least a certain type. For example:

```
function add_a_child[Win : CONTAINS[Window]](w : Win) returns Win
[
  child : Window = make_Window();
  w.insert(child);
  return w;
];
```

Here, CONTAINS[Window] specifies that the type variable Win must be at least as specific as the type Window. We may now call add_a_child to add a child window to any Window or subtype of Window that contains the Window interface:

```
w1 : Window = make_Window();
add_a_child[Window] (w1);

w2 : Bordered_Window = make_Bordered_Window();
add_a_child[Bordered_Window](w2);
```

where the Bordered_Window interface contains the Window interface. We could not, however, write:

```
p1 : Presenter = make_Presenter();
add_a_child[Presenter](p1);
```

because, as we have seen in the previous section, Presenter does not contain the Window interface because its insert method requires an argument that is too specific.

Interestingly, because of polymorphism this new definition of add_a_child knows that the result of calling add_a_child will be the same type as its argument. The C++ definition will only know that the result is a Window*.

Sometimes, however, it is desirable to write functions that operate over not only all interfaces that contain a given interface, but also over all interfaces that are recursive in the same way, i.e., that inherit one another. In other words, these functions can operate on a class and its subclasses, rather than over a type and its subtypes. For this, we use what we call ***f-bounded quantification*** [Canni89b]. F-bounded quantification specifies that any implementation that was derived from a parent is an acceptable type for a function:

```
function foo[Win : INHERITS[Windo]](w1 : Win, w2 : Win) returns Void
[
  w1. insert (w2);
];
```

This function, foo, type checks because w1 and w2 will always have compatible implementations. INHERITS[Window] guarantees that both variables will either be Windows or Presenters but not one of each:

```
foo[Window](some_window, another_window);
foo[Presenter](some_presenter, another_presenter);
```

CONCLUSIONS

This article has shown how contravariance affects object-oriented programming. We have seen that contravariance only comes into play when subtypes and higher-order functions are involved but that these are the exact conditions under which all object-oriented programming languages must operate. We have seen how overloading can be used to alleviate the problems associated with contravariance, but that it carries its own problems. Finally, it has been suggested what a better programming language might look like, one in which parametric polymorphism and the separation of implementations and interfaces plays a crucial role. These ideas can be used to make object-oriented programming both safer and more expressive. ■

REFERENCES

[Bobro88] Bobrow, D., L. DeMitchiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp object system specification, *SIGPLAN Notices*, Special Issue, September 1988.

[Canni89a] Canning, P., W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming, *OOPSLA '89 Proceedings*, 1989, pp. 457–467.

[Canni89b] Canning, P., W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1989, pp. 273–280.

[Cook90] Cook, W., W. Hill, and P. Canning. Inheritance is not subtyping, *POPL '90 Proceedings*, 1990, pp. 125–135.

[Goldb83] Goldberg, A. and D. Robson. *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, Reading MA, 1983.

[7] It is possible that the explicit type application (e.g., to Employee or Manager) at the call site can be eliminated. This is because in most cases it can be inferred from the arguments given that we know the function's signature.

Multilevel secure object-oriented data model — issues on noncomposite objects, composite objects, and versioning

by Bhavani Thuraisingham
The MITRE Corporation, Burlington Road, Bedford, MA 01730

I. INTRODUCTION

Object-oriented systems are gaining increasing popularity due to their inherent ability to represent conceptual entities as objects, which is similar to the way humans view the world. This power of representation has led to the development of new generation applications such as computer-aided design/computer-aided modeling (CAD/CAM), multimedia information processing, artificial intelligence, and process control systems. However, the increasing popularity of object-oriented database management systems should not obscure the need to maintain security of operation. That is, it is important that such systems operate securely to overcome any malicious corruption of data as well as to prohibit unauthorized access to and use of classified data. For many applications, it is also important to provide multilevel security. Consequently, multilevel database management systems are needed to ensure that users cleared to different security levels access and share a database with data at different security levels in such a way that they obtain only the data classified at or below their level.

In a recent article in this journal [Thura90a], we discussed the multilevel security issues of an object-oriented database system and described a simple multilevel object-oriented data model. Like this model, most secure object-oriented data models developed since then (see, for example, [Keefe89, Thura89, Mille90]) have considered only the simple attributes of an object. For example, the title, author, publisher, and date of publication are simple attributes of a book. Such attributes can also be easily represented by a relational model. In contrast, the book cover, preface, introduction, various chapters, and references form the components of a book and cannot be treated as simple attributes of an object. The book, consisting of these components, has to be collectively treated instead as a *composite object*. This was addressed by Kim et al. [Kim87, Kim88] in a nonmultilevel secure environment. Composite objects involve the IS-PART-OF relationship between objects. This relationship is based on the notion that an object *is part of* another object. Note that it is not possible to treat composite objects using a relational model without placing a tremendous burden on the application program to maintain the structure of the complex structures, thus conferring upon the object model another advantage over the relational model.

Hypermedia systems, CAD/CAM systems, and knowledge-based systems are inherently more complex by their very nature and, therefore, can be handled effectively only if their components are treated using composite objects. For example, in hypermedia systems each document is a collection of text, graphics, images, and voice and needs to be treated as a composite object. In a CAD/CAM system, the design of a vehicle consists of designs of its components such as chassis, body, trunk, engine, and doors. Knowledge-based systems are being applied to a wide variety of applications in medicine, law, engineering, manufacturing, process control, library information systems, and education. These applications need to process complex structures. Therefore, support for composite objects in knowledge-based applications is essential.

In many object-oriented applications, such as Hypermedia systems and CAD/CAM, it is necessary to maintain documents and designs that evolve over time. In addition, alternate designs of an entity should also be represented because of the need for choice. If security has to be provided for these applications, then some form of version management should be supported by secure database systems. Another advantage to providing version management in secure applications is the uniform treatment of polyinstantiation and versioning. Note that for many secure applications it may be necessary to support polyinstantiation where users at different security levels have different views of the same entity. Polyinstantiation can be regarded as a type of versioning that cuts across security levels. Therefore, design of the version management component of an object-oriented data model can also be extended to include polyinstantiation.

In this article, we will continue with our investigation on multilevel security in object-oriented database systems and explore the issues on noncomposite objects, composite objects, and versioning. The organization of this paper is as follows: In Section 2 we discuss the issues involved in supporting noncomposite objects in a multilevel environment. Issues on composite objects are described in Section 3. Version management is discussed in Section 4. The paper is concluded in Section 5.

We assume that the reader is familiar with concepts in object-oriented database systems. For a discussion on object-oriented data model concepts such as noncomposite objects, composite objects, complex objects, IS-A hierarchy, and IS-PART-OF hierarchy, we refer to the ORION data model described in [Baner87, Kim87]. We also assume that the reader is familiar with concepts in multilevel secure database management systems (MLS/DBMS). In an MLS/DBMS, users cleared at different security levels access and share a database consisting of data at different security levels. The security levels may be assigned to the data depending on content, context, aggregation and time. It is generally assumed that the set of security levels form a partially ordered lattice with Unclassified < Confidential < Secret < Top Secret. An effective security policy for an MLS/DBMS should ensure that users only acquire the information at or below their level. An overview of multilevel database management systems was given in [Thura90a]. A useful starting point for concepts in multilevel database management systems is the Air Force Summer Study Report [AirFo83].

2. NONCOMPOSITE OBJECTS IN MULTILEVEL DATABASES

Various approaches can be taken to handle noncomposite objects, which are objects with no composite instance variables. In this section, we discuss the various issues involved in handling the noncomposite instance variables of the model at the conceptual level. In Section 2.1, we discuss the basic assumptions of the model and in Section 2.2 we describe how noncomposite variables may be handled.

2.1 BASIC ASSUMPTIONS OF THE MODEL

The entities of classification in an object-oriented data model are the objects. That is, the instances, instance variables, methods, and classes are assigned security levels. The properties C1 to C4 discussed below are the basic security properties that are enforced:

C1. If *o* is an object (either an object-instance, class, instance variable, or method) then there is a security level *L* such that Level(*o*) = *L*.

C2. All basic objects (example, integer, string, boolean, real, etc.) are classified at system low.

C3. The security levels of the instances of a class dominate the security level of the class.

This property is meaningful because it makes no sense to classify a document at the Secret level while the document class that describes the structure of a document is at the Top Secret level. On the other hand, a Secret document class could have Secret and Top Secret document instances:

C4. The security level of a subclass must dominate the security level of its superclass.

This property is meaningful as it does not make sense to classify all documents as Secret and an English document to be Unclassified.

We assume that the following security policy is enforced—subjects (e.g., processes) and objects (e.g., classes, instances, instance variables, methods, composite links, etc.) are assigned security levels:

1. A subject has read access to an object if the subject's security level dominates that of the object.
2. A subject has write access to an object if the subject's security is equal to that of the object.
3. A subject can execute a method if the subject's security level dominates the security level of the method and that of the object with which the method is associated.
4. A method executes at the level of the subject who initiated the execution.
5. During the execution of a method *m1*, if another method *m2* has to be executed then *m2* can execute only if the execution level of *m1* dominates the level of *m2* and the object with which *m2* is associated.
6. Reading and writing objects during method execution are governed by the properties 1. and 2.

2.2 NONCOMPOSITE INSTANCE VARIABLES

In this section, we describe some of the alternate security properties that may be enforced on the noncomposite instance variables (composite instance variables are discussed in Section 5). A similar argument can also be applied to handling methods. However, in this article we focus on structural aspects of an object-oriented data model, only, and not on the operational aspects. Therefore, we do not discuss methods in this article. Also, note that any reference to instance variables in this section implies noncomposite instance variables.

Two ways to assign security levels to instance variables are as follows:

C5. The security level of an instance variable of a class is equal to the security level of the class.

C5*. The security level of an instance variable of a class dominates the security level of the class.

If C5 is enforced, then it is assumed that the objects are single level. This is the assumption made in [Thura89a, Mille90] among others. If C5* is enforced, then it is assumed that an object is multilevel. This is the assumption made in [Keefe89], among others. Note that we consider an object to be multilevel if its properties are classified at different security levels. We discuss each approach in the following two subsections. It should be noted that our main focus is on the representation of the real world entities at the con-

ceptual level. Therefore, we do not address the issues involved in the physical representation of the real world entities.

2.2.1 Single-level objects

If security property C5 is enforced, then the objects are assigned a single level. That is, instance variables have the same security level as that of the class with which they are associated. Therefore, if a document class is Unclassified, then its instance variables, say, title, author, publisher, and publication date are also Unclassified. Suppose a document also has a sponsor who funded its production and the fact that there is such a sponsor must be kept Secret. This means that the document has an additional instance variable that should be Secret. However, the security property C5 will not permit such an instance variable to be associated with a document. There are two solutions for this scenario. One is to create a different document class at the Secret level that has title, author, publisher, publication date, and sponsor as its instance variables (Fig. 1(a); note that the Secret structures are darkened). Note that for every document instance of the Unclassified class there will be a document instance of the Secret document class. Both instances will have the same values for the attributes title, author, publisher, and publication date. The instances of the Secret document class will have the additional attribute of sponsor.

The second solution is to create a Secret subclass of the Unclassified document class (Fig. 1(b)). The Secret subclass inherits all the instance variables of document. It has sponsor as an additional instance variable. Note that for every document instance of the Unclassified superclass there will be a document instance of the Secret subclass. Both instances will have the same values for the attributes title, author, publisher, and publication date. The instance of the subclass will have the additional attribute sponsor.

The instance variables of an object can be regarded as links emanating from the object. The values pointed to by the links are also objects. Although the instance variables of a class have the same security level as that of the class, it does not necessarily mean that an instance variable of an instance of a class must have the same security level as that of the class. This is because property C3 assumes that the security level of an instance dominates the security level of the class. Therefore, if the class is Unclassified and its instance is Secret, then the instance variables associated with this instance must also be Secret. Note also that it does not make sense to classify an instance variable of this instance at a Top Secret level because a Secret user knows that there is such an instance variable. Note also that the level of the object pointed to by

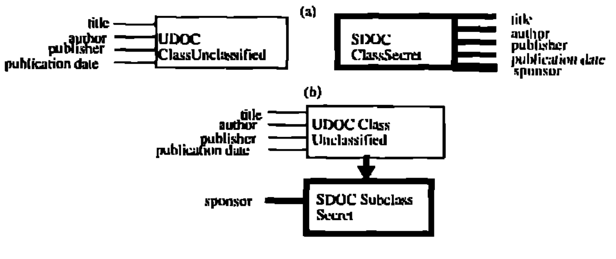


Figure 1. Class/instance variable classifications.

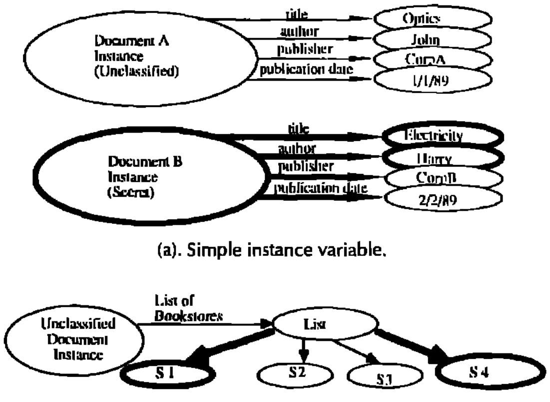


Figure 2. Relationship between instance variables and their values.

the instance variable link (i.e., the value of the instance variable) must be dominated by the level of the link. Therefore, we have the following security properties on instance variables of objects:

$C6_1$. The level of the instance variable of an object must be the same as that of the object.

$C6_2$. The level of the value of an instance variable must be dominated by the level of the instance variable.

$C6_3$. If the instance variable c of an object is a complex instance variable, the security level of c is L, and if $o1$, $o2$, . . . on are the objects that form the value of the instance variable c, then the security levels of $o1$, $o2$, ...on, are dominated by L.

Figure 2(a) illustrates two instances of an Unclassified document class. Note that the Secret document's title and author instance variable values are Secret. The remaining values are Unclassified. Figure 2(b) shows how complex instance variables may be modeled.

Next let us examine how polyinstantiation could be handled (note that by polyinstantiation we mean users at different levels having different views of the same entity—for a discussion on polyinstantiation in relational systems we refer to [Stach90]). Consider the Unclassified document shown in Figure 3(a). This document is Unclassified. It has instance variables title, author, publisher, and publication date. The publisher instance variable link points to NIL because it assumes that an Unclassified user does not know the publisher's name. Let us assume that a Secret user knows of the publisher. Also, the Secret subjects know that the real author of the document is James and not John. There are two ways to handle polyinstantiation. In the first approach, a new Secret document instance is created with attributes as shown in Figure 3(b). Note that in addition to the attributes specified, an attribute such as document-ID will also be necessary to relate the two objects. In the second approach, the polyinstantiated values are attached to the Unclassified document instance as shown in Figure 3(c).

One of the advantages of enforcing the security property C5 is that single-level objects can be mapped into single-level seg-

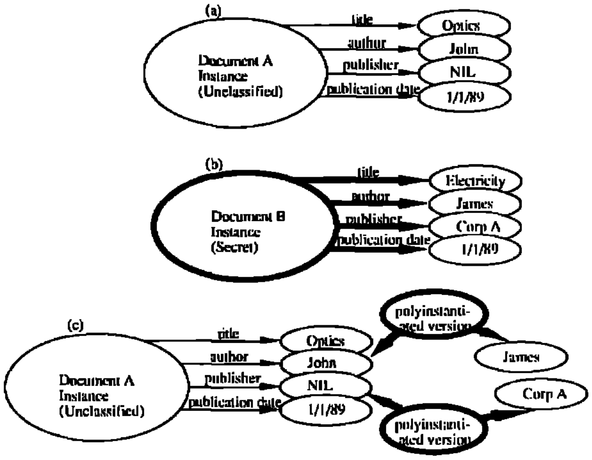


Figure 3. Polyinstantiated objects.

ments or files in a straightforward manner. As a result, traditional security policies (such as the Bell and LaPadula security policy [Bell75]) can be used to control access to the single-level objects. This way, systems with higher levels of assurance can be developed (for a discussion on assurance we refer to [Trust85]). A disadvantage with this approach is that the conceptual representation may not model the real world accurately. This is because in the real world multilevel objects do exist. That is, there could be individuals whose properties are classified at different security levels. A user's view of the database should usually model the real world closely.

2.2.2 Multilevel objects

If we enforce the security property C5* instead of C5, then the objects could be multilevel. That is, the instance variables of the object could have different security levels. Note that in this approach the security level of the instance variables of a class could dominate the security level of the class. Therefore, the document shown in Figure 3 could be represented by the structure in Figure 4.

The instances of UDOC could be multilevel objects. For example, for each Unclassified document instance the instance variables title, author, publisher, and publication date are Unclassified. The instance variable sponsor is Secret. Also, the security level of the value of an instance variable must dominate the security level of the instance variable. That is, the following security properties are enforced:

C6*1. The level of the instance variable of an object dominates the level of the object.

C6*2. The level of the value of an instance variable must be dominated by the level of the instance variable.

C63. If the instance variable *c* of an object is a complex instance variable, the security level of *c* is *L*, and if *o1*, *o2*, ... on are the objects that form the value of the instance variable *c*, then the security levels of *o1*, *o2*, ... on, are dominated by *L*

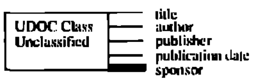


Figure 4. Multilevel instance variables.

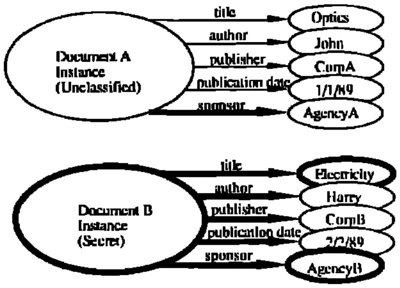


Figure 5. Unclassified and Secret document instances.

Figure 5 illustrates Unclassified and Secret documents that belong to the Unclassified document class of Figure 4. Note that by an Unclassified document we mean that the structure that represents the document is Unclassified. It could, however, have Secret components. Polyinstantiation could be handled either by creating a new object at a different security level or by polyinstantiating the value of an instance variable (see the discussion associated with Fig. 3).

An advantage of enforcing the security property C5* is that it models the real world closely. A disadvantage is that multilevel objects may have to be decomposed into single-level objects that could then be stored in single-level segments or files to provide higher levels of assurance. With such a decomposition, the performance advantages of storing related objects in clusters could be lost. The issues involved in providing performance as well as assurance need to be investigated further.

3.COMPOSITE OBJECTS IN MULTILEVEL DATABASES

In this section, we discuss the various issues involved in supporting composite objects in a multilevel environment. In Section 3.1, the security properties of composite objects are discussed. Representations of composite objects are discussed in Section 3.2. In Section 3.3, some theoretical properties of composite objects are discussed. Composite links connecting a composite object to its components are described in Section 3.3. In particular, the grouping of composite links and its formal semantics are described.

3.1 SECURITY PROPERTIES OF COMPOSITE INSTANCE VARIABLES

A composite object has a composite instance variable. Like noncomposite instance variables, composite instance variables are also assigned security levels. Also, there are two ways to assign security levels to composite instance variables. They are:

C7. The security level of the composite instance variable is the security level of the class with which it is associated.

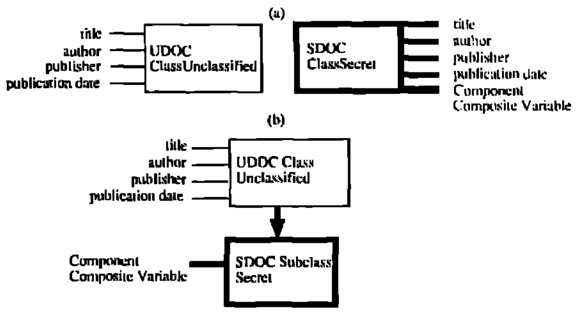


Figure 6. Composite instance variable — approach 1.

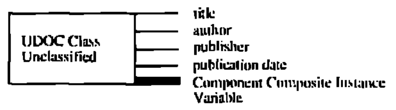


Figure 7. Composite instance variable — approach 2.

C7*. The security level of a composite instance variable dominates the security level of the class with which it is associated.[1]

Figure 6 illustrates an example of security property C7 being enforced. Here, the composite instance variable (which describes the components of an object) of a class is assumed to be Secret. The noncomposite instance variables are Unclassified. The solution is to create an Unclassified class with the noncomposite instance variables and either create a new Secret class with the noncomposite as well as the composite instance variables (Fig. 6(a)) or create a new Secret subclass of the Unclassified class with the composite instance variable (Fig. 6(b)). Note that for every instance of the Unclassified class there is an instance of the Secret class. The Secret instance has the same values for the noncomposite instance variables of the Unclassified instance. In addition, the Secret instance will have a value for the composite instance variable.

Figure 7 illustrates the same example in which the security property C7* is enforced. That is, only one Unclassified class is created. Its composite instance variable is classified at the Secret level. The noncomposite instance variables are Unclassified. Note that for each Unclassified instance of this class the noncomposite instance variables are Unclassified. The composite instance variable is Secret. For a Secret instance of this class, all instance variables (noncomposite and composite) are Secret.

3.2 REPRESENTATION OF COMPOSITE OBJECTS

3.2.1 Alternatives

In this section, we discuss the alternative representations of composite objects. These representations are not affected by the security property enforced on the composite instance variables (i.e., either C7 or C7*). However, the following security property, which describes the relationship between the composite instance variable and the composite links, is enforced:

C8. The security level of a composite instance variable of an object is dominated by the security level of the composite links (or references) from the composite instance variable to one of the components of the composite object.

[1] Note: compare C7 and C7* with the respective properties C5 and C5*.

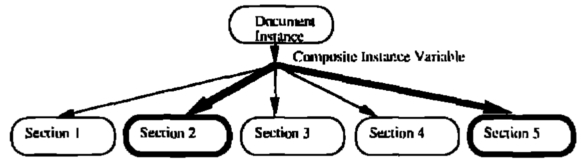


Figure 8(a). Multilevel composite object.

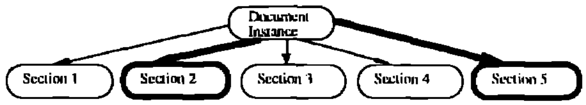


Figure 8(b). Multilevel composite object—alternate notation.

For example, the composite document instance shown in Figure 8(a) is Unclassified. It also has an Unclassified composite instance variable. The composite links connecting it to the component documents must be either Unclassified or higher. Note that had the composite instance variable been Secret (which could be a possibility if C7* is enforced) then the composite links must be Secret or higher. This is a reasonable assumption as in the real world there are cases where Unclassified documents have Secret components. Secret users can read both the Unclassified and Secret components while the Unclassified users can read only the Unclassified components of the document.

A simple approach to handling composite objects would be to assign the same security level to all of the components of such objects. This is not useful because in the real world an object (such as a document) may consist of components (such as sections) at various security levels. If all of the components of a composite object are assigned the same security level, then different documents identifying the same document entity have to be created at the various security levels. This scenario is illustrated in Figures 8, 9, and 10. In Figure 8(a), a multilevel document is represented as it is in the real world. Figure 8(b) shows an alternative notation (not an alternative representation) for the same representation. That is, in Figure 8(b) the composite instance variable is not shown explicitly. Only the composite links are shown in this figure. It is implicitly assumed that the security property C8 is satisfied. This alternative notation is used for convenience, and from now on we assume this notation. In Figure 9, the Unclassified version of the document is represented. In Fig-

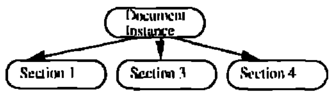


Figure 9. Unclassified version of the multilevel composite object.

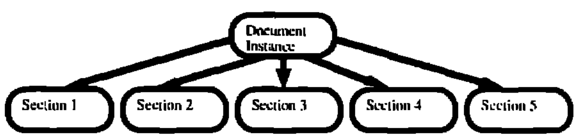


Figure 10. Secret version of the multilevel composite object.

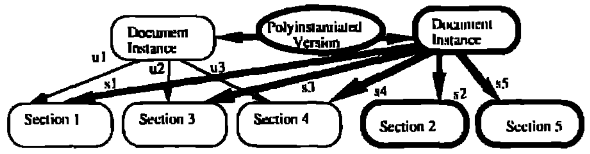


Figure 11. Sharing among polyinstantiated composite objects.

ure 10, the Secret version of the document is represented (note that the darkened structures represent the entities classified at the Secret level).

An alternate approach to representing the composite document of Figure 8 is shown in Figure 11. In this alternate approach, the security level of an object dominates the security level of all of its components. That is, the Secret version of the Unclassified document shares the Unclassified sections with the Unclassified version of the document. The Secret version of the document consists of some additional Secret sections. Note that the polyinstantiated version link between the Unclassified document instance and the Secret document instance can be implemented in various ways. The important point here is that there is some way for a Secret user to know that the Secret document instance is actually a polyinstantiated version of the Unclassified document instance.

Although the complete duplication of the document at different security levels is avoided in the representation of Figure 11, a new document instance at the Secret level still has to be created. A third alternative is to represent the document exactly as it is represented in the real world (see Fig. 8). That is, an Unclassified document could have Secret as well as Unclassified sections. The Secret sections are erased from the view of the Unclassified users. The Secret users can go elsewhere and obtain the Secret sections only. This way, it is not necessary to create a new document instance at a different security level.

3.2.2 Object sharing

Object sharing is an important requirement for hypermedia and CAD/CAM applications. For example, it may be necessary for various sections and paragraphs to be shared between different documents. In a multilevel environment, it is possible for different documents to be at different security levels but share sections and paragraphs. This scenario is illustrated in Figure 12. Object sharing is addressed in more detail in the discussion on composite links given later.

3.2.3 Polyinstantiation

As described earlier, it is possible for two users at different security levels to have different views of the same entity. For example, it is possible for an Unclassified section of a document to be just

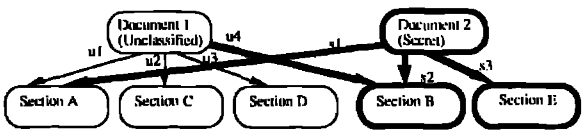


Figure 12. Two documents at different security levels.

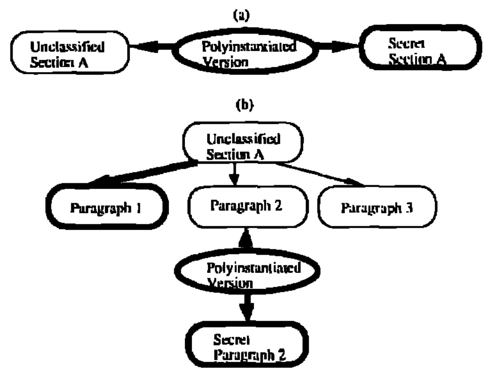


Figure 13. Granularity of polyinstantiated object.

a cover story to a more sensitive version. Polyinstantiation can occur at different stages. At one extreme, one can have the whole document polyinstantiated. At the other extreme, one has a word or a figure polyinstantiated. Figure 13 shows two ways of polyinstantiating sections of a document. In the first approach, the Unclassified section is polyinstantiated at the Secret level (Fig. 13(a)). In the second approach, the cover story is compared with the actual version. If possible, the actual version is decomposed into paragraphs. The sensitive paragraphs are classified at the Secret level. The remaining paragraphs are Unclassified. If an Unclassified paragraph contains false information, then it can be polyinstantiated at the Secret level (Fig.13(b)).

To reduce the amount of polyinstantiated objects, the objects could be decomposed into smaller units, as much as possible, and the smaller units could be polyinstantiated if necessary. It should be noted that polyinstantiation is still a research issue in multilevel database systems. The issues involved in handling polyinstantiation in object-oriented systems are discussed in Section 4 where we regard polyinstantiation as a special form of versioning.

3.3 COMPOSITE LINKS

A composite link is a link that connects a composite object with one of its components. A composite link is also assigned a security level. Figure 14 illustrates possible composite links from a composite object O to one of its components M. We assume that the links are bidirectional. That is, for each link P, there is link P' in the reverse direction. The following security property is enforced:

C9. Let P be a composite link whose reverse link is P'. Then Level(P) = Level(P').

Some of the cases shown in Figure 14 are not meaningful. For example, it does not make sense to form an Unclassified link between a Secret composite object and its Secret component. Further, supporting all the cases of Figure 14 will make certain types of links (to be discussed below) difficult to implement. Therefore, we impose the following security property on the composite objects:

C10. Composite link property

If P is a composite link between a composite object O and its component M, then Level(P) $\geq$ l.u.b.{Level(O), Level(M)}.

We also assume that:

Level(P) = Level(Exist(P))

Level(O) = Level(Exist(O))

Level(M) = Level(Exist(M))

where Level(Exist(e)) is the security level of the existence of an entity *e*.

Enforcing the composite link property will permit only the cases illustrated in Figure 14(a) – (e).

In some cases, it may be necessary for composite objects not to share their components. In other cases, it may be necessary for the existence of a component object to be dependent on the existence of the composite object. These considerations have led object-oriented database researchers to define various types of composite links [Kim87]. We review these definitions and discuss how they may be affected due to multilevel security.

Various types of composite links have been studied in the literature [Kim87, Kim88]. A composite link from object O to component M may be either exclusive or shared. If it is an exclusive link, then it is not possible for another composite object O' to have any link to M. If it is shared, then it is possible for other composite objects to have shared links to M.

The links shown in Figures 14(d) and 14(e) cannot be exclusive or shared. Suppose these links are exclusive. An Unclassified user can see the object M, but he will not know that M is a component of a composite object. Therefore, he could add an exclusive or shared composite link P' from another Unclassified object O' to M. This second link violates the exclusive link property. This scenario is shown in Figure 15(a). If the link P is shared, and if the link P' is exclusive, the exclusive property link is violated. This scenario is illustrated in Figure 15(b).

It does not make much sense to make exclusive the links shown in Figures 14(d) or 14(e). This is because the links shown in Figures 14(d) and 14(e) can only be specified by a Secret user. If this user really wants the link to be exclusive, then he could create a Secret object replicating M and impose an exclusive link from O to this new object. However, if the links shown in Figures 14(d) and 14(e) are not allowed to be shared then it will make the model overly restrictive. A possible solution to overcome this problem is as follows. Suppose an Unclassified user wants to define an exclusive link from O to M. He can do so only if M does not already exist. In this case, he can create M and impose an exclusive link from O to M. Now, no other users can have any composite links from any object to M. If M already exists, there is always a possibility of a higher-level object to have a composite shared link to M. Therefore, there cannot be an exclusive link from O to M. If the Unclassified user wants to impose an exclusive link from O to M, then he will have to replicate M and specify the link.

A composite link from an object O to its component M may be either dependent or independent. If it is dependent, then M cannot exist without O provided there is no other object O' that

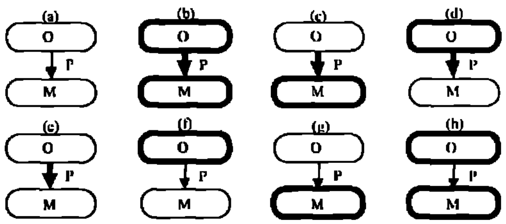


Figure 14. Composite links between objects.

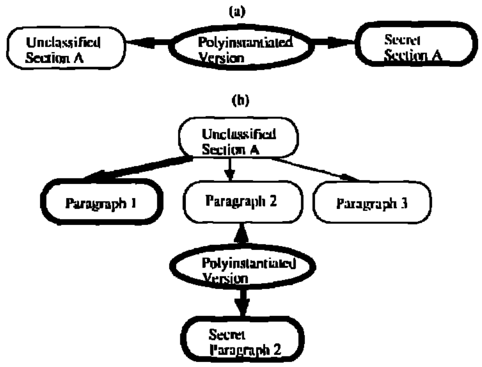


Figure 15. Invalid links.

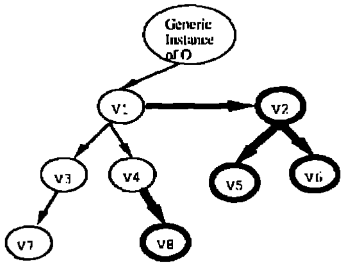


Figure 16. Version derivation hierarchy.

has a link to M. If the link from O to its component M is independent, then M can exist without O.

Note again that the links shown in Figures 14(d) and 14(e) cannot be dependent links. For example, consider the link in Figure 14(e). Suppose this link is dependent. Also assume that no other object has a link to M. Since the object O is Unclassified, an Unclassified user can delete this object. Since he does not know of the existence of P, the object M is not deleted. A Secret user cannot delete M because it is at a lower level. A different problem occurs if the link P in Figure 14(d) is made dependent. If, for some reason, a Secret user wants to delete O, he cannot do so because of the dependent link from O to M. This is because he cannot delete the object M either. He will have to wait until M gets deleted first. Although this situation is not a violation of the dependent link property, it could cause objects that are not in use to consume space. Note that in the link shown in Figure 14(c), the dependent link property can still be enforced. For example, if an Unclassified user deleted O, since he does not know of the existence of the link and also since M is Secret, he will not delete M. However, a consistency checker which runs at the Secret level can detect this problem and delete M to preserve the dependent link property.

4. VERSIONING IN A MULTILEVEL ENVIRONMENT

We first review the model of versions of objects in object-oriented data models such as ORION [Baner87], and then extend the concepts to a multilevel environment. The discussion will be limited to noncomposite objects only. For a discussion on versioning for composite objects in a multilevel environment, we refer to [Thura90b].

A class is defined to be versionable if versions of the instances of the classes can be created. The versions of an instance provide a hierarchy of versions called the version derivation hierarchy. Information about the version derivation hierarchy of an object o is maintained in an object called the generic instance of o.

If the noncomposite instance variable link of an object o' points to a version instance of another object o, then o' is statically bound to o. If the noncomposite instance variable link of an object o' points to the generic instance of another object o, then o' is dynamically bound to o. The system could assign a default version instance of o to be assigned to this link (see Fig. 16).

Let an object o' have an instance variable link to another object o. Suppose a version v of o' is obtained. Then the model should specify as to whether the instance variable link of v should also point to o, or the link is assigned to some other value (e.g., NIL, a generic instance of o, or another version of o).

In a multilevel environment, we identify three types of versions: historical versions, alternate versions, and polyinstantiated versions. Historical versions are due to the evolution of objects over time. Alternate versions store alternate representations of the same entity. Both the historical versions and alternate versions can be handled within as well as across security levels. Polyinstantiated versions are produced when users at different security levels have different views of the same entity. They can only be handled across security levels.

Figure 16 illustrates a version derivation hierarchy of an Unclassified object. Here, versions are created within and across security levels. The generic instance has information on the version derivation hierarchy. Assuming that there are only two security levels, Unclassified and Secret, the generic instance stores Unclassified information of the hierarchy at the Unclassified level and Secret information of the hierarchy at the Secret level.

In this figure, the generic instance of object O has an Unclassified version instance $V1$. $V2$ is a polyinstantiated version of $V1$ at the Secret level. $V3$, $V5$, and $V7$ are historical versions of $V1$, $V2$, and $V3$, respectively. $V4$ and $V6$ are alternate versions of $V3$ and $V4$, respectively. $V8$ could be either a historical or a polyinstantiated version of $V4$ at the Secret level.

The following are possible security properties for versions of noncomposite objects:

C11. Let v be a version instance of the object o. Then Level(v) $\geq$ Level(o).

C12. Let g be the generic instance of an object o. Then Level(g) = Level(o).

C13. Let o' have an instance variable link to version v of object o. Then Level(o') $\geq$ Level(v).

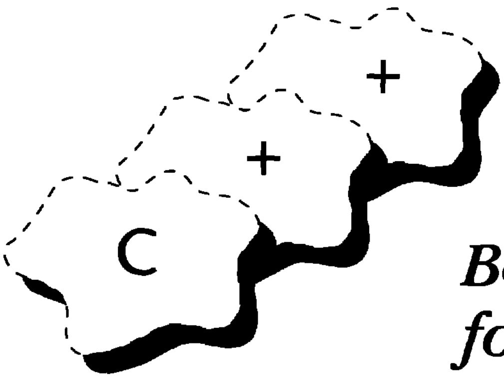
C14. Let o' have an instance variable link to generic instance g of object o. Then Level(o') $\geq$ Level(g).

C15. Let o' have an instance variable link to an object o. Let v' be a version instance of o'. Then the instance variable link of v' points to one of the following:

1. NIL,
2. o, provided Level(v') $\geq$ Level(o),
3. generic instance g of o, provided Level(v') $\geq$ Level(g), and
4. a version instance v of o, provided Level(v') $\geq$ Level(v).

5. CONCLUSION

In this article, we reviewed the developments of security in object-oriented systems and discussed the alternate ways that noncomposite objects could be handled in a multilevel environment. We then focussed on the issues that must be handled in order to pro-

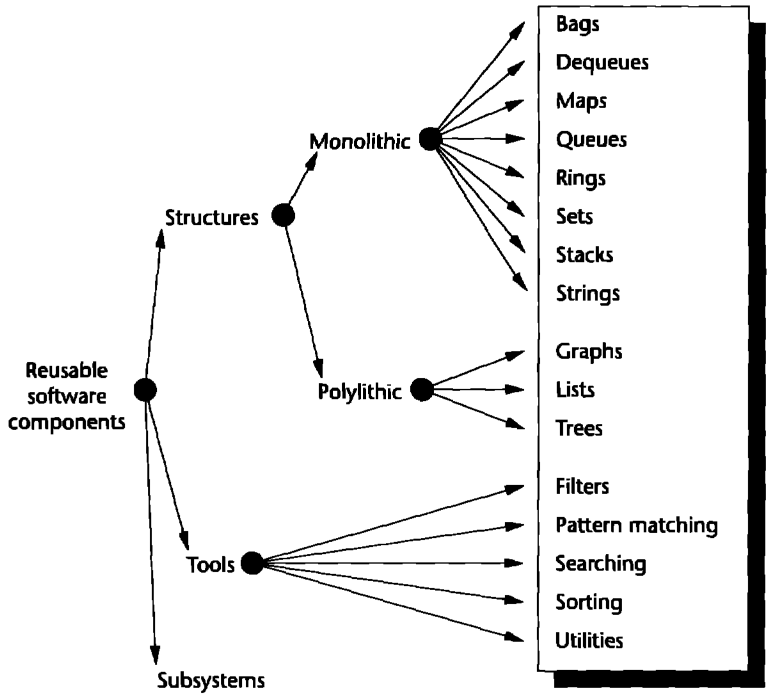


Booch Components for Rapid C++ Development

Rational Consulting introduces the C++ Booch Components®, a domain-independent class library offering the latest in C++ technology: flexibility, extensibility, rapid prototyping, and development.

The C++ Booch Components utilize templates, as defined in the *C++ Annotated Reference Manual* (ARM), to provide maximum versatility. A template preprocessor is included for AT&T C++ version 2.0 compatibility.

The Booch Components are fully supported by Rational Consulting. They are delivered in source code, complete with documentation and test programs.



Organization of the C++ Booch Components

Features

- Simple inheritance lattice of independent structures
- Multiple storage forms
- Multiple concurrency forms
- Exception handling
- Compatibility with other class libraries

Rational Consulting

- Assists management in understanding and adopting advanced software-engineering approaches to improve organizational effectiveness
- Offers consultative problem solving by combining a management perspective with engineering expertise in applying advanced software technologies to real-world software projects
- Provides educational programs for all levels of experience
- Provides software tools in addition to the Booch Components

For more information, contact Brock Peterson at:

Rational Consulting
3320 Scott Boulevard
Santa Clara, CA 95054-3197

Telephone: (408) 496-3700
FAX: (408) 496-3636
Email: blp@Rational.COM

RATIONAL
Object-Oriented
Software
Engineering

vide support for composite objects in a multilevel environment. In particular, the security properties of composite objects, representation of composite objects, and composite links were described. We then discussed issues on version management for a multilevel secure object-oriented database system.

Future research in this area will include the development of a multilevel secure object-oriented data model to support noncomposite objects, composite objects, object sharing, and versioning. The issues discussed in this paper will aid the development of such a model. Another important issue that has not been addressed in this paper is a model for concurrency control. Locking as a concurrency control mechanism for object-oriented database systems was proposed in [Kim88]. However, it is well known that the locking technique causes a covert channel. For example, two users at the Secret and Unclassified levels could request a read lock and a write lock, respectively, to an Unclassified data object. If the Secret user already has obtained the read lock, then the write lock will not be given to the Unclassified user. If the Secret user does not have a read lock then the write lock is given to the Unclassified user. If the Secret and Unclassified users collude, then they can synchronize a series of requests to the Unclassified data object in such a way that from the pattern observed by the granting/denial of the requests to the Unclassified user, information can be covertly passed by the Secret user to the Unclassified user. It has also been argued that the traditional approaches to concurrency control could cause a performance bottleneck. This is because the transactions in object-oriented applications are of very long durations [Kort88]. Therefore, novel concurrency control techniques need to be developed. A preliminary investigation on concurrency control in multilevel object-oriented systems is reported in [Thura90b].

Once a data model has been developed, the next step will be to focus on the security policy and implementation issues. The objects could be multilevel at the conceptual stage and could be decomposed and stored physically in single-level segments (or files) to obtain higher levels of assurance. However, such an approach loses the advantages of storing composite objects in clusters (which has been strongly recommended for operation in a nonmultilevel environment). Storing a composite object together with its components in clusters greatly enhances the performance of database systems [Kim87]. Therefore, it is important to conduct research on the issues involved in enhancing the performance of the system, but at the same time provide higher levels of assurance.

Finally, the design of a multilevel secure object-oriented database system should be based on the data model and security policy that was developed. Such a design should provide the support for query processing, schema management, dynamic schema evolution, update processing, and transaction management and should handle integrity as well as security constraints. Many of these functions are still research topics in object-oriented database systems. Therefore, much remains to be done before multilevel object-oriented database management systems can be developed. ■

ACKNOWLEDGMENT

The authors gratefully acknowledge the Rome Air Development Center (RADC) for sponsoring this work under contract F19628-89-C-0001. We thank Joe Giordano of RADC for his support and encouragement throughout this project. We thank John Faust of RADC for monitoring the project. We thank Maureen Cheheyl for her comments.

REFERENCES

[AirFo83] Air Force Studies Board, Committee on Multilevel Data Management Security, *Multilevel Data Management Security*, National Academy Press, 1983.

[Baner87] Banerjee, J. et al. Data model issues for object-oriented applications, *ACM Transactions on Office Information Systems*, 5(1), 1987.

[Bell75] Bell, D. and L. LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpretation*, Technical Report No: ESD-TR-75-306, Hanscom Air Force Base, Bedford, MA, 1975.

[Keefe89] Keefe, T., W. T. Tsai, and M. B. Thuraisingham. SODA—a secure object-oriented database system, *Computers and Security*, 8(5), 1989.

[Kim87] Kim, W. et al. Composite object support in an object-oriented database system, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Orlando, FL, October 1987.

[Kim88] Kim, W. et al. *Composite Object Revisited*, MCC Technical Report, ACA-ST-387-88, 1988.

[Kort88] Kort, H. et al. On long-duration CAD transactions, *Information Sciences*, 46, 73-107, 1988

[Mille90] Millen, J. and T. Lunt. Security for knowledge-based systems, *Proceedings of the Workshop on Object-Oriented Database Security*, Karlsruhe, West Germany, April 1990.

[Stach90] Stachour, P. and M. B. Thuraisingham. Design of LDV—a multilevel secure database management system, *IEEE Transaction on Knowledge and Data Engineering*, 2(2), 1990.

[Trust85] *Trusted Computer Systems Evaluation Criteria*, Department of Defense Document 5200.28-STD, 1985.

[Thura89] Thuraisingham, M. B. Mandatory security in object-oriented database management systems, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA), New Orleans, October 1989.

[Thura90a] Thuraisingham, M. B. Security in object-oriented database systems, *Journal of Object-Oriented Programming*, 2(6), 18–25, 1990.

[Thura90b] Thuraisingham, M. B. *Issues on Developing a Multilevel Secure O-O Data Model*, Technical Report , The MITRE Corporation, Bedford, MA, 1990.

Delegation in C++

by Ralph E. Johnson and Jonathan M. Zweig

Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801

DELEGATION IS OFTEN VIEWED as a language feature that replaces inheritance. However, it can also be viewed as a relationship between objects that can be implemented in any object-oriented language. It is a useful programming technique that ought to be in the toolbox of every object-oriented programmer. This article shows an example of how to use delegation in C++.

DELEGATION AS A LANGUAGE FEATURE

A few object-oriented programming languages replace class inheritance with delegation between objects [Liebe86, Ungar87]. This is usually part of a language design that eliminates classes, focusing instead on concrete objects. Delegation provides the power of inheritance but also makes it possible to inherit state as well as behavior and to change the behavior of an object dynamically, which is equivalent to changing the object's class.

Languages based on delegation implement method lookup differently than languages based on inheritance. For example, sending a message to a Smalltalk object causes a search for a method in the class of the object. If it is not found, the search is resumed in the class's superclass, and then the superclass's superclass, etc. The method-lookup algorithm results in subclasses inheriting methods from their superclasses.

On the other hand, a delegation-based language like Self [Ungar87, Chamb89] has no classes, and methods can be stored in each object. Each object can delegate messages to other objects so if method lookup does not find the definition of a message in the receiver then it will look in the objects that the receiver delegates to, in the objects that they delegate to, etc. Thus, an object "inherits" the methods of objects to which it delegates messages.

Inheriting state proceeds analogously. When an object accesses an instance variable, a similar search through the delegatees can be performed in the event that the object does not have such an instance variable itself. Another way of accomplishing this (the one used in Self) is to use messages to access state, allowing the message-delegation semantics to provide state inheritance.

Delegation has a number of advantages over inheritance. Some of these fall into the category of simplifying the programming model. For example, it eliminates the complexity of metaclasses without eliminating the power [Borni86]. It makes it easier to implement one-of-a-kind objects and makes programming more concrete. However, the advantage that we are most concerned with is that delegation makes it easier for objects to change their behavior. This is because a class makes many assumptions about the representation in memory of its instances while a delegatee does not make assumptions about the representation of its delegator. Since it is dangerous to change the class of an object, most object-oriented languages do not allow it but it is easy to change the delegatee of an object. Moreover, a language with static type-checking, such as C++, can ensure that a delegatee will understand all the messages delegated to it.

Delegation provides the power of inheritance but also makes it possible to inherit state as well as behavior and to change the behavior of an object dynamically, which is equivalent to changing the object's class.

Although inheritance and delegation are usually described as alternatives in the design of an object-oriented language, we prefer to think of delegation as a way to implement inheritance when an object needs to be able to change its class. Thus, delegation becomes a programming technique, not necessarily a language feature. An important part of the design of an object-oriented system is deciding the relationships between objects [Wirfs89]. There are a number of different ways that objects can collaborate. One is the whole/part relationship [Blake87]. Another is double dispatching [Hebel90]. We propose delegation as another standard way for objects to collaborate.

Delegation is powerful enough to simulate inheritance while simply forwarding a message does not simulate self *properly.*

DELEGATION VS. FORWARDING

Object-oriented programmers often talk of one object delegating a message to another, but they usually do not mean delegating in the sense used here. It is common for one object to have to collaborate with another to carry out one of its responsibilities. For example, reading a file may require reading data from the disk and displaying a complex picture may require displaying each of its components. In both these examples, an object may have to forward a message to one of its components and this is often mistakenly called delegation.

Delegation is more than just forwarding a message to another object [Liebe86]. Delegation is powerful enough to simulate inheritance while simply forwarding a message does not simulate self properly. (The receiver of a message is called self in Smalltalk and this in C++). When a method in a superclass sends a message to self, message lookup starts in the class of the receiver. Similarly, when a delegatee sends a message to self it must use the original delegator as the receiver.

For example, consider a class Car with a superclass Vehicle. Each vehicle has fuel and is able to calculate how much fuel it needs to move a particular distance. In C++, fuelToMove would be a virtual function of Vehicle so that each of its subclasses can have its own function for calculating fuel loss. Vehicle might have a moveTo(Location) method (function) such as:

```
Vehicle::moveTo( Location aLocation ) {
  distanceToMove = distanceBetween( aLocation, currentLocation );
  fuelNeeded = this->fuelToMove( distanceToMove );
  if ( fuel >= fuelNeeded ) {
     currentLocation = location;
     fuel = fuel - fuelNeeded;
  }
}
```

Sending the moveTo message to a Car will call the function defined in Vehicle. When Vehicle sends the fuelToMove message to itself, it calls the fuelToMove function that is defined in class Car. Thus, a function defined in a superclass will call a function in a subclass.

Suppose that this were implemented by giving each Car an instance variable with a pointer to a Vehicle. Then the Car could respond to the moveTo message by forwarding it to the Vehicle. However, the Vehicle would have to send the message fuelToMove back to the particular Car that forwarded the moveTo message. In fact, the Vehicle would have to send all messages overridden by subclasses to the original receiver of the message, which in this case is the Car. Delegation differs from just forwarding a message in that the delegator continues to play the role of the receiver even after it delegates the message. Thus, messages that the delegatee sends to itself are received by the original delegator, which is likely to delegate them back to the delegatee. Of course, the delegatee can delegate messages to another object just as a class can inherit methods that are inherited from it.

Delegation is implemented by including the original receiver as an extra argument to each delegated message. An original message sets this argument to the receiver of the message, but delegated message sends do not change the argument. This is similar to the way languages like C++ implement virtual function calls, where this is an invisible argument to each method and sending a message (i.e., calling a virtual function) binds this to the receiver of the message. Languages based upon delegation, such as Self, will implement this extra argument automatically and invisibly. However, it is possible to implement delegation in any language by using a particular set of programming conventions.

Languages based on delegation usually emphasize flexibility and so rely on runtime type checking rather than static type-checking. However, delegation itself is quite compatible with static type checking. We will show how to implement delegation in C++, one of the least dynamic (and most efficient) of the object-oriented programming languages. This is important because it shows that delegation is a design technique that can be used with any object-oriented language including ones that are statically typed.

DELEGATION IN C++

Our example is taken from an implementation of the Department

Listing 1. The class TCPConnectionDescriptor delegates many of its operations.

```
class TCPConnectionDescriptor {
protected:
    TCPConduit * myCarrier; // Conduit responsible for this connection
    TCPState * current_state;
    ...
public:
    Return_Code   openConnection();
    Return_Code   closeConnection();
    Return_Code   abortConnection();
    Return_Code   processIncomingMessage(TCPMessage * msg );
    Return_Code   processOutgoingMessage(TCPMessage * msg );
    ...
}
```

of Defense (DoD) TCP/IP protocol suite for the Choices operating system [Zweig90]. A TCP network connection can be in one of several states: closed, listening, established, closing, etc. Its behavior, in the sense both of how it responds to incoming network packets and how it interacts with its user, depends on the state it is in. In fact, the behavior of a connection changes so radically depending on its state that it makes sense to think of its class as changing when its state changes. Thus, we could think of a class ClosedConnection, another class EstablishedConnection, etc. Instead of changing its state, a connection object would change its class. Since C++ does not let an object change its class, this alternative is ruled out and another must be used. Although it is hard to change an object's class, it is easy to change the delegatee of an object since the delegatee is determined by a single pointer. Changing an object's delegatee has the same effect as changing its class because the object will now invoke different functions in response to the same messages. Moreover, a delegated function invocation can cost the same as an ordinary virtual function invocation.

The objects responsible for interpreting and delivering network messages are called *conduits*. Conduits can be connected together in a manner somewhat akin to AT&T UNIX System V Streams processing modules. A conduit can call the function to insert messages into another conduit to which it is connected and can call other functions on it when necessary. For example, an application will open a network connection by obtaining a conduit from the system that is connected to the system's TCP conduit. This conduit may then request that a TCP connection be opened on its behalf. The TCP conduit responds to this request by obtaining a connection descriptor, initializing it with information describing the TCP socket with which the application wishes to connect, and calling the openConnection function on it.

Listing 1 shows an excerpt from the definition of the class TCPConnectionDescriptor, which defines the object that contains all state information about a single network connection. The TCP conduit will respond to user requests to manipulate the connection by calling the openConnection, closeConnection, and abortConnection functions of the connection descriptor. A user sends a network message by calling the appropriate connection descriptor's processOutgoingMessage function. When a TCP conduit receives a message from the network (via the IP conduit), it determines which connection the message is intended for and calls processIncomingMessage on the connection's connection descriptor.

Listing 2. The delegatee is an extra argument to delegated functions.

```
class TCPState {
    ...
public:
    virtual Return_Code    openConnection(
                               TCPConnectionDescriptor * cd);
    virtual Return_Code    closeConnection(
                               TCPConnectionDescriptor * cd);
    virtual Return_Code    processIncomingMessage(
                               TCPConnectionDescriptor * cd,
                               TCPMessage * msg );
    ...
}
```

Listing 3. The definitions of delegated functions are all trivial. The delegatee must refer to the delegator instead of THIS.

```
Return_Code
TCPConnectionDescriptor::processIncomingMessage( TCPMessage * msg )
{
    return current_state->processIncomingMessage( this, msg );
}

Return_Code
TCPState::processIncomingMessage(    TCPConnectionDescriptor * cd,
        TCPMessage * msg )
{
    msg->del();
    cd->increment ErrorCount();
    return( ERROR );
}
```

Since the behavior of a connection depends on its state, the connection descriptor delegates these operations to a TCP state object. The state object will need to call functions on the connection descriptor to determine things like sequence numbers, buffers, and so forth. In fact, each TCP state object behaves as though it *is* a connection-descriptor — except that it sends messages to cd in every case where it would send messages to self in a delegation-based language. Listing 2 shows an excerpt from the definition of the class TCPState.

Listing 3 shows the code for the connection descriptor's processIncomingMessage function, which simply delegates to the appropriate state object. It also shows the default behavior for this function on the part of a TCP state object. Any subclasses of TCPState (such as TCPEstablishedState) that are able to accept incoming messages must reimplement this function. States that do not accept messages from other hosts, such as TCPClosedState, will inherit this default behavior, which rejects the message and returns an error code.

PERFORMANCE

Delegation in C++ is fast, involving no more than two function calls. The first is the operation on the delegator and the second is the operation on the delegatee. The second operation is always a virtual function call. If the first operation is a virtual function call, then delegation has twice the cost of a virtual function call. However, the first operation does not have to be a virtual function call. In our example, all connection descriptors implement processIncomingMessage by delegating it—any subclasses would as well — so the processIncomingMessage function can be implemented inline. Thus, delegation can cost the same as a virtual function call plus the time to dereference one pointer.

Since each TCP state object has no local state (instance variables), it is just used to hold a pointer to a virtual function table. It would be nice not to have to pay the penalty for the indirection through the current_state pointer to access this pointer. This might be accomplished by making current_state be an instance of a TCP state object (rather than a pointer to one), which would get overwritten when the connection's state changes. This does not work, however, since in C++ operations on objects declared locally are never virtual — they are statically assigned at compile time since the exact class of such an object is visible to the compiler. It is conceivable that the compiler might recognize that each TCP state object consists only of a pointer to a virtual function table and perform this optimization though we know of no C++ compilers that will.

EASE OF PROGRAMMING

Since C++ is based on class inheritance, delegation requires more work on the part of the programmer than it does in a delegation-based language like Self. The extra work is required in two places: in defining the delegator and in defining the delegatee.

In Self, adding a method to a delegatee automatically makes it available to the delegator but this is not true in C++. Because we are implementing delegation "by hand," we must write a function in the delegator for each operation that it needs to delegate. The function definitions are all trivial just like the definition in Listing 3. However, this is an overhead for the programmer not present in delegation-based languages.

The overhead is smaller in the delegatee. The operations in the delegatee class must all have an extra parameter to refer to the delegator. Instead of performing operations on self, the delegatee must perform operations on the delegator. These rules are simple, but imply that any class designed to be reused by inheritance must be modified before it can be reused by delegation.

Another problem with this way of implementing delegation is that classes reused by delegation are specialized only for that purpose. In contrast, classes that are reused by inheritance are often useful components on their own. This problem does not occur in a language, such as Self, designed to support delegation.

In general, to inherit state the delegatee must send messages to itself (i.e., the delegator) rather than accessing instance variables directly. Compiler optimizations could remove the performance penalty in most cases, however, since the messages that access instance variables might not need to be virtual functions.

CONCLUSION

It is possible that delegation-based languages will replace class inheritance-based languages as the standard in object-oriented programming. However, it is by no means certain. Classes are very useful in structuring large systems and delegation-based systems need programming environment support to simulate classes. Thus, it is not clear whether it is better in the long run to base a language on delegation and simulate classes or to base a language on classes and simulate delegation.

Regardless of which programming style dominates in the long run, most existing object-oriented languages are based on classes. Programmers using class-based languages should learn how to implement delegation. Delegation may not be needed often, but it is easy to implement and should be one of the techniques available to every object-oriented programmer. ■

ACKNOWLEDGMENTS

The second author was supported in this work by a Ph.D. Fellowship from AT&T Bell Laboratories. Both authors thank Brian Marick and Bill Opdyke, who read and commented on earlier drafts of this paper.

REFERENCES

[Blake87] Blake E. and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk, *Proceedings of the 1987 European Conference on Object-Oriented Programming (ECOOP)*, LNCS 276, Springer Verlag, New York, 1987.

[Borni86] Borning, A.H. Classes versus prototypes in object-oriented languages, *Fall Joint Computer Conference (ACM/IEEE), 1986 Proceedings*, Dallas, November 2–6, 1986, pp. 36–40.

[Chamb89] Chambers, C., D. Ungar, and E. Lee. An efficient implementation of SELF, *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) '89 Proceedings*, New Orleans, October 1–6, 1989, pp. 49–70.

[Hebel90] Hebel, K.J. and R.E. Johnson. Arithmetic and double dispatching in Smalltalk, *Journal of Object-Oriented Programming*, 2(6), 40-44, 1990.

[Lieb86] Lieberman. H. Using protypical objects to implement shared behavior, *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) '86 Proceedings*, Portland, OR, September 29–October 2, 1986, pp. 214–223.

[Ungar87] Ungar, D. and R.B. Smith. Self: the power of simplicity. *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) '87 Proceedings*, Orlando, October 4–8, 1987, pp. 227–242.

[Wirfs89] Wirfs-Brock, R. and B. Wilkerson. Object-oriented design: a responsibility-driven approach, *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) '89 Proceedings*, New Orleans, October 1–6, 1989, pp. 71–76.

[Zweig90] Zweig, J.M. and R.E. Johnson. The conduit: a communication abstraction in C++, *Proceedings of the Second USENIX C++ Conference*, San Francisco, April 9–11, 1990, pp. 191–204.

Real-world reuse

by Mark Lorenz
IBM, Box 60000, Cary, NC 27511

OBJECT-ORIENTED (O-O) developers currently spend much of their time thinking about and working with the hierarchical structure of the classes in the system. Their views of this hierarchy may be through a variety of means including paper- and computer-based presentations.

This article takes a look at how application developers currently work with and view their application classes; how this relates to analysis, design, and the class hierarchy; and how application development can be more effective in the future.

The examples used in this article are based on Smalltalk/V PM, but the concepts apply to all O-O development.

A LOOK AT THE HIERARCHY

Class hierarchies are typically wide and shallow (Fig. 1), which is indicative of the fact that subclassing only goes so far. Further extensions to the system are usually subclasses of classes that are at relatively close proximity to the root (the Object class, shown at level 0).

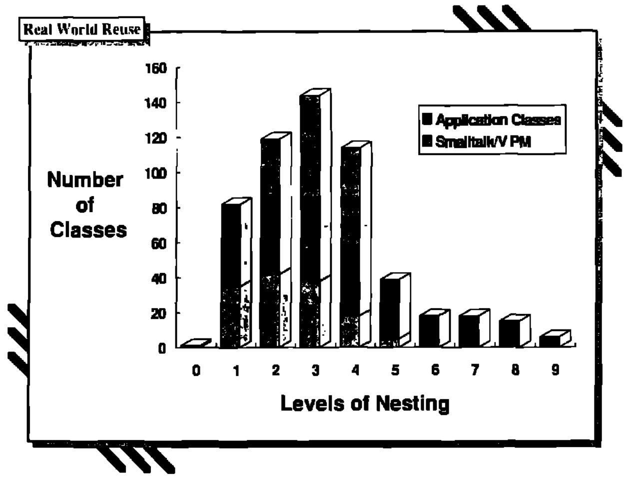


Figure 1.

The system that Figure 1 is based on has over 500 classes defined (Smalltalk/V PM comes with over 100 classes initially). The system has a PersistentObject framework class under Object and application framework classes as shown in Figure 2. These frameworks provide basic implementation functions that can be inherited. For example, a window class subclassed under the application framework class(es) would potentially have menubars with items such as "File" on them already. Some of the pulldown actions, such as "Open...," would be functional up to a point. It is then up to the new subclass to fill in the blanks for the task at hand. Similarly, the persistent object framework would provide functions to allow objects to exist across developer sessions.

In this system, persistent object classes and application windows are subclasses of these framework classes. So, a large number of the new classes defined in the system start 2–4 levels of nesting within the hierarchy (classes that come with the Smalltalk system are concentrated at levels 1–3). As the chart shows, the number of classes nested deeper than this drops off dramatically. In developing the classes in this hierarchy, efforts were put forward to use abstractions where possible. There have obviously been some abstractions found and used in the system, but the vast majority of the classes were located off of the "framework root."

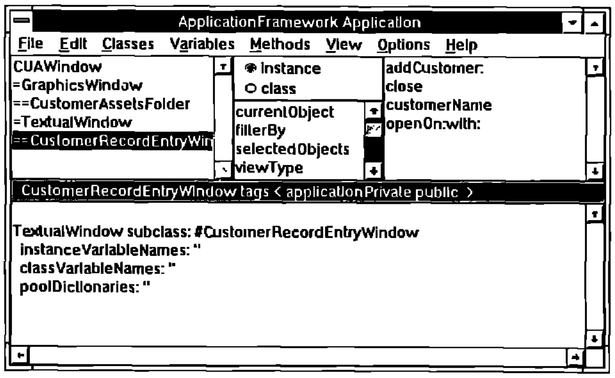


Figure 2. Application framework class structure.

Looking at the Smalltalk system classes, including abstractions such as Collection and Magnitude, and the classes developed in this system, there appears to be a significant limit to the amount of abstractions that can be developed. I have talked to other groups about the depth of their hierarchies with similar conclusions.

ANALYSIS AND DESIGN OF APPLICATIONS

A variety of O-O analysis and design methodologies and notations exist today, including [Booch90, Coad90, Jacob90, Wirfs90] These techniques focus on the objects needed to model the application's problem domain. The notations typically have different types of relationships between the application classes, such as:

1. ***has-a*** — a container relationship to facilitate collaboration.
2. ***is-a*** — a hierarchical relationship for subclassing within an application.
3. ***protocol message*** — application-level messaging.

However, there has been very little written to help the developer decide where to locate classes that are identified. Class positioning has been largely ad hoc, with developers subclassing the root (e.g., Object) or a class perceived as similar to the new class.

APPLICATION

An application could be defined as: *a group of classes that work together to provide some user function, accessible through a public protocol.*

This is the unit of work that an application developer works on at any one time. Treated as a black box object itself the application can be documented and packaged as a salable unit. This could take forms such as source code or executable or dynamic link library (DLL) files.

In Smalltalk/V PM, the user sees classes through the class hierarchy browser (Fig. 3). All classes in the system are listed in hierarchical order in the top left pane. This view focuses attention on the inheritance structure of the reusable classes defined in the system and not the classes in the application itself.

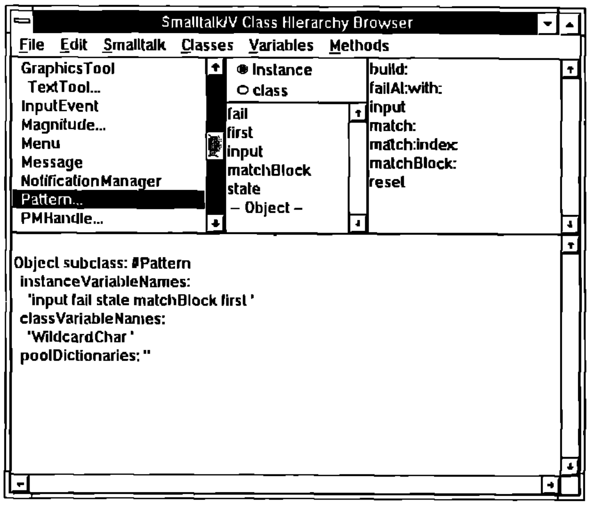


Figure 3. Smalltalk/V PM class hierarchy browser.

The classes in an application are drawn from various places throughout the hierarchy (Fig. 4). These classes collaborate to accomplish the purpose of the application through messages. Usually, the classes hold other classes as instance variables that give them handles to the objects.

An alternative view of classes would show only the application's classes (Fig. 5). In the future, application views will begin to show more information about the structure of applications and how they solve end user requirements. Ivar Jacobson's methodology, e.g., allows the developer to view an end user functional "thread" (called a "use-case" by Jacobson) as it relates to the classes and behaviors of an application.

ABSTRACTION OF SUBHIERARCHIES

During application development, an attempt should be made to create abstractions. For example, in a banking application the analyst may identify a need for SavingsAccount and CheckingAccount classes. The designer may recognize common behaviors needed for these classes and create an abstract Account class (Fig. 6).

This creation of hierarchical relationships between classes is important to the architecture of the application and has benefits in inherited behavior and ease of maintenance. However, placement of this "mini-hierarchy" within the overall hierarchy is rarely a fundamental decision for the application itself but instead relates to implementation decisions that are best put off until later or handled by the system.

POSITIONING CLASSES IN THE HIERARCHY

So where do the application classes go in the hierarchy? There are typically a few basic choices made by the designer/programmer in deciding the location of classes:

1. *Is this a view object (e.g., a window)?* If so, the class should probably be a subclass of the user interface framework classes. In Smalltalk\V PM, these are usually ApplicationWindow, DialogBox, SubPane, or a development shop's application frame-

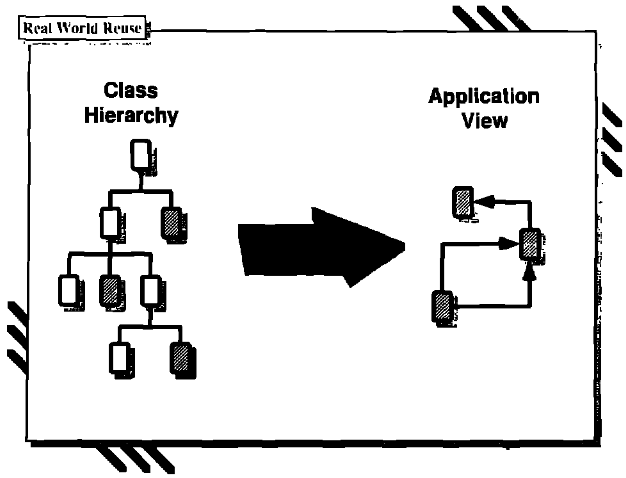


Figure 4.

work. A framework could involve a number of generic function classes (as was shown in Fig. 2).

2. *Does this object persist outside my image (e.g., an instance of a checking account)?* If so, the class should probably be a subclass of the persistent object framework class(es).

3. *Does the class have characteristics that match an existing class very closely or is a superset of the behavior of an existing class?* If so, subclassing is probably called for. An example of this would be CDAccount, as shown in Figure 6. A CDAccount has most of the same behavior as a SavingsAccount, with the exception of withdrawal penalties. So, a CDAccount could be a subclass of SavingsAccount overriding the "withdraw" behavior.

The first two cases focus on inheriting *behavior*, and are O-O design/programming concerns. The desire is to inherit basic functional capabilities such as window or persistence services. These are important concerns, but they are also *implementation* con-

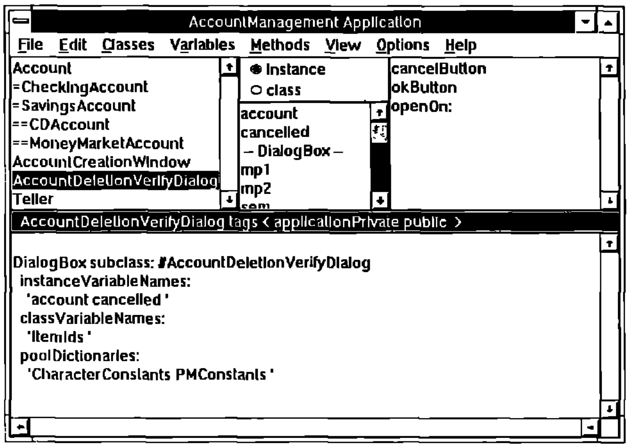


Figure 5. Application browser.

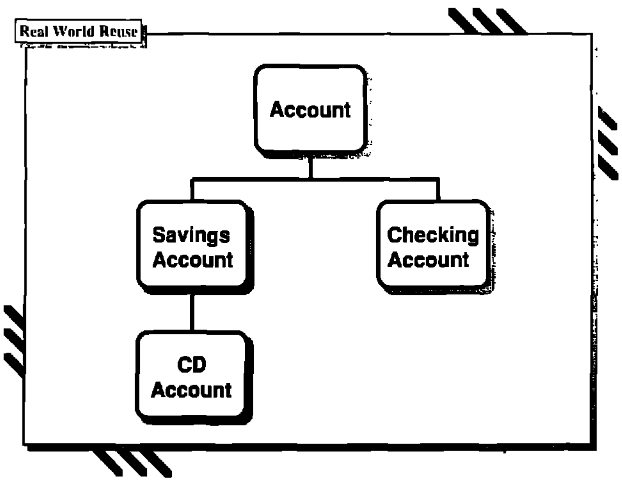


Figure 6.

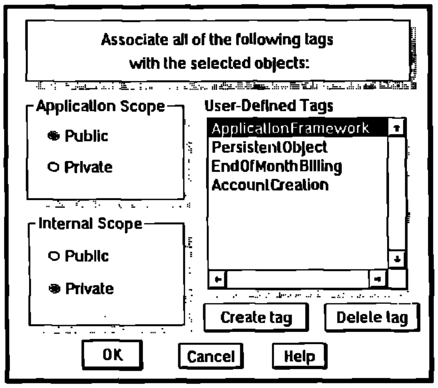


Figure 7. Associating keyword tags to objects.

cerns. They are relationships of convenience (and productivity!). They have little to do with the business' problem domain.

The last case focuses on *types* of business objects and their abstractions, and is an O-O analysis concern. The desire is to create reusable class architectures that model the characteristics of the business. So, the fact that a CDAccount exists and that it acts like a SavingsAccount is fundamental to a bank's business. In fact, the way it goes about this behavior is what gives the Bank its competitive edge.[1]

If none of the listed conditions are met, the class can usually be a subclass of the root class such as Object in Smalltalk/V PM.

TOOLS FOR POSITIONING CLASSES

Categorizing tags and informational descriptions can be captured and used by the system to help the developer find possible classes to subclass. The simplest case is to ask the user the types of questions listed above. A more involved case is to use informational tags necessary for categorization and retrieval of reusable objects to help position classes at implementation time.

By allowing the developer to specify categorization information about classes, the system could suggest the optimal location within the hierarchy without requiring the user to focus on this structure. Figure 7 shows an example dialog to allow the user to tag objects with system- and user-defined keywords.

New objects that are of type window or persistent object are obvious candidates as a subclass of the appropriate framework. The more difficult positioning concerns relate to matching characteristics. Aids in this positioning can be based on:

1. Similarity of public protocols (methods).

   For example, if the new CDAccount class discussed earlier fulfills roles such as "withdraw" and "deposit" it matches these characteristics of the SavingsAccount class and would be suggested as a possible position in the hierarchy.

2. Keyword tags.

[1]Note that this is also the way that frameworks come into existence—data processing abstractions such as TreeGraph, Collection, and Magnitude are created and leveraged by future developers. These are just as important during design and implementation as the business abstractions are during analysis.

For example, if the CDAccount class is tagged by keywords such as "banking," "account," and "savings" it will closely match the user-defined characteristics of a SavingsAccount. The system could certainly provide views of existing keywords and searches of objects tagged by related keywords to help the developer.

SUMMARY AND RECOMMENDATIONS

Subclassing and inheritance are important, but perhaps overemphasized, aspects of an O-O system. Developers will tend, over time, to focus more on *application* and less on *implementation* concerns.

Systems need to include services to capture and utilize developer information about classes to recommend class locations within the hierarchy. ■

ACKNOWLEDGMENTS

The author thanks Pete Dimitrios and Bill Haynes for their comments on the article.

REFERENCES

[Booch90] Booch, G. *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1990.

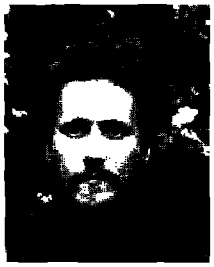
[Coad90] Coad, P. and E. Yourdan. O*bject-Oriented Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Jacob90] Jacobson, I. *Object-Oriented Development in an Industrial Environment*, Objective Systems SF AB, Kista, Sweden, 1990.

[Wirfs90] Wirfs-Brock, R. et al. *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

For free, fast information on the products and services advertised in this issue, consult the advertiser index on page 66.

Understanding constructor initializers in C++



by Andrew Koenig

"WHAT DOES THAT funny syntax with the colon mean?" Every time I have taught a C++ class, at least one person has asked that question. This is often true even *after* I have explained the answer. For some reason, people seem to have a particularly hard time understanding this specific detail. I don't know why; it's not particularly difficult or counterintuitive. Perhaps it is because this is one of the places that C++ carefully distinguishes between things that C does not. I suppose *something* has to be the most commonly misunderstood part of C++, and this just happens to be it.

I am talking, of course, about constructor initializers. For example:

```
class Complex {
public:
    Complex(double x, double y):
        re(x), im(y) { }
    // ...
private:
    double re, im;
};
```

Whenever I show an audience this example, someone is sure to ask me about the purpose of the re(x) and im(y) parts of the constructor and will want to know why I didn't just say:

```
Complex(double x, double y) {
    re = x;
    im = y
}
```

Of course, if I do use the second form, someone is sure to ask me why I don't use the first!

The answers to these questions are all tied up in the difference between assignment and initialization, as well as the different kinds of constructors one can have for a class. To make it all clear, we will have to go over these things in detail. Please be patient if you've seen some of this before.

CONSTRUCTORS

A constructor is a member of a class that is executed to create an object of that class. Strictly speaking, objects of built-in types, such as int, do not have constructors. However, the following presentation will be easier if we pretend they do. Let's pretend, therefore, that objects of built-in type have "constructors" that automatically initialize such objects to zero if they are of static storage class (or part of an object of static storage class) and to an undefined value otherwise. With this generalization, it is possible to state a rule:

- Every object is created by executing a constructor!

Here is a simple example:

```
#include <stream.h>

int x;

main()
{
    int y;
    static int z;

    cout << x << " " << y << " " << z << "\n";
}
```

This program creates three variables named *x*, *y*, and *z* and prints their values. What does it print?

Each of these variables is of type int so the value of each is determined by the "constructor" associated with that type. Because x and z are of static storage class (global variables are always of static storage class although that is not stated explicitly), their "constructors" give them initial values of 0. The initial value of y, on the other hand, is undefined.

COPY CONSTRUCTORS AND ASSIGNMENT

If every object is created by executing a constructor, what about this?

```
main()
{
    int x = 7;
    int y = x;

    x = y;
}
```

The variables x and y are each created and simultaneously given an explicit initial value. The declarations of x and y are requests to *copy* existing values into new objects. Because these are *new* objects, our previous rule says that they must be created by executing constructors. Evidently, then, there must be some kind of constructor that can create an object that is a copy of some existing object; we call that a *copy constructor*. As before, we can simplify the presentation by pretending that even built-in types like int have copy constructors.

In the example above, then, the object x is created by executing its "copy con-

structor," which gives it an initial value of 7, and the object y is created by executing *its* "copy constructor" to give it an initial value that is a copy of the value of x.

Now let's look at the last statement in the example. That statement says to set the value of x equal to the present value of y. Of course, they happen already to be equal, but that doesn't matter. This involves (potentially) changing the value of the object x, ***but it does not create any new objects!*** Because no objects are created, no constructors are called. This operation is therefore fundamentally different from the previous two ***even though the same symbol is used to represent it.*** The act of giving a new value to an object is called *assignment.*

If you are ever uncertain whether a piece of C++ code involves assignment or construction, ask yourself "Is a new object being created here?" If the answer is "yes," then a constructor is involved. If it's "no," then constructors are not involved.

If you are ever uncertain whether a piece of C++ code involves assignment or construction, ask yourself "Is a new object being created here?" If the answer is "yes," then a constructor is involved. If it's "no," then constructors are not involved.

DEFAULT CONSTRUCTORS AND ASSIGNMENT FOR CLASSES

Suppose we write a simple class:

```
struct Point {
    int x, y;
};
```

This class is so simple that it has no private data at all. That explains the choice of struct rather than class to introduce it. Indeed, as written, it is nothing more than a C structure. For that reason, it had better behave the same way as its C counterpart. For example, it must be possible to say things like this:

```
struct Point {
    int x, y;
};

Point zero; // x=0, y=0

main()
{
    Point p, q;
    p.x = 3;
    p.y = 7;
    q = p;
}
```

To preserve C behavior, C++ causes some things to happen automatically:

- A class with no explicit constructors gets an empty constructor automatically.
- A class without an explicit copy constructor gets one automatically.
- A class without an explicit assignment operator gets one automatically.
- These functions, if automatically generated, are recursively defined in terms of the corresponding functions for the members and base classes.

This seems like quite a mouthful but is actually quite simple. In the case of our Point class, e.g., it tells us that we can construct, copy, and assign Point objects and that the meaning of doing so is defined recursively in terms of the corresponding operations for x and y. This is, of course, exactly what happens in C. Thus, in the example above zero.x and zero.y are both initialized to 0 by the automatically generated constructor for the Point class, which is recursively defined in terms of the "constructors" for the members x and y. Similarly, the members of p and q are recursively initialized by their "constructors" to undefined values.

One useful consequence of having automatic constructors of this sort is that it makes it much easier to build simple data structures out of classes others have defined. For example:

```
struct Person {
    String name;
    String address;
    int id;
};
```

One can easily imagine some kind of recordkeeping system with a data structure like this to keep track of people. Such a data structure might reasonably use a String class taken from some library to store names and addresses. Here's a simple example:

```
main()
{
    Person p;

    getrecord(inputfile, p);  //Read into p
    Person q = p;

    //...
}
```

What is the effect of the declaration of p? What initial value does p have? Because the object p is created here, we must execute a constructor, but the Person class doesn't have one. A constructor is therefore created for us by using the String constructor twice and the int "constructor" once. The effect will therefore be to initialize p.name and p.address to whatever the default value is for the String class and leave p.id undefined (because the "constructor" for int says that's the right thing to do).

Similarly, the declaration of q creates an object so it must execute a constructor. Because it is creating an object from another of the same class, the constructor to use is evidently the copy constructor. Because the Person class has no explicit copy constructor, one is generated automatically. That constructor executes the String copy constructor twice (for the name and address members) and the int "copy constructor" once (for the id member). The result is exactly what one might expect: copying p into q has the effect of copying each member of p into the corresponding member of q.

OVERRIDING THE DEFAULTS

Our Person class is a bit of a nuisance to use: every time we create a Person object, we must eventually give a value to each of its members. For example:

```
Person s;
s.name = "Santa Claus";
s.address = "North Pole";
s.id = 31415927;
```

We would like instead to be able to write:

```
Person s("Santa Claus", "North Pole", 31415927);
```

The way to do that, of course, is to give the Person class an explicit constructor, which is most straightforwardly written this way:

```
// first try: not quite right
struct Person {
    String name;
    String address;
    int id;
    Person(String n, String a, int i) {
        name = n;
        address = a;
        id = i;
    }
};
```

This will indeed make possible the second declaration of s shown above. However, this is not quite the right way to go about this for reasons we are about to uncover.

The first problem can be seen by looking again at the rules for default constructors: a class without any explicit constructors gets an empty constructor. We have taken our Person class, which did not have an explicit constructor before, and given it one. That means that the empty constructor it formerly had is no longer there, which in turn means that we can no longer say:

```
Person p;
```

at all! That would be fine were that what we had in mind, but in this case we do not wish to give up the old behavior to acquire the new behavior.

We must therefore explicitly insert the constructor that is no longer being created for us:

```
// second try: still not quite right
struct Person {
    String name;
    String address;
    int id;
    Person(String n, String a, int i) {
        name = n;
        address = a;
        id = i;
    }
    Person() { }
};
```

We have inserted a constructor that does nothing at all. Does that mean that when we say:

```
Person p;
```

we are foregoing initialization of p.name and p.address? That would be a disaster! After all, we know nothing of the workings of the String class. Its author could be counting on all objects of that class being initialized appropriately. To be sure that happens, C++ has another rule:

- If a constructor doesn't say explicitly how to initialize the members or base classes of its class, the default constructors for those members or base classes are used automatically.

That means that the constructor we just added to the Person class:

```
Person() { }
```

actually does three things: it uses the String constructors to initialize the name and address members and the int "constructor" to initialize the id member. This is, of course, exactly the right thing in this case: it gives us an easy way of saying "please preserve the default behavior even though this class has an explicit constructor."

But this analysis exposes a problem in the other constructor:

```
Person(String n, String a, int i) {
    name = n;
    address = a;
    id = i;
}
```

Look again at the last rule. This constructor never says anything about how to initialize name, address, or id. The statements in the constructor are assignments, not initializations, because they do not construct any objects! By the time the constructor begins execution, the name, address, and id members of its object must therefore already exist. Because objects come into existence only through constructors, that means that their constructors have already been executed.

In other words, the effect of the Person constructor above is to construct name and address, "construct" id, *and then to assign new values to* name, address, and id as shown in the constructor body itself. The difference is precisely the difference between:

```
String s = "Santa Claus";
```

and

```
String s;
s = "Santa Claus";
```

The first of these forms is clearly preferable because it gives s the desired value immediately instead of giving it the wrong value first and then correcting it.

CONSTRUCTOR INITIALIZERS

Because of all this, when writing a constructor we need some way to say "give value *x* to member *y* at the time that member is constructed." The syntax for that looks like this:

```
Person(String n, String a, int i):
    name(n),
    address(a),
    id(i) { }
```

The constructor's list of formal parameters is followed by a colon and then a list of *initializers* separated by commas. Each initializer is the name of a member or a base class followed by a parenthesized list of expressions to be used to initialize that member or base class.

... when writing a constructor we need some way to say "give value x to member y at the time that member is constructed."

One might, therefore, read the example above as "To construct a Person from two Strings called n and a and an int called i, construct the Person's name member from n, its address member from a, and its id member from i, and then do nothing." The "do nothing" part corresponds to the empty body of the constructor proper: this particular constructor now does all its work in its initializers.

The entire class definition now looks like this:

```
// third try: this is how to do it
struct Person {
    String name;
    String address;
    int id;
    Person(String n, String a, int i):
        name(n),
        address(a),
        id(i) { }
    Person() { }
};
```

To confirm our understanding, we can add an explicit copy constructor and assignment operator to the Person class that does exactly what the default ones do:

```
// fourth try: equivalent to the third
// but with everything written out explicitly
struct Person {
    String name;
    String address;
    int id;
    Person(String n, String a, int i:
        name(n),
        address(a),
        id(i) { }
    Person() { }
    Person(const Person& p):
        name(p.name),
        address(p.address),
        id(p.id) { }
    Person& operator=(const Person& p) {
        name = p.name;
        address = p.address;
        id = p.id;
        return *this;
    }
};
```

Note how the Person copy constructor makes use of the String copy constructor for the name and address members and the int "copy constructor" for the id member. Note also that the Person assignment operator does not use constructors at all because no new objects are being constructed.

CONCLUSION

It should now be possible to understand the difference between our first two examples. If we write:

```
Complex(double x, double y):
    re(x), im(y) { }
```

we are being formally correct by saying that a Complex object should be constructed by constructing its real and imaginary parts, after which we're done. If instead we write:

```
Complex(double x, double y) {
    re = x;
    im = y
}
```

we are first constructing re and im with their default values (which, because they are of built-in types, are undefined), and then assigning x and y to them.

Because x and y are of built-in types, the conceptual difference between these forms does not translate itself into a practical difference and indeed many compilers are likely to generate identical machine code for both. For that reason, many people (including me) are apt to be careless about distinguishing between assignment and initialization in simple cases like this and that surely doesn't make it any easier to understand when encountering such things for the first time.

The important thing is to be completely clear about the difference between assignment and initialization and to realize that they are expressed differently—especially in constructors. ■

Andrew Koenig is a Distinguished Member of the Technical Staff at AT&T Bell Labs in Warren, NJ. He is working on C++ tools in a department dedicated to reducing the cost of software development. He is also an enthusiastic musician and an instrument-rated private pilot. Koenig can be contacted at Room 4N-R12, AT&T Bell Laboratories, 184 Liberty Corner Rd., Warren, NJ 07059, or through email at attmail!ark or ark@europa.att.com.

The evolution of bugs and systems

by James Rumbaugh

IN WRITING THIS SERIES of columns, I hope to show the value of an object-oriented analysis and design methodology and how to apply it to the solution of problems. I want to show that object-oriented technology is more than just programming and languages. For the most part, I intend to give examples that illustrate various aspects of analysis and design as I have found that a single concrete example is often more illuminating than a broad but abstract theoretical presentation. In presenting these examples, I will use the object modeling technique (OMT) methodology and notation developed by my colleagues and me and described in the book *Object-Oriented Modeling and Design* published recently by Prentice Hall [Rumba91]. In the process, our philosophy of design should become clear as will both similarities and differences in outlook between us and other authors. Keep in mind that developing software (or anything else) is a complex creative task and there is no one best way to do it. Neither our methodology nor any of the others is the final word; they will all evolve as new ideas and new combinations of old ideas are developed. My goal is to get you to use some methodology of analysis and design rather than just sitting down and starting to program.

Object-oriented development provides a seamless path from analysis through design and implementation. You don't have to change notation at each stage of development but this doesn't mean that all stages of development are the same or differ just in the amount of detail. The different stages focus on different aspects of a problem and emphasize different object-oriented concerns. In this column, I will illustrate object-oriented analysis using a simple example. Other stages of the process will be discussed in future columns.

CREEPING BUGS

We will consider an evolution simulation based on a *Scientific American* "Mathematical Recreations" column [89]. The goal is to simulate the evolution of "bugs" in a simple two-dimensional world. The world contains bugs and bacteria, which the bugs eat. The bacteria are "manna from heaven." They appear at random and persist at fixed locations until they are eaten. Bacteria do not spread, age, or reproduce. Bugs move around the world randomly under the control of motion genes. Each bug has a variable position and orientation within the world. For simplicity, time is divided into uniform time steps. During each step, each bug rotates randomly to a new orientation, then moves one unit forward in its new direction. Rotation is controlled by the motion gene, which codes for a probability distribution of rotating by an arbitrary angle from the previous orientation. Initially, the distribution is uniform so a bug performs a random walk. For simplicity, we divide the world into uniform cells with a finite number of angles such as a hexagonal grid with six possible angles. A bug eats any bacteria it finds within its cell, gaining a fixed amount of weight for each meal. Each time step the bug loses a fixed amount of weight to maintain its metabolism. If its weight becomes zero, the bug starves. If its weight exceeds a certain "strong" value, then the bug reproduces by splitting itself into two identical bugs each with half the original weight. Each new bug suffers a single mutation in its motion gene modifying the probability distribution.

If you program this simulation and choose appropriate values for the various parameters so that all the bugs do not die out quickly, over time you observe a kind of evolution. At first the bugs jitter about randomly, but over time they evolve so that they move more or less in straight lines with an occasional turn to the left or right (but not both for any one bug). The explanation is that bugs that move randomly tend to eat up the food supply in one place and starve while bugs that move in lines have a better chance to find new food but they must turn occasionally to avoid getting stuck against the edges of the world.

This problem is well-suited to an object-oriented approach and is fairly simple to program. There is some ambiguity in the specification and many possible extensions can be considered such as carnivorous bugs. I will illustrate my solution to it using the OMT notation. I cannot show all the details that would accompany a full solution of the problem but I hope to touch on the major points, at least.

STAGES OF DEVELOPMENT

To solve a problem, you must identify a problem, describe what you need to do about it, decide how to do it, and then go and do it. These steps are the development stages of *conceptualization, analysis, design,* and *implementation.* Other things you might do include verifying that you actually solved the problem and carefully describing your solution so that someone else could repeat it. These steps correspond to *testing* and *documentation.*

Of necessity, methodology books (including ours) lay out the development process as a sequence of steps. This pedagogical need has been misinterpreted as the infamous "waterfall diagram" showing development as a one-way flow of information through well-defined stages. In practice, the distinction among the stages is not always clear-cut because software development is a creative act that requires some judgment from the practitioner. More importantly, the development of any real system involves a lot of iteration within and among stages, more of a "whirlpool" than a waterfall.

Analysis, design, and implementation could be called "synthetic" stages of development. During these stages, the designer must synthesize a system out of a jumble of potential requirements and parts striving for a result that is both understandable and efficient while solving the problem. During synthesis, it is useful to have a well-defined notation to specify exactly what has been created at any step in the process. The development notation should flow easily from stage to stage so that work will not be lost, ignored, or repeated as the design process proceeds. We claim that an object-oriented modeling notation can be used throughout the development process without a change in notation or reentry of information.

Today I will focus on analysis. The analysis model forms the framework on which the entire design is built and fleshed out.

ANALYSIS

During analysis we identify *what* must be done without saying *how* it will be done. During analysis, we identify the object classes in the problem domain, their significant attributes, and the relationships among objects. We capture this information in an *object diagram.* The object diagram describes a snapshot of information at a point in time.

The first step is to identify object classes and describe them briefly. Table 1 is a data dictionary in which we have identified five object classes from the problem description: Bug, Gene, Bacterium, Cell, and Grid.

Table 1. Data dictionary.

Bug	An organism that inhabits a cell, moves under control of a motion gene, eats bacteria it finds, and reproduces by fission under suitable conditions. The bug dies if it doesn't eat enough.
Gene	A set of discrete values that codes for the probabilistic motion of a bug. Genes are copied and mutated during bug reproduction.
Cell	A discrete location within the grid world that contains (possibly multiple) bugs and bacteria. The cells are uniformly spaced within the grid.
Bacterium	Food for bugs. Each bacterium is worth a specified amount of weight when eaten. Bacteria are created randomly on the grid and persist on the same cell until they are eaten.
Grid	A tessellated world inhabited by bugs and bacteria. Bugs can move to neighboring cells. The edges of the grid block motion.

You should always prepare a data dictionary containing a brief description of every class, attribute, operation, relationship, or other element of a model. A simple name by itself has too many interpretations.

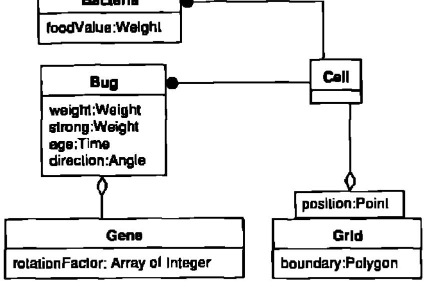


Figure 1.

OBJECT MODEL

Figure 1 shows an *object diagram* for the Bugs simulation. An object diagram is a graphic representation of the classes in a problem together with their relationships, attributes, and operations. Each class is shown as a box with the name of the class in the top part, an optional list of attributes in the second part, and an optional list of operations in the third part. We have omitted operations from the first diagram.

Each bug has a weight and an age, a direction of movement, and a weight at which it is "strong" enough to reproduce. These attributes have been pulled directly from the problem statement. Similarly, each bacterium has a food value. A gene contains an array of rotation factors, each an integer. We want rotation factors to be discrete values subject to quantum mutations; therefore, we have represented them as integers but we have not yet said how a factor value maps into a probability; we must specify this mapping during design. Finally, we have called the world Grid to capture its discrete nature within our simulation. The boundary of the grid is a polygon although in the first version of this program it will likely be a simple rectangle.

More important even than the attributes of an object are its relationships to other objects. Relationships indicate how objects interact, how information flows among them, and how objects can be assembled into a complete system. Relationships affect the organization of the entire system while attributes (and operations) are often used by only a single class. Relationships include association, aggregation, and generalization.

Association is any relationship among the instances of two classes. In most cases, binary associations are sufficient. A binary association is indicated by a line between two classes (or a loop on a single class) with a *multiplicity* symbol at each end to indicate how many of each class may be related to an object of the other class. For example, each cell may contain zero or more bugs and zero or more bacteria. The line between Cell and Bug indicates an association; the black dot next to Bug shows that "many" (zero or more) bugs may be associated with a given cell; the lack of a symbol next to Cell indicates that exactly one cell is associated with a given bug. An association and its two ends may have names but they may be omitted if there is no ambiguity.

Aggregation is a special kind of association indicating a part-to-whole relationship. For example, a gene is part of a bug. The diamond next to Bug on the line from Gene indicates that Bug is the aggregate and Gene is the part. The lack of a multiplicity symbol on either end indicates that each bug contains exactly one gene and each gene is part of exactly one bug. In the case of a one-to-one relationship such as Gene is part of Bug, the two classes could be merged into a single class containing all the attributes, but we choose to distinguish Gene and Bug because they have distinct names in the application domain and a clear separation of properties.

> ***The real essence of an object-oriented analysis is not inheritance but thinking in terms of objects. An object-oriented model is object oriented because the potential to add inheritance to the model is always present.***

A grid is composed of many cells as shown by the aggregation line between Cell and Grid. Each cell has a unique position within the grid that distinguishes it from all other cells. The position is not really an attribute of Cell; rather, it is an attribute of the Cell–Grid association since it defines the position of the cell uniquely with respect to the grid. The association line between Cell and Grid with the box next to Grid is a *qualified association.* The *qualifier* in the little box indicates an index value unique within the qualified class. A grid and a position determine a unique cell; a cell corresponds to a grid and a position. There is a one-to-many relationship between Grid and Cell; there is a one-to-one relationship between the pair (Grid, Point) and Cell.

Why bother to even have a Grid class? After all, the grid is unique within the problem and it seems wasteful to represent associations to fixed global objects. Don't fall for this reasoning. If you build unique global objects into your problem, you will often find that you eventually want to extend the problem to accommodate multiple instances of the "unique" object. Therefore, define a class for each object in the system, even those that you think are unique, and define associations between those classes and other classes that depend on them.

This completes the basic object diagram. It defines a snapshot of a system at a moment in time in terms of objects, their attributes, and their relationships. The goal is to include enough information, and just enough information, to fully define the state of the system and the objects in it without redundancy. Don't show redundant attributes during analysis. For example, we could replace age by birthDate but we would not show both at once because to do so would indicate more freedom than is actually present in the system. Don't show attributes or associations that are derivable from other attributes or associations. For example, don't indicate position as an attribute of Bug; a unique position value can be derived by navigation from Bug to Cell to Grid. Don't show associations between classes as attribute values. For example, we could have an attribute gene within Bug and an attribute bug within Gene, but this again would indicate that the two values could be set independently, which they cannot. Associations should always be used for showing relationships between objects because they are inherently bidirectional; pointers (attribute values referencing other objects) are inherently an implementation concept and do not belong in analysis.

What is not present in this analysis object diagram? First of all, this diagram contains no inheritance (or *generalization,* as the relationship between the classes is called). Some readers will be shocked that I dare to describe an object-oriented problem without using inheritance. It is true that an object-oriented language or notation needs the concept of inheritance to be fully object-oriented. But that doesn't mean that you have to *use* inheritance on every problem. The real essence of an object-oriented analysis is not inheritance but thinking in terms of objects. An object-oriented model is object-oriented because the potential to add inheritance to the model is always present. For example, we could specialize Bug into Herbivore and Carnivore subclasses in the future. Inheritance may or may not be necessary in the analysis of a particular problem; don't think you have to use it all the time.

What else is missing from the analysis model? You might note the absence of methods. Although some authors would disagree, we feel that identification of application-domain objects should come first. The object diagram defines the universe of discourse on which behavior operates. It is important to define what something *is* before describing what it *does.* Once the objects and their structural relationships are identified, you can describe what they do. Operations can then be added to the model.

The analysis model does not attempt to encapsulate information. The analyst should take a "God's eye" view of the problem and capture all the information available. Accessing attributes and traversing associations are legitimate sources of information that do not require any special dispensation. How can you make a good design if you conceal information from yourself? Encapsulation is a design construct intended to limit the effect of changes within an implementation; it is not an analysis construct.

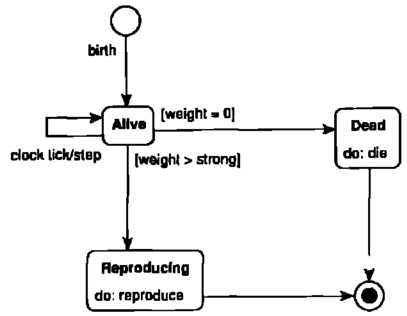


Figure 2.

DYNAMIC MODEL

The *object model* specifies the structure of the objects in the Bugs simulation. During analysis, you must, of course, define the behavior that you want your system to have. Behavior can be specified by the interactions that occur between objects and the transformations that objects undergo. In the OMT methodology, interactions are specified by the *dynamic model* and transformations by the *functional model*.

The dynamic model specifies the external interactions of the system with outside agents. The dynamic model is represented graphically by state diagrams: one for each class with dynamic behavior. Figure 2 shows a *state diagram* for class Bug. The state diagram shows the life history of a bug. Each rounded box is a different state. The behavior of a bug is very simple. It only has one state, Alive, during much of its life. The other states are initialization or termination states. An arrow between states shows a state transition in response to an *event*, which is an interaction between objects. The open circle labeled "birth" points to the initial state of the object, the state Alive.

The only event a bug responds to is clock tick, i.e., the passage of a unit of time. The passage of time may be regarded as an event from the universe to an object. When an event occurs, the object takes a transition from the current state labeled by the event. When a transition occurs, an object may perform an operation and transition to a new state. When clock tick occurs, the bug performs operation step and returns to the Alive state. The bug also responds to two possible conditions shown as transition labels in brackets. A transition occurs whenever one of the conditions becomes true. If the bug starves (weight = 0), then it transitions to state Dead, where it performs operation die and then ceases to exist (shown by the bull's eye). If the bug gets fat enough (weight > strong), then it transitions to state Reproducing where it performs operation reproduce, which creates two new bugs to take its place. The original bug then ceases to exist. (We could have drawn the state diagram so a reproducing bug made a single copy of itself and continued to exist but the way I have drawn the diagram is more symmetric.)

The event clock tick affects every bug. In what order do the various bugs perform their operations? For this simulation, it doesn't much matter so we don't specify it. *Objects are inherently concurrent.* Since all the major object-oriented languages are sequential, during design we must serialize the execution of Bug operations but during analysis a concurrent viewpoint is just fine.

This state diagram completely defines the behavior of the system. All operations are ultimately initiated by clock ticks. But where do we specify the effect of an operation? That is done in the functional model.

FUNCTIONAL MODEL

The functional model specifies the effect of operations on data values. It is expressed by *data flow diagrams*, one per nontrivial operation. Figure 3 shows the step operation on Bug that is performed every clock tick. In the diagram, boxes represent objects, ovals represent functions, and arrows represent the flow of data values.

The diagram for step shows there are three independent computations within the operation: updating of age, weight, and spatial parameters. For example, the arrow leaving Bug labeled "age" represents the age attribute of Bug. The growOlder function takes an age as input and yields a new age as output (most likely a simple increment). The arrow from growOlder to Bug' labeled age' represents updating the age attribute of Bug. The prime symbols are included merely to distinguish original and updated values. They could be omitted but the diagram is easier to read if old and new values are visually distinguished.

Operations growOlder, metabolize, and eat are all simple operations that can be described by formulas. For example, growOlder might be age' = age + 1 and eat might be weight" = weight' + foodValue.

The find operation is a simple data access within the object diagram. Its inputs are the location attribute of a bug and the

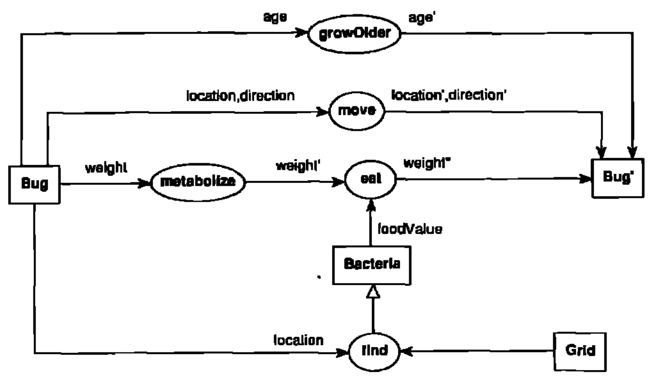


Figure 3.

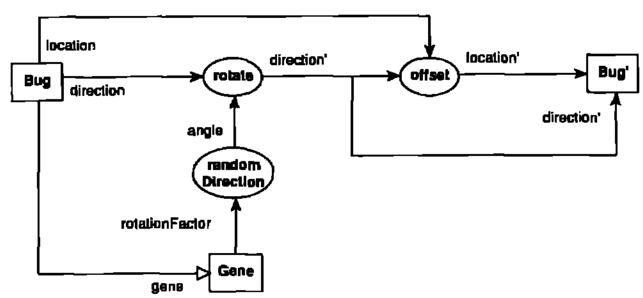


Figure 4.

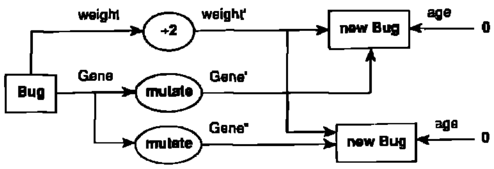


Figure 5.

grid itself. Its output is the bacterium (if any) found at the location within the grid. However, we don't want the bacterium itself but its food value as input to the eat function. The solid arrowhead on the output of the find operation indicates a shift in viewpoint about the data value, looking at it as an object rather than just a value. We can then pull the attribute foodValue out of the Bacterium object.

Operation move from Figure 3 has been expanded into an entire data flow diagram in Figure 4. This operation updates two attributes simultaneously.

Figure 5 shows the reproduce operation on Bug. In this diagram, two new bugs are created from scratch and their attributes initialized from the attributes of the original bug. The age of the new bugs is set to the value 0, however.

THREE MODELS

The analysis is now complete and described by three separate but related models. The object model describes the information structures of the system. The dynamic model describes the external stimuli that initiate activity on objects and the operations that are invoked. The functional model describes the computations on values performed by each operation. Together, all three models describe what a system does with minimal constraints on how it must be implemented.

As a final step of analysis, you may summarize operations from the dynamic and functional models onto the object model. Figure 6 shows the Bugs object diagram with operations allocated to object classes. Operations that update attributes have been allocated to the class owning the attributes. For example, growOlder and metabolize have been assigned to Bug.

We can use the analysis model to answer all kinds of questions about the system we are building. We can ask and answer queries about the state of the system, the response of the system to stimuli, and how values are computed. We can execute the simulation to a certain level of detail. We cannot completely execute the model because we left some details open such as the mapping of the gene rotation factors into probability vectors. We omitted these details because we did not care exactly how they are implemented.

This example is brief and I do not have the space to explain it in full detail. There are details in the diagrams that you can puzzle out on your own. In future columns, I will follow the problem through the design and implementation stages.

During design, we must resolve any open issues and expand the details of any loosely specified operations. We must also transform and optimize the analysis model so that it is efficient enough for implementation. During implementation, we must map the design into a specific programming language and satisfy all of the rules and conventions of the chosen language.

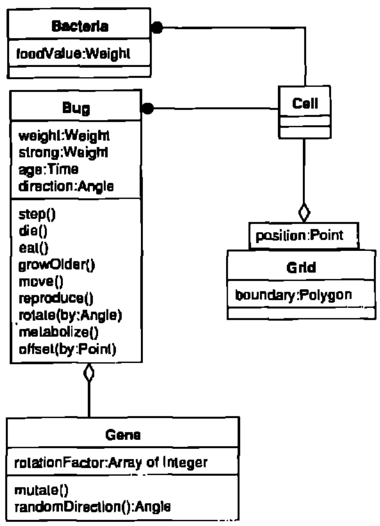


Figure 6.

OF MODELS AND COLUMNS

In future columns, I hope to look at different aspects of modeling and design sometimes taking a high-level view of a broad area and sometimes exploring some interesting narrow issue in detail. I do not intend to recapitulate the material in our book in detail but I will touch on some of it in passing and also bring up some new issues. We are still learning from others and we hope they will learn from us so you may see changes and inconsistencies over time. That's life, real and artificial. Methodologies as well as bugs and designs must evolve, so I would welcome feedback from readers. ■

ACKNOWLEDGMENT

This month's column includes material from the Object-Oriented Modeling and Design Tutorial *by James Rumbaugh et al. Used by permission of the authors.*

REFERENCES

[Dewtn89] Dewtney, A.K. Mathematical recreations, *Scientific American*, 260, 5, 1989.

[Rumba91] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

James Rumbaugh is a computer scientist at General Electric Research and Development Center in Schenectady, NY. Dr. Rumbaugh has been active in object-oriented technology for many years. He developed the object-oriented language DSM, the OMT methodology, and the OMTool graphic editor. He is author (with Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen) of Object-Oriented Modeling and Design *by Prentice Hall. He can be reached at GE R&D Center, Bldg K1-5B42A, PO Box 8, Schenectady, NY 12301, by phone at (518)387-6358, or by email at rumbaugh@crd.ge.com.*

Making inferences about objects

by Paul Harmon

IN MY LAST COLUMN, I discussed some of advantages that could be gained by combining the features of frames, a concept derived from the AI world, with the class/instance approach found in the world of object-oriented programming. The mixture of frames and objects, as exemplified by the best of the current expert system-building tools, offers greater power and flexibility. In making that argument, I relied on the features that frames bring to objects including defaults and constraints on the values of attributes, class-specific attributes, and the ability to control inheritance in various ways. In this column, I want to consider the additional power that can be gained when you combine object-oriented systems with inference/rule-based systems.

I propose to describe a scheduling problem that I call the Trucks & Drivers problem. It provides a modest but interesting example of how one can combine an inference-based set of rules with an object-oriented system to solve a problem much more efficiently than either technology could by itself.

The Trucks & Drivers problem is simple: we want to develop a truck scheduling system that will identify pairs of trucks and drivers that are available at the same location and ready to be dispatched. Rather than just pairing any truck with any available driver, we will also need to apply some criteria to assure that we use the "best" available driver at any point in time.

We will need to create three classes, one to describe drivers, one to describe trucks, and one to describe successful matches between the two. Our classes take the form shown in Table 1.

We will describe the uses of the slots and methods in a moment. Note first, however, that this application assumes an expert system tool that can automatically link with various relational databases. In this case, we include a dBASE method in both the Truck and Driver class. (This is a prespecified method available in the tool.) This method will automatically generate and execute the code necessary to obtain information from records in database files on trucks and drivers. In other words, the Truck and Driver classes will be instantiated by drawing on values stored in records in Truck and Driver database files.

In addition to the three classes, we will write a single rule that will be manipulated by an inference engine. (An "inference engine" is simply an algorithm for searching for rules and evaluating them. The use of an inference engine assures that the application will use dynamic binding just as an object-oriented application that incorporates virtual methods makes certain decisions at runtime.) We could, of course, write an inference engine from scratch but that wouldn't be very efficient. It makes a lot more sense to acquire an expert system-building tool, develop our objects and rules within that tool, and then embed that tools' inference engine in the final application when it is compiled.

To solve our scheduling problem, we will need the following rule:

```
If      orderby (Driver?.Score)
        and Driver?.Return_Status = available
        and Truck?.Return_Status = available
        and Truck? and Driver? with
            Truck?.In_City = Driver?.In_City

Then    send (Make_Unavailable to Driver?)
        send (Make_Unavailable to Truck?)
        send (Create_Result, Truck_License
            and Driver_Name to class (Results))
```

This rule is a pattern-matching rule because it does not refer to any specific instance of either Truck or Driver. Instead, the inference engine automatically seeks out instances of trucks and drivers and successively binds them with this rule to determine if there are one or more successful implementations of this rule.

To make this rule even more powerful, we have included an orderby command in the rule. The orderby command evokes the A* algorithm, an AI search technique that will prioritize any list of drivers according to some set of criteria. In this specific case, the orderby command sends a message to a method, Driver.Score. That method, in turn, applies a formula to the values of the Seniority, Safety_Record, and Layover slots associated with each instance of the Driver class and creates an index that orders the drivers according to a score assigned to each instance. This index is held in memory so it can be reused. The instance of Driver with the highest score is returned to the rule. Each time the rule is reinstantiated the instance of Driver with the next highest score is returned. This continues until the entire list of Driver instances is exhausted.

Driver? indicates that the rule will examine instances of the Driver class. As each instance is identified, it will be bound with Driver? (e.g. Driver1, Driver2, etc.) and substituted into the rule wherever Driver? oc-

Table 1.

Class: Driver		
	slots:	Name (any name)
		Status (available/unavailable)
		City (SF, LA, NY, InRoute)
		Seniority (no. of years with firm)
		Safety_Record (no. of accidents)
		Layover (number of days that the driver has had off since last trip)
	methods:	Return_status (returns value for status slot)
		In_City (returns value for city slot)
		Driver_Name (returns value of Driver_Name)
		Make_Unavailable (changes value of status slot to unavailable)
		Score (returns a value derived by applying a formula to the values associated with the seniority slot, the safety record slot and the layover slot)
		dBASE (automatically generates code to obtain record information from database)
Class: Truck		
	slots:	License (License number)
		Status (available/unavailable)
		City (SF, LA, NY, InRoute)
	methods:	Return_Status (returns value for status slot)
		In_City (returns value for city slot)
		Make_Unavailable (changes value of status slot to unavailable)
		License_Num (returns value of License)
		dBASE (automatically generates code to obtain record information from database)
Class: Results		
	slots:	Truck (license)
		Driver (name)
	methods:	Create_Result (creates an instance of results class)
		Truck_License (places value of truck license in the new instance)
		Driver_Name (places value of driver in the new instance

curs. Next, the inference engine will identify an instance of the Truck class and bind it with the term Truck?. By binding and unbinding instances of Truck and Driver, the rule will be used over and over again.

The clause: Driver?.Return_Status = available sends a message to the bound instance of Driver to fire a method called Return_Status. This method, in turn, checks the slot of the Driver instance called Status and returns its value. If the value of the Driver1.Status slot is available, this clause succeeds and the inference engine moves on to the next clause of the rule.

In a similar manner, the rule initiates a message to the Truck instance (e.g., Truck1) that has been bound to determine if the truck is available. Assuming the value of the Truck1.Status slot" = available, the inference engine proceeds to check the next clause. The fourth clause sends messages to both the Truck and the Driver instances to determine what city each instance is in. If they are in the same city, the rule proceeds.

Whenever a match is found, the inference engine proceeds to the Then portion of the rule and sets the value of each of the instances' Status slots to unavailable. Next, it creates an instance of the Results class and assigns the driver's name and the truck's license to the new instance. (The entire application is controlled by an Agenda that began by initiating the forward chaining rule. When the rule has fired as many times as it can, the second item on the Agenda, which calls for a printed list of all instances of the Results object, is triggered and the application is complete.)

Figure 1 illustrates the status of our Truck & Driver application at the point when the system has successfully fired the rule once and identified one match. The inference engine has now reinstantiated the rule with new instances of Truck and Driver and is now ready to try for a second match. (Note that the second rule will fail since Truck2 is in a different city than Driver2.)

If you think of an instance as similar to a relational database record, and you consider the instances of different classes (files) as records belonging to different files, then our pattern-matching rule is doing what a database programmer would call "joins." In most cases, however, pattern-matching rules are much more efficient than database joins since the inference engine dynamically sets successful matches to "unavailable" thereby successively reducing the set of available trucks and drivers that must be checked during each successive round of search. In addition, the use of the A* algorithm assures that the search will be prioritized. In other words, the use of inferencing, pattern-matching rules, and classes that can be instantiated from a database provides developers with a much more efficient way to handle complex configuration, planning and scheduling problems that either rules or objects, by themselves, could provide. (It is exactly these types of problems that have led all major expert system tool vendors to add object-oriented capabilities to their tools.)

In addition, since an inference engine examines whatever rules it finds in the knowledge base when the application is run we could easily modify our program by adding additional rules to the knowledge base. We could add rules to handle exceptions. Similarly, in some emergency, we could add or modify rules to handle special situations. All the arguments that can be made for the advantages of the mod-

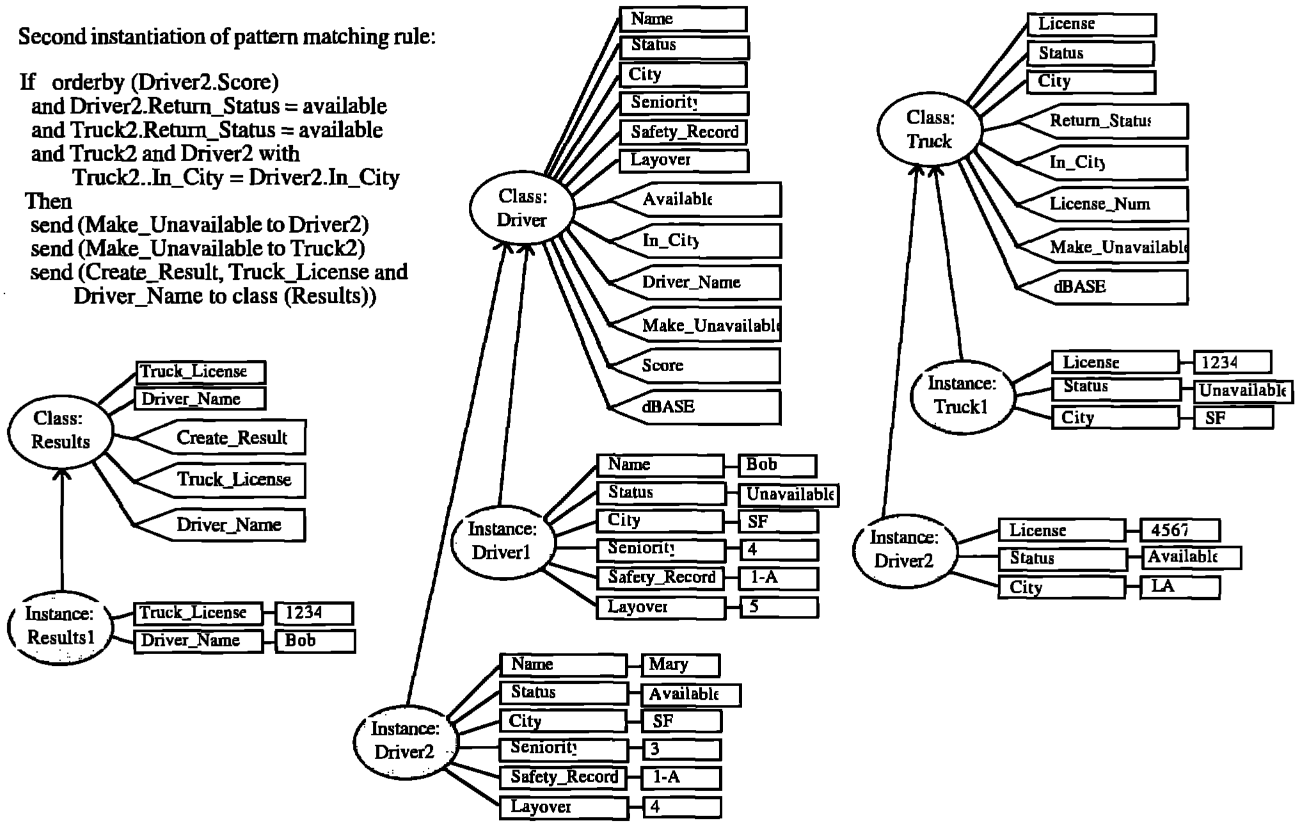


Figure 1. The Trucks & Drivers situation after one rule has fired.

ularity inherit in object-oriented programming can also be made for the use of inferencing and rules. The two techniques combined, each representing a slightly different type of modularity, are much more powerful than either by itself.

There are certainly simpler problems for which languages like Smalltalk and C++ are well suited. When you consider integrating object-oriented programming capabilities into a CASE tool to facilitate the development of large, complex commercial applications, however, it's hard to imagine that users aren't going to want the advanced object-oriented features that result from combining frames and objects. They will also want the additional capabilities that can only be obtained by combining objects with inferencing and rules.

Several popular expert system-building tools could come close to the solution I have reviewed. As far as I know, however, only Aion Corporation's ADS tool has the capability of combining the inferencing and pattern-matching capabilities illustrated with message passing and the full encapsulation that is illustrated. (Most expert system tool vendors are still in the process of adding message passing and figuring out how to enforce encapsulation while still running efficiently in mainframe environments.) When you consider that Aion's ADS is written in C, runs on mainframes in environments like IMS and CICS, and accesses all the mainframe databases, you realize why I believe that the most powerful and practical object-oriented programming environments are being sold by expert systems vendors. ■

PRODUCT INFORMATION

AION DEVELOPMENT SYSTEM (ADS)
AION CORP.
101 UNIVERSITY AVE.
PALO ALTO, CA 94301
(415)328-9595, fax (415)321-7728.

ACKNOWLEDGMENTS

The author wishes to acknowledge the help received from Jan Aikins and Bernadette Kowalski of Aion Corporation in setting up and testing this problem. The syntax of the rule and the classes listed in this article, however, are not from Aion's ADS. Aion's syntax is more elegant, but would require more information about how an inference engine works. I modified the syntax to make it easier to describe the Trucks & Drivers *application in such a short space.*

Paul Harmon is the editor of two newsletters: Object-Oriented Strategies *and* Intelligent Software Strategies. *He is the coauthor of three popular books on expert systems and the CEO of ObjectCraft Inc. He can be reached at Harmon Associates, 151 Collingwood, San Francisco, CA 94114, by phone at (415)861-1660, or by fax at (415)861-5398.*

Combining modal and nonmodal components to build a picture viewer


by Wilf LaLonde & John Pugh

OVER THE PAST several months, we watched a colleague develop an application interface that had a requirement for large numbers of iconic buttons and static pictures. A great deal of his time was spent importing color pictures from a Microsoft Windows paint program through the clipboard, finding that minor variations were needed, moving them back to the paint program, and repeating the cycle.

There were several annoyances in this cycle. Since there were many dictionaries of such pictures, the picture to be updated had to be located, often by inspecting successive candidates and displaying them by sending each an explicit display message to get a visual check. Next, care had to be taken to place a copy of the picture on the clipboard because the operation that actually moves the bits into the clipboard ultimately destroys (releases) the picture when a new picture is placed in the clipboard. Of course, if you could be guaranteed that the transfer was actually going to be successful you could avoid making a copy. When the clipboard picture was successfully pasted into the paint program, it was necessary to come back to Smalltalk to explicitly release the original picture because Smalltalk/V Windows keeps handles into operating system memory where the bits are actually kept. Coming back the other way is much simpler because a new picture is created in the process.

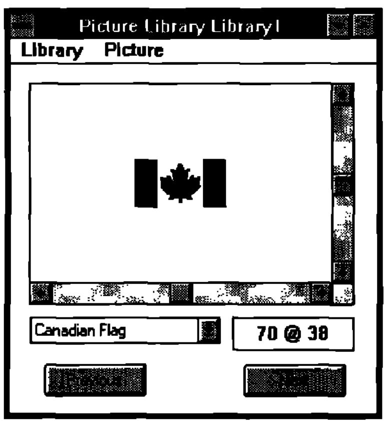


Figure 1. The picture viewer.

What makes the process painful is that you have to continually execute bits and pieces of code that are kept, say, in a special workspace. Every now and then, this code is discarded, sometimes deliberately and sometimes accidentally, and must be regenerated.

What was needed was a simple picture browser (Fig. 1) that supported these operations transparently. The browser we describe is based on an original design by Wayne Beaton but has undergone substantial modifications. In particular, the new design subscribes to the usual editing paradigm whereby a user is always editing a copy rather than the original. It also makes use of modal dialog boxes for opening and saving information. The modal dialog boxes and the browser, which we call the picture viewer, were all developed with Acumen's Window Builder for Smalltalk/V Windows. It may be a surprise to some of you that dialog box functionality is already supported by the builder; i.e., there is no need for an external dialog box editor.

DESIGNING THE PICTURE VIEWER

The picture viewer is designed to keep track of a number of different picture libraries that it maintains in a class variable called **PictureLibraries** — a dictionary in which the key is the name of the library and the value is another dictionary of pictures keyed by the picture name. We can also file out the libraries but we won't focus on that issue here.

In a typical session with the viewer, a user might open an existing library using **Open...** in the **Library** menu (Fig. 2). Next, he might look at the pictures it contains by clicking on the **Next** (or **Previous**) buttons. The name of the picture is displayed in the combo box while its extent is displayed to the right. It is also possible to go directly to a specific picture by selecting the appropriate name in the combo box.

To copy a picture into the clipboard or paste the clipboard over an existing picture, the **Copy** or **Paste** operation, respectively, in the **Picture** menu can be used (Fig. 3). Menu command **New...** requires a prompt for the name of the picture; it produces an empty picture that can subsequently be pasted over.

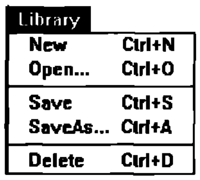


Figure 2. The **Library** menu.

Listing 1. Class **ListQueryDialog.**

```
class               ListQueryDialog
superclass          WBTopPane
instance variables  result listPane list

class methods

examples

example1
    "ListQueryDialog example1"
    ^ListQueryDialog new
        label: 'Choose a color';
        openOn: #('red' 'green' 'blue')

instance methods

generated by builder

addSubpanesTo: aPane
^   aPane
        owner: self;
        when: #opened perform: #opened:;

        addSubpane: (
            Button new
                owner: aPane;
                setStyle: #defaultPushButton;
                when: #clicked perform: #ok:;
                contents: 'OK';
                framingBlock: (23 @ 152
                    rightBottom: 128 @ 180);
                yourself);

        addSubpane: (
            Button new
                owner: aPane;
                when: #clicked perform: #cancel:;
                contents: 'Cancel';
                framingBlock: (139 @ 152
                    rightBottom: 244 @ 180);
                yourself);

        addSubpane: (
            listPane := ListBox new
                owner: aPane;
                nameForInstVar: 'listPane';
                when: #doubleClickSelect
                    perform: #selectListEntry:;
                framingBlock: (22 @ 23
                    rightBottom: 244 @ 137);
                yourself)

buildMenuBarFor: aPane
    "Nothing"

defaultFrameStyle
    Smalltalk isRunTime
        ifFalse: [
            ^(WinConstants at: 'WsOverlapped') |
            (WinConstants at: 'WsClipchildren') |
            (WinConstants at: 'WsCaption')]
        ifTrue: [^46137344]

initWindowExtent
    ^270 @ 218

builder override

isModal
    ^true
label
    ^label

opening and closing

openOn: aCollection
    list := aCollection.
    ^self open

result
    ^result

top pane event handling

opened: aPane
    listPane
        contents: list;
        selectIndex: (list isEmpty
            ifTrue: [0]
            ifFalse: [1]);
        setFocus

list pane event handling

selectListEntry: aPane
    "Assumes the list entry is already selected."
    self ok: nil.

button pane event handling

cancel: ignore
    result := nil.
    self closeWindow.

ok: ignore
    result := listPane selectedItem.
    self closeWindow.
```

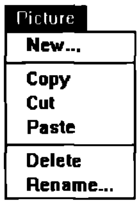


Figure 3. The **Picture** menu.

DESIGNING AND IMPLEMENTING APPROPRIATE MODAL DIALOG BOXES

The original implementation of the picture viewer used simple prompters for reacting to **Open...** and **SaveAs...** in the **Library** menu. To improve on this, we used the Window Builder to design two dialog boxes as shown in Figures 4 and 5.

Initially, these dialog boxes were specifically designed for the picture viewer but it quickly became apparent that little work had to be done to make them more generally useful. We called them **ListQueryDialog** (for picking and choosing an arbitrary element of a list) and **ListExtensionDialog** (for picking and choosing as before but also permitting the new element to be supplied by typing it). See the corresponding example class methods in Listings 1 and 2 for how they might be used.

The dialog box was designed to respond to four events:

- the top pane's **#opened** event, which has to place the list of items in the list pane.
- the list pane's **#doubleClickSelect** event that doubles for a click on the **OK** button.
- the **OK** and **Cancel** buttons' **#clicked** event that, respectively, set the value of

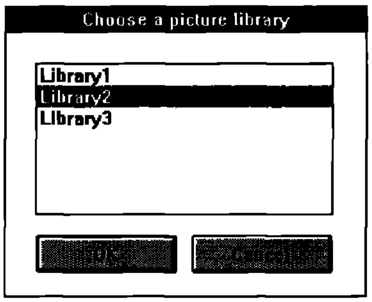


Figure 4. The **Open...** dialog box.

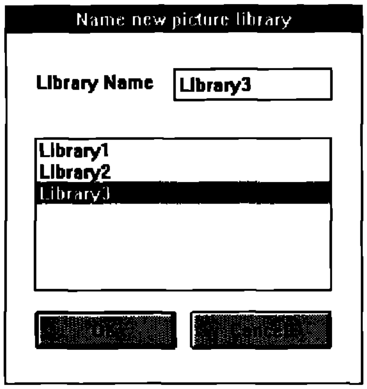


Figure 5. The **SaveAs...** dialog box..

the **result** instance variable to the item selected in the list pane or nil.

We had to browse the Smalltalk library to find out that modal dialog boxes send the message **result** to obtain the value to be returned (an Acumen extension).

When designing the dialog box in the Window Builder, no option or switch was located that enabled us to specify whether or not the resulting window was to be modal. A modal window prevents users from carrying on in an application until a response is provided. Making a window modal is simply a matter of clicking a switch in the builder.

There was, however, one problem that was caused by the builder. We needed to be able to supply an arbitrary title. Normally, this is done by sending the message **label:** aString to the window. This causes the window to redisplay the string it obtains by sending itself the message **label**. However, the builder insists on changing the code for this method to ^label:=userSuppliedStringConstant, which causes any label changes to be ignored. What the builder should have done is add the required **label:** aString message in the generated pane construction method **addSubpanesTo:**. We simply replaced the problem method with the correct version that exists in a superclass.

The dialog box for class **ListExtensionDialog** was obtained by editing the **ListQueryDialog** window to add two more panes: a static text pane (referenced by instance variable subtitlePane) and an entry field (referenced by instance variable namePane). The static text pane's contents could be supplied by the user by sending the window the message **subtitle:** aString. The entry field permits an element not in the list to be supplied.

An additional handler, method **clickListEntry:**, for event **#select** in the list pane was added to ensure that the selected list element was inserted into the entry field. Selecting an element didn't require a handler in the previous dialog box because the selected element was retrieved only when the **OK** button was pressed. Of course, even though there was no handler, the list element was still selected as a user clicked on it in the list pane.

The only other complication involves the **subtitle:** aString message. Normally, a

Listing 2. Class **ListExtensionDialog.**

```
class              ListExtensionDialog
superclass         ListQueryDialog
instance variables namePane subtitlePane subtitle

class methods

examples

example1
    "ListExtensionDialog example1"
    ^ListExtensionDialog new
        label: 'Choose a color';
        subtitle: 'Color name';
        openOn: #('red' 'green' 'blue')

instance methods

generated by builder

addSubpanesTo: aPane
    ... similar to Listing 1 except for ...

    addSubpane: (
        subtitlePane := StaticText new
            owner: aPane;
            nameForInstVar: 'subtitlePane';
            contents: 'untitled';
            framingBlock: (22 @ 29
                rightBottom: 116 @ 57);
            yourself);

    addSubpane: (
        listPane := ListBox new
            owner: aPane;
            nameForInstVar: 'listPane';
            when: #select
                perform: #clickListEntry:;
            when: #doubleClickSelect
                perform: #selectListEntry:;
            framingBlock: (21 @ 79
                rightBottom: 243 @ 193);
            yourself);

    addSubpane: (
        namePane := EntryField new
            owner: aPane;
            nameForInstVar: 'namePane';
            framingBlock: (125 @ 27
                rightBottom: 243 @ 51);
            yourself)

initWindowExtent
    ^267 @ 282

dialog box initialization

subtitle: aString
    subtitle := aString.
    subtitlePane isNil
        ifFalse: [subtitlePane
            contents: aString]

top pane event handling

opened: aPane
    namePane contents: (list isEmpty
        ifTrue: [''] ifFalse: [list first]).
    self subtitle: subtitle.
    super opened: aPane.

list pane event handling

clickListEntry: aPane
    "Assumes the list entry is already selected."
    namePane contents: listPane
        selectedItem

selectListEntry: aPane
    "Assumes the list entry is already selected."
    namePane contents: listPane
        selectedItem.
    self ok: nil.

button pane event handling

ok: ignore
    result := namePane contents.
    self closeWindow.
```

user would supply the subtitle (see "Library Name" in Fig. 5) before the window is opened. At that time, the subtitle pane doesn't exist so the subtitle must be stored in a local variable (subtitle). When the window is opened, the **#opened** event handler can place the string in the subtitle pane. Of course, users might want to dynamically change this subtitle. The short (but nevertheless complex) implementation of method **subtitle:** handles these possible scenarios.

SMALLTALK/V WINDOWS EXTENSIONS TO SUPPORT THE PICTURE VIEWER

To support the manipulation of the pictures conveniently, it was necessary to add obviously missing methods to class Bitmap, e.g., deep and shallow copy operations as shown in Listing 3.

More fundamental and problematic was that fact that halfway through our implementation we discovered that copy operations for dictionaries were incorrectly implemented. We were taking deep copies of libraries (dictionaries of bitmaps) and finding that releasing the bitmaps in the copy destroyed the originals, too. Our initial reaction was to implement our own private method that performed the copy correctly but we ultimately decided that a proper solution required a change to the system.

The problem stems from the fact that the original implementers provided an implementer's view of the solution rather than a user's view. Intuitively, a shallow copy of an array provides a user with a new array sharing the elements of the old. Moreover, changes to the new array don't affect the original. Similarly, a shallow copy of a dictionary should provide a user with a new dictionary sharing the keys and values of the old. Changes to the new dictionary should not affect the old (which was not the case). A deep copy is similar except that a shallow copy of the elements is made in the case of an array (a shallow copy of the keys and values in the case of a dictionary). Consequently, users expect to be able to change the elements (keys and values) in the deep copy without affecting the corresponding elements (keys and values) of the original. As implemented, neither the shallow or deep copy operation for dictionaries makes copies of the keys and value. The revised methods are shown in Listing 4.

Listing 3. Extensions to class Bitmap.

```
class    Bitmap

instance methods

copying

deepCopy
    | bitmap |
    bitmap := self class
        screenExtent: self extent.
    bitmap pen
        copyBitmap: self
        from: self boundingBox
        at: 0@0.
    ^bitmap

shallowCopy
    ^self deepCopy
```

IMPLEMENTING THE PICTURE VIEWER

The picture viewer maintains two instance variables, **libraryName** and **pictureName**

Listing 4. Extensions to class Dictionary.

```
class    Dictionary

instance methods

copying

shallowCopy
        "Answer a copy of the receiver which
         shares the receiver keys and values
         (but not the same association
         objects)."
    | answer |
    answer := self species new.
    self associationsDo: [:element |
        answer add: element shallowCopy].
    ^answer

deepCopy
        "Answer a copy of the receiver with
         shallow copies of the keys and values
         (which requires a deep copy of the
         association objects)."
    | answer |
    answer := self species new.
    self associationsDo: [:element |
        answer add: element deepCopy].
    ^answer
```

Listing 5. Class PictureViewer.

```
class               PictureViewer
superclass          WBTopPane
instance variables  library libraryChanged
                    libraryName pictureName
                    picturePane pictureNames
                    Pane pictureSizePane
class variables     PictureLibraries

class methods

examples

example1
    "PictureViewer example1"
    PictureViewer new open

class initialization

initialize
    "PictureViewer initialize"
    (PictureLibraries isKindOf: Dictionary)
        ifTrue: [self release].
    PictureLibraries := Dictionary new

release
    "PictureViewer release"
    PictureLibraries do: [:library |
        library do: [:picture | picture release]]

library access and modification

libraries
    "PictureViewer libraries"
    ^PictureLibraries

libraries: aDictionary
    self initialize.
    PictureLibraries := aDictionary

instance methods

generated by builder

addSubpanesTo: aPane
^   aPane
        owner: self;
        when: #opened perform: #opened:;
        when: #close perform: #closed:;

        addSubpane: (
            StaticBox new
                owner: aPane;
                setStyle: #blackFrame;
                framingBlock: (166 @ 189
                    rightBottom: 257 @ 214);
                yourself);

        addSubpane: (
            Button new
                owner: aPane;
                when: #clicked
                    perform: #clickedNext:;
                contents: 'Next';
                framingBlock: (175 @ 224
                    rightBottom: 251 @ 248);
                yourself);

        addSubpane: (
            picturePane := GraphPane new
                owner: aPane;
                nameForInstVar:'picturePane';
                framingBlock: (14 @ 14
                    rightBottom: 257 @ 180);
                yourself);

        addSubpane: (
            Button new
                owner: aPane;
                when: #clicked
                    perform: #clickedPrevious:;
                contents: 'Previous';
                framingBlock: (26 @ 224
                    rightBottom: 102 @ 248);
                yourself);

        addSubpane: (
            pictureNamesPane := ComboBox new
                owner: aPane;
                nameForInstVar:
                    'pictureNamesPane';
                setStyle: #dropDownList;
                when: #select
                    perform: #selectPictureName:;
                when: #doubleClickSelect
                    perform: #selectPictureName:;
                framingBlock: (15 @ 189
                    rightBottom: 157 @ 293);
                yourself);

        addSubpane: (
            pictureSizePane := StaticText new
                owner: aPane;
                nameForInstVar: 'pictureSizePane';
                setStyle: #centered;
                contents: '32@32';
                framingBlock: (168 @ 193
                    rightBottom: 255 @ 212);
                yourself)

buildMenuBarFor: aPane
    ... code not shown ...

defaultFrameStyle
    Smalltalk isRunTime
        ifFalse: [
            ^(WinConstants at: 'WsOverlapped') |
            (WinConstants at: 'WsClipchildren') |
            (WinConstants at: 'WsCaption') |
            (WinConstants at: 'WsSysmenu') |
            (WinConstants at: 'WsMaximizebox') |
            (WinConstants at: 'WsMinimizebox') |
            (WinConstants at: 'WsThickframe')]
        ifTrue: [^47120384]

initWindowExtent
    ^282 @ 307

isModal
    ^false

builder override

label
    ^label

library menu commands

libraryNew
    self promptForSaveIfChanged.
    self privateClearLibrary: library.
    libraryName := pictureName := nil.
    libraryChanged := false.
    self update

libraryOpen
    | name keys |
    self promptForSaveIfChanged.
    name := ListQueryDialog new
        label: 'Choose a picture library';
        openOn: PictureLibraries keys
            asSortedCollection.
    name isNil ifTrue: [^self]. "User cancelled."
    self privateClearLibrary: library.
    library := (PictureLibraries at: name)
        deepCopy.
    libraryName := name.
    keys := library keys asSortedCollection.
    pictureName := keys isEmpty
        ifTrue: [nil]
        ifFalse: [keys first].
    libraryChanged := false.
    self update

librarySave
    libraryName isNil
        ifTrue: [^self librarySaveAs].
    self privateClearLibrary: (PictureLibraries
        at: libraryName).
    PictureLibraries at: libraryName
        put: library deepCopy.
    libraryChanged := false
```

Listing 5. Class PictureViewer *(continued)*.

```
librarySaveAs
    | name |
    name := ListExtensionDialog new
        label: 'Name new picture library';
        subtitle: 'Library Name';
        openOn: PictureLibraries keys
            asSortedCollection.
    name isNil ifTrue: [^self]. "User cancelled."
    self privateClearLibrary: (PictureLibraries
        at: name ifAbsent: [Dictionary new]).
    PictureLibraries at: name
        put: library deepCopy.
    libraryName := name.
    libraryChanged := false.
    self updateLabel

libraryDelete
    libraryName isNil ifTrue: [^self].
    (MessageBox confirm: 'Delete library ',
    libraryName)
        ifFalse: [^self].
    self privateClearLibrary: (PictureLibraries
        at: libraryName).
    PictureLibraries removeKey: libraryName.
    self privateClearLibrary: library.
    libraryName := pictureName := nil.
    self update

picture menu commands

pictureNew
    | name picture |
    name := self promptForName: 'picture'
        in: library.
    name isNil
        ifTrue: [^nil]. "User changed his mind."
    picture := Bitmap screenExtent: 0@0.
    library at: name put: picture.
    pictureName := name.
    libraryChanged := true.
    self updatePictureNames; updatePicture

pictureCopy
    | picture |
    (picture := self picture) isNil ifTrue: [^self].
    Clipboard setBitmap: picture copy

pictureCut
    ^self pictureCopy; pictureDelete

picturePaste
    pictureName isNil ifTrue: [^self].
    (library at: pictureName) release.
    library at: pictureName put:
        Clipboard getBitmap.
    libraryChanged := true.
    self updatePicture

pictureDelete
    | names nameIndex |
    pictureName isNil ifTrue: [^self].
    (library at: pictureName) release.
    library removeKey: pictureName.
    names := pictureNamesPane contents.
    nameIndex := pictureNamesPane
        selectedIndex.
    pictureName := nameIndex > 1
        ifTrue: [names at: nameIndex - 1]
        ifFalse: [names size > 1
            ifTrue: [names at: 2]
            ifFalse: [nil]].
    libraryChanged := true.
    self updatePictureNames; updatePicture

pictureRename
    | name picture |
    pictureName isNil ifTrue: [^self].
    name := self promptForName: 'new picture'
        in: library.
    picture := library at: pictureName.
    library removeKey: pictureName.
    library at: name put: picture.
    pictureName := name.
    libraryChanged := true.
    self updatePictureNames; updatePicture

top pane event handling

opened: aPane
    library := Dictionary new.
    libraryName := pictureName := nil.
    libraryChanged := false.
    self update

closed: aPane
    self promptForSaveIfChanged.
    self privateClearLibrary: library.
    ^super close

list pane event handling

selectPictureName: aPane
    pictureName := aPane selectedItem.
    self updatePicture

button pane event handling

clickedNext: aPane
    self privateMovePictureByOffset: 1

clickedPrevious: aPane
    self privateMovePictureByOffset: -1

support operations

picture
    ^library at: pictureName ifAbsent: [nil]

promptForName: title in: aDictionary
    | name |
    name := Prompter
        prompt: 'Enter ', title,
            ' name or nothing to cancel'
        default: ''.
    (name isNil or: [name isEmpty])
        ifTrue: [^nil]. "User changed his mind."
    (aDictionary keys includes: name)
        ifTrue: [
            (MessageBox confirm:
                'Name already exists. Try again.')
                ifFalse: [^nil].
            ^self promptForName: title
                in: aDictionary]
        ifFalse: [^name]

promptForSaveIfChanged
    | name |
    libraryChanged ifFalse: [^self].
    name := libraryName isNil
        ifTrue: ['']
        ifFalse: [' ', libraryName].
    (MessageBox confirm: 'Changes made. ',
    'Save Library', name, '?')
        ifFalse: [^self].
    self librarySave

updating

update
    self
        updateLabel;
        updatePictureNames;
        updatePicture

updateLabel
    self label: 'Picture Library ',
        (libraryName isNil
            ifTrue: ['Untitled']
            ifFalse: [libraryName])

updatePictureNames
    pictureNamesPane contents: library keys
        asSortedCollection.
    pictureNamesPane selectItem: pictureName
```

Listing 5. Class PictureViewer (*continued*).

```
updatePicture
    | picture offset |
    picturePane pen
        deleteAllSegments; erase.
    pictureSizePane contents: ''.
    pictureName isNil ifTrue: [^self].
    picture := self picture.
    offset := (picturePane extent - picture
extent)
        // 2.
    picturePane pen
        drawRetainPicture: [
            picturePane pen
                copyBitmap: picture
                from: picture boundingBox
                at: offset].
    pictureSizePane contents: picture extent
        printString.

private

privateClearLibrary: aDictionary
    aDictionary
        do: [:picture | picture release];
        initialize: aDictionary size + 10.

privateMovePictureByOffset: anInteger
    | names nameIndex newIndex |
    names := pictureNamesPane contents.
    nameIndex := pictureNamesPane
        selectedIndex.
    nameIndex isNil
        ifTrue: [nameIndex := anInteger
                                positive
            ifTrue: [0]
            ifFalse: [names size + 1]].
    newIndex := nameIndex + anInteger.
    (newIndex between: 1 and: names size)
        ifFalse: [^self].
    pictureName := names at: newIndex.
    self updatePictureNames;
                        updatePicture
```

(see Listing 5), for keeping track of the current library and picture names (either can be nil) and an instance variable **library** for maintaining a copy of the library being edited — a library is a picture dictionary. By working on a copy, arbitrary changes can be made without fear of undoable changes. The **update** operation can redisplay the complete user interface from these three variables. Instance variable **libraryChanged** ensures that we don't prompt the user for a save if no changes have been made. The remaining instance variables provide access to the important panes.

The picture viewer is similar to the modal dialog boxes in terms of the complexity of the panes and their interractions. What differentiates it from the dialog boxes is the extensive **Library** and **Picture** operations. To provide a flavor for the implementation, let's consider one sample from each group, say method **libraryOpen** and **picturePaste**.

Method **libraryOpen** begins by prompting the user to save the current library if changes were made. If the library name is nil, a **librarySaveAs** message is sent (which prompts for a new name); otherwise, a **librarySave** message is sent. Next, a dialog box is created to obtain the new library name from the user. If the user doesn't cancel (the name is nil if he does), the existing library (the working copy) must be discarded by explicitly releasing each bitmap. The working library must be replaced by a copy of the library specified by the user. If there are pictures in the library, the name of the first picture in the sorted list is recorded; otherwise, nil is recorded. Once instance variables **libraryName** and **pictureName** are set, the **update** method can display all the required information in the user interface.

Method **picturePaste** implements the code that permits a user to paste over an existing picture. If a new picture is needed, the user should have performed a **New...** operation prior to the paste. The implementation begins by making sure that there exists a selected picture for modification. Next, the old picture must be explicitly released before a new one can be obtained from the clipboard. The fact that the working library has been changed is recorded and the subset of the user interface affected is updated (in this case, just the picture portion).

In general, the most worrisome problems with this specific application have to do with making sure bitmaps are released when they are no longer needed and making sure that proper working copies of libraries are obtained; i.e., copies that properly duplicate the bitmaps. Placing bitmaps in the clipboard also requires a copy because clipboard operation **setBitmap:** ultimately releases the bitmap when a new bitmap is added via a subsequent **setBitmap:** message.

CONCLUSIONS

Smalltalk programmers (ourselves included) have a tendency to be forever building new tools. With the aid of a window builder, such diversions can be easily justified since they don't take very much time and often end up saving time in the long run.

Also, it should be clear that there is little difference between designing a dialog box and designing a nonmodal window since the same tool can be used for designing both.

Tools that eliminate the problems inherent with the need for releasing bitmaps are a step in making it easier to avoid mistakes. ■

ACKNOWLEDGMENT

This article owes a great deal to Wayne Beaton, who produced the first prototype. His interractive demonstration convinced us of the utility and simplicity of a picture viewer.

Wilf R. LaLonde and John Pugh are Professors of Computer Science at Carleton University in Ottawa, Canada. Their research interests include object-oriented systems, connectionist systems, visual programming and user interfaces. They are co-authors of Inside Smalltalk: Volumes 1 and 2; *two books that survey the entire Smalltalk system including the complete window classes. Pugh is Coeditor of* The Smalltalk Report.

They are cofounders of The Object People Inc. specializing in introductory and advanced courses in Smalltalk, object-oriented programming, and object-oriented design.

Professors LaLonde and Pugh can be contacted at the School of Computer Science, Carleton University, Ottawa, Ontario K1S 5B6, Canada, by phone at (613)788-4330, or by email (bitnet:) at jpugh@carleton.ca.

EIFFEL, THE LANGUAGE

Dr. Bertrand Meyer
Prentice Hall, Englewood Cliffs, NJ, 1991

Reviewed by Steven C. Bilow

IN 1887, THE FRENCH engineer Gustave Eiffel defended himself against the Parisian artists who protested his creation of a huge iron tower in the center of the city. In response to their petition he stated: "Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant ? ... Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony ?" Eiffel boldly defended the premise that perfection in design was the quintessential combination of beauty, elegance, robustness, strength, harmony, and purpose. This was an admirable goal for Eiffel, the engineer, and the application of these concepts to software design is the equally admirable goal of Eiffel, the language, developed by Dr. Bertrand Meyer.

Dr. Meyer is well known to the object-oriented software community. His first book, *Object-Oriented Software Construction,* is a principal work in the field. His language, Eiffel, embodies several important software engineering concepts. Some were taken from earlier languages including Simula, Smalltalk, Algol, Ada, and CLU. But, regardless of their origin, the concepts behind Eiffel have influenced the software community as a whole. Dr. Meyer's new book is titled *Eiffel: The Language* and is intended to be the definitive reference for both users and implementors. It is an extensive rewrite of the language reference provided with the Eiffel distribution from Interactive Software Engineering. The book was published in September, 1991, by Prentice Hall.

The book is intended to be as ingenious as the language itself. It implements this uniqueness through the its structure. While frequently adding to the text's usefulness as a reference, the unconventional organization sometimes detracts from its readability. Dr. Meyer dislikes traditional language documentation in which software engineers must search through volumes of books for answers to simple questions. So, instead of writing several books he integrates his user's guide, tutorial, reference book, and philosophical statement into a single contiguous unit. His goal is to provide a complete reference book for anyone using, studying, or implementing tools for the language. It is extremely difficult to satisfy such a diverse intended readership and Dr. Meyer should be commended for his success. The biggest problem for his readers will lie in acclimating themselves to his system for maneuvering through the book.

The navigation system involves "road signs" placed in the left margin of the page. It is quite workable but initially a bit confusing. The reader is led through the book by eleven different road signs. Two of these indicate that the text is either a preview of coming ideas or a reminder of previously explored ones. The other nine denote that the text covers either a feature's purpose, examples, syntax, semantics, rules, comments, methodology, caveats, or ways of shortcutting the book. The concept is excellent but Meyer is addressing so many different audiences that moving through the text is slightly flustering and takes some practice.

The book is divided into five parts covering syntactic and semantic conventions, linguistic organization and architecture, internals, a description of the kernel library, and ten appendices. Each section contains between two and twelve chapters. This is not light reading, but its author has set himself a very specific goal and reminds us that his book is designed to be "against all odds, not TOO boring." In this, he has succeeded unconditionally.

It is necessary to stress that while the book is certainly not boring it is rather complex reading. The manuscript provided for this review numbers 594 pages plus thirty-five pages of preface. Those who desire only an overview of the language will require proficient mastery of the "sign post" notation. Those wishing to actually use or implement Eiffel will have an easier time since they will neither need, nor want, to circumvent the extensive detail. Regardless of the reader's level of expertise, the book provides significant insights and much new knowledge. But, like many of the best things in life, reading it will require work.

The book begins with a brief introduction to the language and describes its principle features. Unique concepts such as assertions, exception handling, contracting, and genericity are introduced as well as the more established ones like inheritance, polymorphism, and data abstraction. These form the first section of the book and lead into the subsequent, and more substantial, chapters. Readers who desire only a basic overview of Eiffel may actually find this section alone sufficient.

The second section deals primarily with the structure of the language. It presents concepts, syntax, and examples for each major linguistic element. Eiffel is based on a very extensive concept of *class.* An Eiffel class is much more than simply an abstract data type: it is the primal form from which all else is derived. Rather than limiting a class definition to data and *functions*, Eiffel classes consist of constructs for indexing, genericity, specific inheritance relations, instance creation, data and functions (collectively called features), invariants, and even the specific indication of obsolescence. Many of these constructs are optional but are provided for flexibility.

Among the primary class components are the group of constructs called *features*. These are similar to what Smalltalk groups into *methods* and *instance variables* or C++ calls *data members* and *member functions*. In Eiffel, the client-level distinction between data and algorithm is purposely clouded. An account balance in a banking application is the same to the outside world regardless of whether it is computed or simply stored away. Thus, to a client it is unnecessary for a supplier to distinguish between a function that returns the balance and a variable that simply stores it. On the level of supplier internals, however, there must remain differentiation. Eiffel implements this through four types of features: variables, constants, procedures (which do not return results), and functions (which do return results). These are detailed in Chapter 5 and expanded throughout the book.

Progressing to deeper levels of detail we are presented with extensive descriptions of every aspect of the language. In Chapter 6, inheritance is discussed in both its usage as a module extension mechanism and its use in type creation. The client/supplier relationship, and its provisions for "design by contract," is covered in Chapter 7. Chapters 8 through 13 complete the section with extensive discussions of routines, correctness, feature adaptation, repeated inheritance, types, and conformance.

Internals and libraries are the subjects of the third and fourth parts of the book. The third deals primarily with the internals of classes, control constructs, external language interfaces, and lexical details. This is what Dr. Meyer refers to as the meat of Eiffel. We are introduced to many of the elements that make the writing of programs possible including mechanisms such as sequencing, branch instructions, loops, and rescue clauses for exception handling. Object creation, duplication, and comparison are also discussed, as are feature calls and type checking.

The final section covers an aspect of Eiffel that is not, per se, part of the language. This is the basic set of class libraries. These Chapters discuss only the libraries that make up the basic Eiffel system. A future book, *Eiffel: The Libraries*, will discuss this aspect of the language in greater detail. The present book limits its discussion to those classes required by every implementation of the language.

Within the seven chapters that comprise this section many aspects of the Eiffel philosophy are revealed. Most notable are the discussion of the universal class, called ANY, and the descriptions of the classes for I/O, strings, arrays, and arithmetic. The section also includes a discussion of a set of classes called *persistence classes* that provide a secondary storage mechanism.

The discussion of Class ANY is among the more insightful aspects of the book. This class is a child of the class PLATFORM and a grandchild of the class GENERAL. The resulting hierarchy establishes the backbone of the class structure. Any class that does not have an explicit inheritance clause will be a child of ANY. While the object orientation of the language does not extend to the minute level present in Smalltalk or SELF, its inherent object structure does come close to that which the designers and users of those languages desire. Unfortunately, the chapter that discusses this important aspect of the language is limited to five pages and, while it is one of the rare succinct discussions in the book, a more extensive treatment would have given some significant insight into Dr. Meyer's concept of the language.

The final chapters discuss classes that are slightly less unique but just as essential. These include input/output, exception handling, arrays and strings, and the arithmetic classes like integers, reals, and doubles. For simplicity, there are a limited number of basic classes and redundancy is minimized. The book discusses these classes in the same concise manner as before only this time the brevity is refreshing.

Eiffel: The Language concludes with a set of appendices that discuss such elements of the language as style, history, references, and the development environment. Also included are summaries of such items as reserved words and syntax. Among the more useful appendices are those that assist the reader in migrating from Version 2.3 to Version 3 and vice versa. Each appendix is well focused and well written.

Eiffel is a unique and rigorous language and Dr. Meyer's book maintains those traits. The book is distinctive in its structure and hence requires a special approach on the part of the reader. While intended to be read from cover to cover, doing so may prove somewhat tedious unless one desires tremendous detail. Dr Meyer realizes this and has provided several methods for more general readers to circumvent the technicalities. His navigation system is a bit complex but adequately accomplishes his goal. The book is relatively difficult reading but those who tackle it will find much enlightenment. There is no question that this publication is a tremendously significant contribution to the literature and it comes highly recommended. It provides a definitive description of every aspect of the language. In comparison to similar language references, it is quite readable and Dr. Meyer should be applauded for his novel approach. Just as Gustave Eiffel promoted elegance, rigor, and robustness in architecture, Bertrand Meyer's Eiffel carries those characteristics into the world of software. I recommend the book to anyone interested in Eiffel, object-oriented design, or rigorous software engineering methods. I also recommend patience.

Steven C. Bilow is presently a Senior Technical Support Specialist for the Computer Graphics Group at Tektronix, Inc. in Wilsonville, Oregon, and an independant consultant in computer graphics software. His interests are in the areas of mathematical surface rendering and object-oriented architectures for graphics systems.

– Advertiser Index –

What's new?

C++

ObjectCraft announced new version of object-oriented CASE tool

On November 15, ObjectCraft, Inc. will begin shipping Version 2.0 of its C++ CASE tool, ObjectCraft. Version 2.0 incorporates several significant improvements to the existing product that have been requested by the users. The major new features include the ability to import existing C++ files into the ObjectCraft environment, print ObjectCraft diagrams, and write C++ methods inside ObjectCraft. ObjectCraft is a PC-based productivity tool that lets programmers develop object-oriented programs visually.

For further information, contact ObjectCraft, Inc., 2124 Kittredge St., Ste. 118, Berkeley, CA 94704, (415)621-8306.

ParcPlace supports team programming with new release of Objectworks\C++

ParcPlace Systems has announced a major upgrade to its integrated development environment for C++. Objectworks\C++ Release 2.4 now supports team programming and provides complete integration with popular UNIX development tools, cooperating with the UNIX environment and permitting tools such as 'make' to be used without modification. New features include increased performance and debugger enhancements for peer and light weight processes support.

For further information, contact ParcPlace Systems, 1550 Plymouth St., Mountain View, CA 94043, (415)691-6700.

Network Integrated Services announces model and simulation C++ class library

Network Integrated Services, Inc. is now shipping MEJIN++ Version 1.1, a 109-class library that allows programmers to use the finest features of the C++ language to develop mathematical, statistical, and queuing models efficiently.

MEJIN++ allows developers to reduce complex models to a collection of interacting entities at runtime. The main features are an exception handling mechanism, persistent data collections, statistics and math tools, and discrete event simulation. MEJIN++ includes object code libraries for Borland and Zortech compilers under MS-DOS and documented, portable C++ 2.1-compliant source code.

For further information, contact Network Integrated Services, Inc., 221 West Dyer Rd., Santa Ana, CA 92707-3426, (714)755-0995.

Rational offers C++ Booch Components

Rational Consulting announced that it is distributing and supporting The C++ Booch Components, a reusable software component library. The C++ Booch Components represent the second generation of a widely used and mature component library, the Ada Booch Components. The Booch Components are available on a variety of platforms including IBM PCs, Macintoshes, and UNIX workstations, as well as minicomputers and mainframes. The Booch Components provide a reusable, extensible class library of structures and tools implemented and delivered in C++ source code.

For further information, contact Rational, 3320 Scott Blvd., Santa Clara, CA 95054-3197.

Sequiter Software announces new CodeBase++ release

Sequiter Software announced the release of CodeBase++ 1.04, a C++ class library for database management, which now includes support for the Clipper .NTX index files. CodeBase++ gives C++ developers the flexibility of using the three most popular index formats: .NDX (Clipper, dBASE III+, IV), .MDX (dBASE IV), and .NTX (Clipper).

For further information, contact Sequiter Software, Inc., #209, 9644-54 Ave., Edmonton, Alberta T6E 5V1, Canada, (403)448-0313.

Object-oriented asynchronous communication library

Greenleaf Software, Inc. has released Greenleaf Comm++, a class library for asynchronous communications. As a C++ library, it provides a hierarchy of classes that give the programmer simple access and control of serial communications with or without terminal emulation. Classes are provided for serial port controls, modem controls, file transfer protocols, and calculation of check values. There are also classes that support hardware dependent features.

For further information, contact Greenleaf Software, Inc., 16479 Dallas Pkwy., Ste. 570, Dallas, TX 75248, (800)523-9830.

Smalltalk

First Class Software announces performance analysis tool for Smalltalk/V

First Class Software has announced Profile/V, an efficient, interactive performance analysis tool for Digitalk's Smalltalk/V Mac and Smalltalk/V 286. Profile/V helps programmers get the most out of Smalltalk/V by showing where time is being spent: both which methods are most expensive and which statements within each method are costliest. Profile/V also includes a novel filtering mechanism called "gathering" that helps users profile the recursive methods common in object-oriented programs.

For further information, contact First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, (408)338-4649.

Apprentice program for Smalltalk/V Windows

Knowledge Systems Corporation is now providing a new training program, "The Smalltalk Apprentice Program," for Digitalk's Smalltalk/V Windows. This program is a customized, project-focused training course devoted to both developing internal Smalltalk experts and advancing the specific corporate project with which

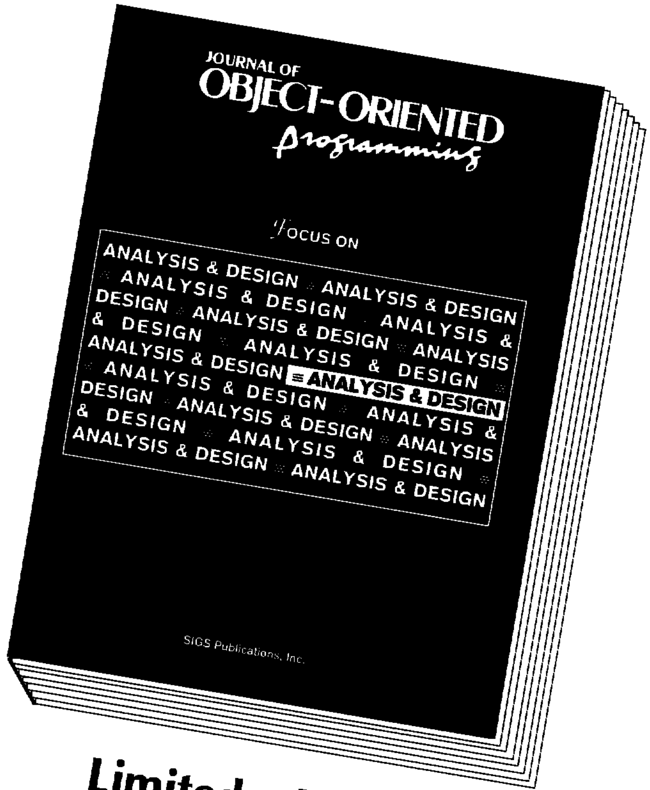
New!

The focus is on analysis and design

JOOP Focus on Analysis & Design gives you the what, where, when, how, and why of object-oriented analysis and design.

Published with the same editorial integrity as the Journal of Object-Oriented Programming, this expert-reviewed selection of editorial proceedings delivers the latest thinking, insightful perspectives, mind-opening techniques, and applicable case studies in this crucial stage of object development.

JOOP Focus on Analysis & Design discusses the most critical issues and provoking questions facing this process. Written by many of the originators of object methodologies — such as Grady Booch, Steve Mellor, Sally Schlaer — these articles define, demonstrate, simplify, compare, and contrast their approaches. This info-packed softcover book takes you through the inner workings of the process, explaining each step and concept, giving you a frame of reference you can draw from immediately.



Limited edition!

ORDER FORM

JOOP FOCUS ON ANALYSIS & DESIGN

❑ Yes, send me a copy of JOOP FOCUS ON ANALYSIS & DESIGN. Your satisfaction is guaranteed — your money will be refunded if you are not satisfied. Just return the book within ten days.

$29.00 — *Add $4.00 per copy for shipping and handling in the U.S., $8.00 in Canada, and $15.00 per copy for overseas mailing.*

Method of Payment
o Check enclosed *(payable to JOOP in $U.S. drawn on a U.S. bank)*
o Charge to my o Visa o MasterCard
Card number__________ Exp. Date____
Signature __________
o Bill me (shipped upon receipt of payment)

Return by FAX (212) 274-0646
or by mail to JOOP FOCUS ON A&D
588 Broadway, Suite 604
New York, NY 10012
(212) 274-0640

Name __________
Company __________
Address __________
City__________ State ____ Zip____
Country __________ Telephone ______
Signature __________

All orders must be signed to be valid. **D1LA**

Circle 45 on Reader Service Card

the students are tasked. Participants are provided with individual workstations in secure office space, access to KSC development staff expertise, and training within the context of their project. The Smalltalk Apprentice Program is also available for Objectworks/Smalltalk Release 4, Smalltalk/V PM, and Smalltalk/V 286.

For further information, contact Knowledge Systems Corporation, 114 MacKenan Dr., Ste. 100, Cary, NC 27511-6446, (919)481-4000.

OODBMS

KnowledgeMan and GURU introduce BLOBs, multimedia, and object-based technology in Version 3.0

Micro Data Base Systems, Inc. is now shipping version 3.0 of both KnowledgeMan and GURU. KnowledgeMan is a relational database management system for business applications. GURU is a comprehensive expert system environment. Version 3.0 allows developers to incorporate object-based elements into their applications.

KnowledgeMan and GURU are both available for single-user MS DOS-based PCs, OS/2, most popular LANs, DEC VAX/VMS, and Sun UNIX environments.

For further information, contact Micro Data Base Systems, Inc., Two Executive Dr., P.O. Box 6089, Lafayette, IN 47903-6089, (317)463-2581.

Servio announces first commercially available Kanji object database

Servio's Gemstone now supports manipulation of extended UNIX code (EUC) standard Japanese character strings. Kanji support is immediately available in Japan and will be made available worldwide this fall. Gemstone is an object database management system that merges advanced object-oriented technology with a full-featured, multiuser database management system.

For further information, contact Servio Corporation, 1420 Harbor Bay Pkwy., Alameda, CA 94501, (415)748-6200.

Object Databases announces the release of GTX object repository

GTX is a multimedia object repository providing real-time performance to mission-critical applications and commercial products. GTX provides a high-performance object repository to act as the underlying data store for multimedia applications. GTX is a VAX/VMS database server that supports large multimedia databases consisting of complex, linked data types with image, voice, and video objects; fault-tolerant network applications requiring a strong transaction model and detailed audit trails; real-time, high-volume data capture with a requirement for immediate query capability; and recall of temporal object versions required for group work and online back-up. GTX's most important feature is intrinsic versioning, the automatic generation and management of historical object versions.

For further information, contact Object Databases, 238 Broadway, Cambridge, MA 02139, (617)354-4220.

ONTOS, Inc. ships new version of object database for C++

ONTOS, Inc. announced it is shipping to its customers Release 2.1 of its ONTOS object database management system for UNIX. This release was designed to address the needs of the growing number of ONTOS customers ready to deploy distributed applications, such as network management and data integration systems. ONTOS Release 2.1 also adds support for IBM's RISC System/6000 workstation.

The ONTOS database was designed as a distributed, client-server database for C++ programmers and provides object-oriented, graphical tools to assist the database layout, object manipulation, and application development process. Key features of ONTOS Release 2.1 include open access to its internal data structures, or "metaschema," flexible and optional transaction and concurrency control models, extensible storage management, and an integrated object SQL.

For further information, contact ONTOS, Inc., Three Burlington Woods, Burlington, MA, 01803, (617)272-7110 ext. 500, or (800)388-7110 ext. 500.

OO CASE

Object-oriented support added to CASE tool

Object-oriented support for software development has been added to the Macintosh CASE tool TurboCASE. TurboCASE 4.0 supports five new editors: four graphics editors create different class diagrams and a fifth editor, a dictionary, gives the user the ability to define classes. The diagrams, which show class specifications and relationships, are integrated through a project database providing multiple views of the software design. TurboCASE 4.0 is an integrated tool following the standard Macintosh user interface. The package supports the most widely used methodologies for analysis, design, and modeling.

For further information, contact StructSoft, Inc., 5416 156th Ave. SE, Bellevue, WA 98006, (206)644-9834.

Visual Programming

TGS Systems Prograph 2.5 Release adds suite of new features

TGS Systems introduced Version 2.5 of Prograph — its Eddy award-winning, object-oriented visual programming environment for the Macintosh. In addition to adding a wide array of new features to the Prograph environment, this new version provides high-level System 7.0/IAC support and a database engine. Prograph will also connect to SQL databases through interfaces for DAL and Oracle; these interfaces are part of the company's new line of add-on products.

For further information, contact TGS Systems, 2745 Dutch Village Rd., Ste. 200, Halifax, Nova Scotia B3L 4G7, Canada, (902)455-4446.

CAREER OPPORTUNITIES & TRAINING SERVICES

To advertise, call Diane Morancie at 212.274.0640.

Statement of Ownership
Management and Circulation
Required by 30. U.S.C.3688

1. Title of publication: the Journal of Object-Oriented Programming
2. Date of filing: 10/21/91
3. Frequency of issue: Monthly except for March/April, July/August, and November/December
   A. Number of issues published annually: 9
   B. Annual Subscription Price: $59
4. Location of known office of publisher: 588 Broadway, Suite 604, New York, NY 10012
5. Location of headquarters or general business offices of the publishers (not printers): 588 Broadway, Suite 604, New York, NY 10012
6. Name and address of publisher, editor, and managing editor: Publisher, Richard P. Friedman, 588 Broadway, Suite 604, New York, NY 10012; Editor, Richard Wiener, 588 Broadway, Suite 604, New York, NY 10012; Managing Editor, Elisa Varian, 588 Broadway, Suite 604, New York, NY 10012
7. Owner: SIGS Publications, Inc., 588 Broadway, Suite 604, New York, NY 10012; Richard Friedman, 588 Broadway, Suite 604, New York, NY 10012; Richard Wiener, 588 Broadway, Suite 604, New York, NY 10012
8. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of lands, mortgages, or other securities: None.
9. For completion by nonprofit organization authorized to mail at special rates (Section 424.12 DMM only): Not applicable
10. Extent and nature of circulation: Average number of copies each issue during preceding 12 months: A. Total no. copies printed (net press run): 16,213; B. Paid circulation: 1. Sales througl dealers and carriers, street vendors and counter sales: 1,407; 2. Mail subscriptions: 8,565; C. Tot paid circulation: 9,772; D. Free distribution by mail, carrier and other means, samples, complem tary, and other free copies: 4,044 E. Total distribution (sum of C and D): 13,816; F. Copies not tributed: 1. Office use, left over, unaccounted, spoiled after printing: 1,377; 2. Returns from new agents: 1,020; G. Total (sum of E and F, should equal net press run shown in A): 16,213.

Actual number of copies single issue published nearest to filing date: A. Total no. copies printed (net press run): 19,625; B. Paid circulation: 1. Sales through dealers and carriers, street vendors and counter sales: 2,517; 2. Mail subscriptions: 10,486; C. Total paid circulation: 13,0(D. Free distribution by mail, carrier and other means, samples, complementary, and other free copies: 2,953; E. Total distribution (sum of C and D): 15,956; F. Copies not distributed: 1. Offi use, left over, unaccounted, spoiled after printing: 1,130; 2. Returns from news agents: 2,539; C Total (sum of E and F, should equal net press run shown in A): 19,625.