



## DESIGNING SCIENTIFIC COMPONENTS

By Paul F. Dubois

**C**ORRECTNESS IS MORE PRECIOUS TO SCIENTIFIC PROGRAMMERS THAN IT IS TO BUSINESS PROGRAMMERS BECAUSE OF THE GREAT DIFFICULTY IN DISTINGUISHING

between programming errors, errors in modeling, and errors in algorithms (see Figure 1). We've all sat in meetings and discussed whether a peculiar wiggle in a graph represents an algorithm problem (such as neglecting to include a possibly negligible term) or a modeling one (such as ignoring a possibly important physical process). Usually it turns out to be nothing so esoteric: we come back the next week and learn that it was a bug.

Reliability is the principle benefit of reuse, not the saving of coding time.<sup>1</sup> Given that correctness is so important to us, you would think that reusing existing, reliable components would be the dominant behavior in our field, but this is far from true. The reasons why are more technical than social, but we can solve the problem. As we move into the era of open-source science, however, we must not repeat our poor history in this regard.

### Learning from history: Libraries and context

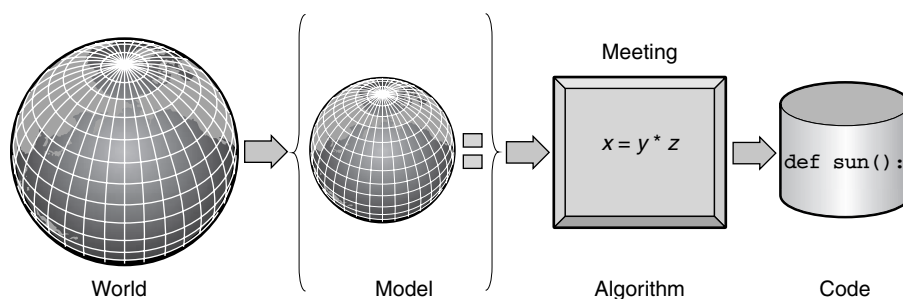
Mathematical libraries are often thought of as a reuse success story. Many people use commercial and government-

sponsored libraries such as NAG, IMSL, and Lapack. However, their market penetration is far below what it could be—and not just because of cost or a design flaw.

For the first eight years of my career, I worked in a group that was responsible for helping people with numerical mathematics and statistics. We saw that even when the central computing facility paid for the library, component reuse was poor. People would come in for advice, and we would find that they had copied an elementary algorithm from a book, implemented it (not always well), and were startled when it didn't work on their problem.

If they knew about a library routine, they usually said they found the large number and variety of arguments to be too intimidating and that they didn't know how to set some of them. Having decided in frustration that what they wanted couldn't possibly be so complicated, they had set out to "roll their own." Even in the presence of their own failure, they were inclined to believe that they had just overlooked one simple thing and that a sophisticated library routine would be overkill, so would we please just fix it?

Most of the reuse that does occur in our field is confined to routines that are leaves in the call tree. We get the most use out of the simplest components that represent the least technical sophistication. We use components that represent the most expertise—such as ordinary differential equation (ODE) solvers, nonlinear optimization routines, and integration routines—much less frequently than simple matrix solvers or special functions. This unfortunate fact is an



**Figure 1.** Scientific programmers have trouble distinguishing the source of an error because of the many stages of approximation between reality and the simulation's output.

## Dave's Sideshow

### Adventures in type systems

For the past year, I've found myself deeply buried in the land of programming languages and type theory. How did I get here? What does this have to do with scientific computing? Have I gone crazy? Well, other than not being sure about that last question, a strange path has led to this odd place.

To pick up the trail, wander over to the ongoing problem of "systems integration." Maybe you want to make your next application a mix of Fortran IV and C++ templates. Or, perhaps you want to build a data analysis system an Intercal interpreter can control. Or, perhaps you want to be the first person to write a molecular dynamics code entirely in middleware (a bold feat of extreme middleware-metaprogramming, perhaps). To solve these kinds of problems, you would often turn to software development tools such as code translators, wrapper generators, component builders, and so forth. In fact, you might be inclined to write your own tool for the task. Write a small parser, throw in some regular expressions, mix in some files, add a bunch of print statements and voilà—you've got yourself a new systems integration tool.

In my case, this scenario might describe the SWIG project—a tool I created several years ago to build scripting interfaces to C++. Although creating a tool that basically worked was relatively easy, it also had many nasty corner cases waiting to bite people who strayed too far from the beaten path. Clearly, it would have been nice to fix these problems, but I could never quite wrap my head around them well enough to make a fix. Apparently the task of writing a robust systems integration tool was much harder than I thought.

Cutting to the chase, I recently sat in on a class about type theory. Although it is difficult to see how type theory might be immediately applicable to scientific computing, it got me thinking about tools and what I had done with my



systems integration project. In some sense, the whole problem of systems integration is nothing more than an issue of type systems (hooking types together, type conversion, type checking, and so on). Therefore, the class succeeded in changing my view of what I had been doing, and it let me see how to fix all those nasty corner cases. In many respects, the solution had been staring me in the face all along—I just couldn't see it. The bottom line? It is never too late to pick up new tricks from other areas of computer science.

### Systems for computational science

On the subject of computational science education, our department recently updated its numerical computing courses to have more of a software/systems component. I'm currently scheduled to teach an experimental course on systems programming for computational science—a mix of topics that span operating systems, networking, and scientific computing. To be honest, I'm not entirely sure what this course will cover. For now, I'm just telling people that the course is intended to give future computational scientists enough systems background to be "dangerous." I'd love to hear your ideas about what should or should not be covered in such a class. I'm just hoping that the subject matter is still legal by next spring.

### New and interesting

Pyrex is an extended version of Python that easily lets you write C extensions and seamlessly mix them with Python code (see [www.cosc.canterbury.ac.nz/~greg/python/Pyrex](http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex)).

Maple 8 was just released (see [www.maplesoft.com](http://www.maplesoft.com)).

Paul tried Ximian Evolution ([www.ximian.com](http://www.ximian.com)), an open-source tool similar to Microsoft Outlook. So far, he is delighted. For some of us, the lack of an Outlook equivalent has been Linux's weak point. Evolution gives you not only email but calendar, task, and contact management.

evitable consequence of using first-generation languages. If our answer to the cost and unavailability of commercial libraries is to create open-source libraries in the same languages, we will fail in the same way. I'm not implying that we can't use legacy code; rather, we must package it correctly.

To be sure, cost is an issue for some people, and the availability of products on odd hardware is an even greater problem. Even scientists at "wealthy" labs have collaborators in

the "poor" third-world arena of universities (not to mention the real third world) and therefore have a great reluctance to rely on commercial components. It would be a real boon if we could create, as a community, an open-source series of libraries in mathematics and statistics.

Although I concentrate on mathematical software in this article, the same principles apply to physics middleware. In general terms, there is potential here because listing things

that lots of people do in a given problem area is easy. For example, for those doing time-dependent simulation of fields using grids, we see a lot of

- Calculating time evolution by diffusion
- Integrating fields over regions
- Interpolating fields to nongrid points

Thus, I hope that what I suggest here is of use beyond the mathematical context.<sup>2</sup> The idea of these ideas being usable in a variety of contexts is in fact a perfect metaphor for the central problem of software reuse, the gist of which is that every function has context. By context, I mean the collection of information that must be available to a function for it to do its job. Thus

- For  $\cos(x)$ , the context of  $\cos$  is  $x$ .
- In  $dy/dt = f(y, t)$ ,  $f$  might need access to a wide variety of information needed to compute the rate equations, such as material properties or external boundary conditions.

Fortran and C have two methods for supplying context to a function: the function's argument list and static data areas (for example, common blocks in Fortran) of either function, global, or file scope.

Using a static data area limits reusability and hinders maintenance. Routines to get and restore the calculation's state must be provided. However, an improvement to the algorithm that changes the required state could change the signature of these get-set routines. Furthermore, the recursion on the call might be hidden and thus a source of mystery when someone tries to use it, not knowing that it is already in use.

Thus, static data areas are out of the question in most cases, which leaves only the argument list. It is therefore no surprise, and implies no criticism of their authors, that routines from libraries such as NAG and IMSL have many, many arguments.

### Case study: Integrating functions of one variable

At first glance, the design for a routine that integrates a function of one variable seems simple:

```
real function integrate (f, a, b)
external f
real a, b
```

However, two problems exist here. One, where do we get  $f$ 's context?  $f$  is going to be called  $f(x)$ . In Fortran or C, the

only possible context other than  $x$  is common. Even when solvable, such problems represent inertial barriers to reuse. Two, how do we control the algorithm and get results? Different algorithms have different requirements. For example,

- Algorithm 1: Fixed integrator, Gaussian 25-point method. This algorithm always succeeds if the function can be evaluated on  $(a, b)$ . No controls on the algorithm or optional outputs are available.
- Algorithm 2: Adaptive integrator. This algorithm needs error-tolerance specifications as well as the amount of work it's expected to do before giving up. It returns optional information on the intervals used and error estimates but can fail to get an answer to the tolerance requested.

Here is a call to an adaptive integrator in the NAG C library:

```
Nag_QuadProgress *progress;
NagError fail;
double a, b, ans, err, epa, epr;
Integer npts;
double (*f)(double);
...
d01ajc (f, a, b, epa, epr, npts, &ans, &err,
        progress, &fail);
if (fail.code != NE_NOERROR) { error...}
```

The inertial barrier to using such a routine is substantial. The optional information structure returned in `progress` will be leaked memory if we call `d01ajc` again, so we must add more coding for memory management. This routine cannot pass any context for  $f$ .

### Arguments plus types equals inertia

This example is typical of how programmers get context to the function. They must add more arguments (or have fewer arguments but have them be complex structures), often quite artificial in nature. If arrays must be returned, programmers have a choice of risking memory leaks if the called routine allocates the memory or adding complexity to the caller's task and error checking to the callee if the caller allocates the memory.

All of this amounts to what I call inertia. By analogy with the physics meaning of the term, inertia requires force to overcome it—in this case, effort by the routine's potential user to learn how to supply all those arguments and to supply values for them. Initial values for algorithm control—such as “initial step size” or “maximum number of inter-

vals”—might be quite mysterious to a person not familiar with the mathematics involved.

### Methods have their context in the instance

Object-oriented programming offers a solution to the context problem. If a method is in a class, and we’re using true object-oriented programming without modifiable data outside of class instances, then that method’s context is contained in the argument list together with the data for the instance on which the method is invoked.

Some C libraries attempt to provide context by providing a user-defined argument to be passed through as a void pointer. This structure is not really natural, and such an approach doesn’t deal with issues like optional outputs and memory leaks.

Let’s do this example in C++ and the next one in Python; focus on the technique, not the language.

For example, consider method *f* in class *Aclass*:

```
class Aclass {
    float f (float x) const { ...}
    —
}
```

When we have an object *a* of class *Aclass* and we call *a.f (x)*, we can use the argument *x* plus any methods or data available in *a*.

### Make the integrator a class

The solution then is to make the integrator a class. We start with an abstract base class that represents the general concept of integrating the functions of one variable. (In an actual C++ library, we would probably template this function on the type of argument so that we could reuse the coding for different types such as *float* and *double*. However, for clarity, I omit that here.)

```
class Integrator {
    virtual double integrate (T t, double (*T f)
                             (double), a, b) ;
    //calls t.f (x) to calculate integral;
    virtual ~Integral();
    virtual bool error_occurred ();
}
```

The idea is that we will inherit from this class to specify a particular algorithm. In C++, making the destructor virtual is important, so we can specialize the release of workspaces

in the children:

```
class GaussianIntegrator: public Integrator {
    // 25-point Gaussian integration
    ...implement integrate
    bool error_occurred () { return false; }
}

class AdaptiveIntegrator: public Integrator {
    // Adaptive integrator
    ... implement integrate, ~AdaptiveIntegrator
    ... add constructors, methods, to hold control
        parameters
    ... add methods to return results
}
```

The adaptive algorithm’s control parameters would be kept as private instance data with reasonable defaults supplied, such as

```
double absolute_error = 1.e-6;
double relative_error = 1.e-5;
```

with public accessor methods. The methods for setting them can check the supplied values for validity:

```
void set_error_criteria (double abs, double rel);
// checks that abs >= 0.0, rel > 0.0
```

Other methods to return optional output can also exist:

```
const vector<float> errors() const;
// returns error estimates on each interval
used
const vector< pair<float,float> > intervals()
const;
// returns intervals used
```

To use the integrator, we create it and apply it to a method and its specific instance:

```
Aclass a;
AdaptiveIntegrator ai;

answer = ai.integrate (a, Aclass::f, 0., 1.);

if (ai.error_occurred() ) { ...}
// can examine ai.intervals(), etc.
```

## Advanced Design Considerations

Experienced C++ users might have questions about the design of the classes shown in the integrator example, such as

*Why have a test for whether an error occurred; why not just throw an exception?*

This is a matter of philosophy. Some believe that we shouldn't use exceptions for "normal" processing. Throwing an exception if an iteration failed to converge, for example, is using exceptions rather than regular flow control. Using exceptions could have performance implications.

*Would it be better to have `integrate` be a routine that does not return a value and then have a separate query `integrate1()` to return the answer if no error occurred?*

That would be in line with Bertrand Meyer's idea of separating queries and commands.<sup>1</sup> It allows clean design by contract. I didn't do it here because less experienced scientists would find it surprising and that would divert attention from my main point.

*Isn't it a red flag in OO design when your class has a name that is a verb?*

Yes, and sometimes in designing these kinds of mathematical classes, it is tempting to use names such as `Integrate` or to have abstract classes that have other red-flag properties such as only having one method. If you think of these classes as representing an expertise, then a name like `Integrate` or `Integrator` is short-hand for some concept whose name is a noun, such as `IntegrationFacility` or `IntegrationExpert`, and their lack of methods at the top level reflects the fact that only specific algorithms have specific control and reporting facilities.

Another way to answer such questions is, "Sure. Whatever." There is no one right answer in OO design. Learn a methodology you are comfortable with and use it consistently so that your library users know what to expect.

Note also that the clumsiness with the integrand being passed in two separate parts (`a` and `Aclass::f`) is just a C++ problem because there is no standard class for a bound method. You could invent one if you wanted.

### Reference

1. B. Meyer, *Reusable Software*, Prentice Hall, Upper Saddle River, N.J., 1994.

Note the reduction in inertia for new users. The integrator class can supply defaults for things such as error control. Complicated data structures can be returned as options on demand, but beginners can ignore them (see the "Advanced Design Considerations" sidebar).

### Case study: Solving systems of ODEs

Let's now consider the solution of an initial value problem  $dy/dt = f(y, t)$ , with  $y = y_0$  at time 0. Using the method of lines, we end up solving a system of ordinary differential equations where  $y$  is a vector of length  $n$ .

Many algorithms for this problem exist, generally divided into explicit and implicit methods. Implicit methods must remember one or more past values of  $y$ . The solver can represent tens of man-years of work on algorithms and heuristics, so making it reusable is imperative.

Here is the traditional approach:

- Write  $f(y, t)$  as a function.
- Call the solver subroutine with initial value, initial time step, and  $f$  as arguments.
- Solver updates  $y$  and  $t$ .
- The user supplies extra arguments to hold state and control the algorithm, or the solver uses its own static data area.

The dialog required between the solver and the user is substantial:

- Error control, reporting, and handling
- Interpolation of desired output values
- Control of step size, signal discontinuous changes in  $f$ , and so on

### The traditional approach

Consider a simple Runge-Kutta solver from the IMSL Fortran library:

```
subroutine ivprk (ido, n, fcn, t, tend, tol,
                 param, y)
```

The argument `ido` is both input and output:

```
1 = initial entry
2 = normal reentry, return value after 1
3 = final call to release workspace
4, 5, 6 on output, error occurred.
```

Auxiliary routines can be used to supply our own work-



space and control the error norm used. Variables  $t$  and  $y$  are output as well as input.

The argument `param` (50) is a floating-point array. The fact that the user must initialize `param` (which is considered to be both input and output) requires carefully reading the example. Some of `param` is input, some is output, and some is unused. The user must hold on to  $y$ , `param`,  $t$ , `tend`, and `tol`. There is no default value for `tol`.

The user must understand all of this before first use on even the simplest problem, which raises the inertial barrier to using this routine.

```
class ODESolver:
    def __init__ (self, integrand,
                  start_time,
                  initial_value,
                  error_tolerance=1.0e-6,
                  ):
        self.integrand = integrand
        self.time = start_time
        self._error_tolerance = error_tolerance
        self.y = initial_value[:]

    def advance (self, tend):
        "Integrate up to time tend."
        pass # left abstract here

    def tolerance ():
        "Return the present tolerance"
        return self._error_tolerance

    def set_tolerance (self, tolerance):
        "Set the error tolerance to tolerance."
        assert tolerance > 0.0
        self._error_tolerance = tolerance
    ...
```

## ODE solvers as classes

An object-oriented design for an abstract class `ODESolver` (this time in Python) might resemble Figure 2. I omit all but one pair of accessor functions; the point is that the various controls live in the instances and can be set on creation or changed later and that the state between steps is kept in the instance.

### The Runge-Kutta algorithm as a child of `ODESolver`

To specialize to the RK algorithm in particular, we define a class `RungeKuttaSolver` as a child of `ODESolver`. It then implements the time advance that was left undefined in the parent. Each child of `ODESolver` could have additional arguments to its constructor, additional methods for accessing them, and additional methods for optional outputs. However, all the state between calls to integrate is held in the instance, and when the integrator is no longer needed and is destroyed, all the state information goes with it. (Sensible default values for controlling the algorithms are provided where possible.) The actual use of the solver becomes much easier:

```
solver = RungeKuttaSolver (f, y, t0);
solver.set_tolerance (1.e-8);
solver.advance (tfinal);
print "Solution at", solver.time, "is",
      solver.y
```

## Wrapping good software

Does this mean I should write the RK algorithm myself? No, that would be a bad idea—other people know more

Figure 2. Object-oriented design for `ODESolver`.

about this than I do. As I mentioned earlier, the only real problem is in the packaging. The solution is to implement our class using others' subroutines. As a result, we get a working equivalent that can't leak memory, is easier to use, and is safer.

Someone still has to understand the original routine, but now it is the `RungeKuttaSolver` class's designer rather than the end user, and that designer only has to do it once. When the user comes back months later, he or she might see an exception trap for a failed step,

```
try:
    solver.advance (tfinal)
except solver.error, explanation:
    print explanation
```

but not the mysterious sort of thing he or she would have seen with the IMSL routine:

```
if (ido .gt. 3) then ...
```

Being able to make instances of the solver permits multiple uses per code.

## Avoiding Fortran++

In designing your new middleware, watch out for Fortran++. Fortran programmers who have learned an object-oriented language sometimes write something such as

```
def takeStep (dt, A, B, C, D, E, F, G) :
    AOLD = A[:]
    BOLD = B[:]
    ...
    GOLD = G[:]
    A[:] = AOLD[:] + dt * ...
```

Here *A*, *B*, *C*, ... taken separately have no real meaning, but in the aggregate, they are a description of a physics package's state. It's better to just admit it:

```
class HydroState:
    has data members A, B, C, ..., G
    def clone (self):
        (or in C++ the copy constructor)
        embodies once and for all saving the old state
```

Each instance represents the state at a certain time. Here, the rate equations for advancing the pieces are methods.

## Algorithms without containers

Another way reuse is diminished is in the use of container classes. Fortran programmers, think of it this way: if you program a routine to handle double-precision arrays, you can't use it with single-precision arrays. The concept of an array of a given precision is a *container class*. As people learn object-oriented programming, they want to produce abstract container classes that correctly express the concepts with which they are working. But this means less mathematical software is easily available to process these new kinds of containers.

The Standard Template Library in C++ has pioneered a new paradigm of programming, by separating the containers from the algorithms.<sup>3</sup> For example, using the STL, the algorithm `sort` can sort all kinds of containers:

```
sort (vector<double> d);
sort (queue<tasks> t);
```


Through the wonder of C++, you don't have to actually write all possible versions of this such as `sort<vector<double>>(d)`. All that `sort` needs from `T` is that `T` have the interface to supply `sort` with what it needs to do its job, so `class T` must be able to

- Tell the size of the list to be sorted
- Access elements by index
- Compare elements
- Create a temporary to do a switch

We can copy this approach. The trick is to be able to document exactly what interface a class must have to plug and play with our algorithm. An application class that can do what must be done, but that uses a different interface, can be easily wrapped in an adaptor class.

(Geoff Furnish wrote a good article on this subject.<sup>4</sup> Andrew Lumsdaine's article and the Matrix Template Library's Web page [www.lsc.nd.edu/research/mtl](http://www.lsc.nd.edu/research/mtl) also have an excellent discussion of these ideas.<sup>5</sup> The MTL is a library of mathematical classes based on STL ideas.)

An essentially functional C++ version of Lapack is now being replaced by a new linear algebra library based on a template approach (<http://gams.nist.gov/tnt>).<sup>6</sup> It is scheduled for release about the time that this article will appear, so I have not had an opportunity to review it.

I have been an advocate of the open-source approach for science, but I think there is no denying the danger of group projects being driven to the lowest common denominator in terms of design and implementation choices. Designing proper middleware in mathematics, statistics, and science will require experienced leadership, not just the consensus of the mob in a mail list. If good computer scientists act alone, they will produce nice interfaces to second-class contents. Only by working together across disciplines can we achieve quality, reusable components on which we all can rely. The EiffelMath library ([www.eiffel.com/products/math.html](http://www.eiffel.com/products/math.html)) has put into practice the principles I discuss here.<sup>2</sup> 

## References

1. B. Meyer, *Reusable Software*, Prentice Hall, Upper Saddle River, N.J., 1994.
2. P.F. Dubois, *Object Technology for Scientific Computing*, Prentice Hall, Upper Saddle River, N.J., 1997.
3. A. Stepanov and M. Lee, *The Standard Template Library*, tech. report HPL-95-11, HP Laboratories, Menlo Park, Calif., 1995.
4. G. Furnish, "Container-Free Numerical Algorithms in C++," *Computers in Physics*, vol. 12, no. 3, May/June 1998, pp. 258–266; [www.aip.org/cip/source.htm](http://www.aip.org/cip/source.htm).
5. J. Siek and A. Lumsdaine, "The Matrix Template Library: Generic Components for High-Performance Scientific Computing," *Computing in Science & Eng.*, vol. 1, no. 6, Nov./Dec. 1999, pp. 70–78; [www.computer.org/cise/cs1999/pdf/c6070.pdf](http://www.computer.org/cise/cs1999/pdf/c6070.pdf).
6. J. Dongarra, R. Pozo, and D. Walker, "LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra," *Proc. Supercomputing93*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 162–171.

Paul F. Dubois is a mathematician/computer scientist in the Advanced Software Technologies Group at Lawrence Livermore National Laboratory. Contact him at the Ctr. for Applied Scientific Computing, Lawrence Livermore Nat'l Lab., Livermore, CA 94566.