



## PYNSOL: OBJECTS AS SCAFFOLDING

By Michael Tobis

**S**OMEHOW THE FATES ALWAYS HAVE ME LIVING IN COLDER CLIMATES THAN I WOULD PREFER, BUT I CHOOSE TO BE PROACTIVE ABOUT IT. I'VE NOTICED, ON VISITING MORE TROPICAL

places, that palm trees are much more common there than in places with severe winters. I conclude that severe winters are the result of inadequate attention to modern civic planning: places that are too cold simply haven't planted enough palm trees.

Cold places, obviously, can dramatically reduce their snow-removal budgets by planting more palm trees. So far, my efforts to convince the cities of Chicago, Madison, Montreal and Ottawa, which I have called home at various times, to use these proven contemporary methods of climate enhancement have been rejected. This stubborn attachment to obsolete horticultural practice continues to interfere with the quality of life in these otherwise wonderful cities.

No, of course I'm not serious. Consider, though, the possibility that the use of object-oriented programming in high-performance scientific software is sometimes the result of this same sort of thinking. Richard Feynmann called this "cargo cult" science ([www.physics.brocku.ca/etc/cargo\\_cult\\_science.html](http://www.physics.brocku.ca/etc/cargo_cult_science.html)); a cargo cult is an isolated tribe that worships airplanes and builds runways in the hope that one will land. Object-orientation doesn't cause good design any more than runways cause aviation or palm trees cause warm weather. Although most successful large software engineering efforts of the past 15 years have been built on the foundation of object-oriented programming, it doesn't follow that objects are a precondition of successful software design.

I present here a design for a software framework in which object-orientation is used intensively during the construction of an executable but where there is no runtime presence of objects at all. Indeed, they can't be present because the final executable will be compiled and linked from Fortran 77 source. This counterintuitive strategy emerges from the problem domain and its constraints; it is presented here both

for its intrinsic interest and for the new light it sheds on how object-orientation can be useful in scientific programming.

The hypothesis I present here isn't that object-oriented programming is generally inappropriate—rather, I propose that the effective use of object techniques is a consequence of good design, not a precondition for it. Both runways and objects have their places, but those places aren't universal. Objects are most useful in the construction of codes, not in their execution—they're the scaffolding, not the edifice.

### Formula Translation Redux

The primary purpose of many physical models is to support reasoning about the real world. Typically, the investigation begins with a mathematical representation of boundary conditions, forcings, or parameterizations of unresolved processes that haven't previously been attempted. These representations are terse mathematical expressions, often literally scribbled on a napkin.

We can find the intention to convert such mathematical expressions to code in what passes for ancient history in computer science; Fortran, the first successful high-level language, was intended to be a "formula translator." This early implementation of the "separation of concerns," which, as Dijkstra pointed out,<sup>1</sup> is the necessary precondition of coping with complexity, hid the CPU architecture's details from the scientific programmer.

With the arrival of vector machines and then of multiprocessor machines with various memory models, this separation of concerns broke down. For almost two decades, we've seen no satisfactory formula translators for the most demanding computations that require the most complicated platforms.

No general solution is known for determining an optimal or even satisfactory way of recasting an arbitrary algorithm to an arbitrary multiprocessor architecture, and probably no such solution is possible. However, there could well be strategies for doing so in specific cases of practical importance. PyNSol is intended to address many such cases.

### PyNSol

PyNSol, the Pythonic Numeric Solver (<http://geosci.uchicago.edu/~tobis/psorland/pynsol.html>), is an application

development environment for finite difference and finite volume numerical methods. Currently under development at the Climate Systems Center in the Department of Geophysical Sciences at the University of Chicago, PyNSol is targeted to support researchers in the environmental sciences, but it could also be useful in other settings. PyNSol is discussed here primarily as an example of an alternative approach to scientific computing; this discussion's objective is to cast a fresh look on the relationship between high-performance computing and object-oriented computing. Interested readers can track PyNSol's progress at <http://geosci.uchicago.edu/~tobis>. When released, it will be available through the Climate Systems Center at the University of Chicago (<http://climate.uchicago.edu>).

PyNSol's initial release supports only regular (logically rectangular) grids. A substantial fraction of the real work in environmental physics and chemistry consists of such models or sets of them with relatively loose coupling among them. In particular, PyNSol doesn't support adaptive mesh refinement, nested grids, or finite elements, but the approach I describe here could be extended to more sophisticated numerical methods. Indeed, the inspiration for PyNSol is in large measure due to Robert Kirby and Matt Knepley's work on FIAT (the FInite Element Automatic Tabulator; [www.cs.uchicago.edu/files/tr\\_authentic/TR-2004-04.pdf](http://www.cs.uchicago.edu/files/tr_authentic/TR-2004-04.pdf)), a very general finite element solver that also begins with passing a terse representation of a system to a Python interpreter.

Scientists who study environmental continuum problems often work with multiple phenomena that operate on a large, shared data set—the system state—but on multiple time scales. To avoid unreasonably slow performance, these scientists must factor out phenomena that don't operate on the fastest time scales, resulting in hybrid methods in which a multiplicity of spatial-differencing schemes, parameterizations, and time steps must be woven into a single executable.

Typical representations of such codes are difficult to produce via conventional methods and often leave the simulation's underlying logical structure opaque and difficult to work with. One of PyNSol's objectives is to expose the structure of the algorithm's multiple time sequences while hiding discretization and processor details. Another is to enhance the visibility and understandability of the top-level time-stepping control structures. Essentially, PyNSol attempts to revert to the original goal of high-level programming languages (formula translation), although it's hardly unique in this effort.<sup>2-4</sup>

Many scientists find the Python programming environment to be enormously more productive than the alternatives with which they're familiar (see [http://python.oreilly.com/news/python\\_success\\_stories.pdf](http://python.oreilly.com/news/python_success_stories.pdf)).<sup>5</sup> Much of its

appeal is in its support for an interactive development style, which is similar to Matlab and Mathematica (see Technology Reviews on p. 14 for a comparison). Capitalizing on this, PyNSol endeavors to expose the domain-specific language at the Python interactive prompt. The model developer can then inherit time- and space-differencing schemes from previous PyNSol users.

### Example

Let's look at an example of some code PyNSol can already handle. Specifically, we'll focus on the simplest possible geophysical problem, the *linearized shallow-water problem*, which represents the limiting case of shallow ripples on a shallow pond as the ripples become very small.

We can express this system in a way that is instantly comprehensible to any fluid dynamicist:

```
u.t = - h.x
v.t = - h.y
h.t = - csq * ( u.x + v.y )
```

Here,  $u$ ,  $v$ , and  $h$  are fields ( $x$ -velocity,  $y$ -velocity, and perturbation height expressed as a fraction of mean fluid depth) distributed over a two-dimensional surface, and  $csq$  is a constant. The "dot" notation indicates a simple derivative;  $u$ ,  $v$ , and  $p$  are prognostic fields in that their time derivative is specified in the system of equations.

We want to insert these three lines of code into our working environment and evolve the system from its initial conditions. PyNSol requires the programmer to enclose the code in a multiline string and pass it to a constructor. To obtain this system's simulation, we must choose a grid scheme, apply space-differencing schemes to the equations, apply a time-differencing scheme to the result, and initialize the system. In PyNSol, this specification is a straightforward process, leaving the time loop clearly exposed to the programmer:

```
from PyNSol import *

grid9x9 = torus(9,9)

init = {
    "u":0.,
    "v":0.,
    "h":"hinit.nc"
}

params = {"csq":0.2}
shallow = process("""
```

```

def x_codegen(self,code):
    N = self.name
    (XMAX,YMAX) = self.data.shape
    XM1 = XMAX-1
    YM1 = YMAX-1
    code.main += """
inc(ix_y,ix_x) = inc(ix_y,ix_x) + ($(ix_y,ix_x+1)-$(ix_y,ix_x-1))/2.
""" .replace("$",N)
code.setdefault("ix_x .eq. 1", "")
code.cond["ix_x .eq. 1"] = code.cond["ix_x .eq. 1"] + """
inc(ix_y,ix_x) = inc(ix_y,XMAX-1)"""
code.setdefault("ix_x .eq. XMAX", "")
code.cond["ix_x .eq. XMAX"] = code.cond["ix_x .eq. XMAX"] + """
inc(ix_y,ix_x) = inc(ix_y,2)"""
code.setdefault("ix_y .eq. 1", "")
code.cond["ix_x .eq. 1"] = code.cond["ix_x .eq. 1"] + """
inc(ix_y,ix_x) = inc(YMAX-1,ix_x)"""
code.setdefault("ix_x .eq. 1", "")
code.cond["ix_x .eq. 1"] = code.cond["ix_x .eq. 1"] + """
inc(ix_y,ix_x) = inc(YMAX-1,ix_x)"""

```

Figure 1. A code-generation method for a finite difference operator of a toroidal grid.

```

    u.t = - csq * h.x
    v.t = - csq * h.y
    h.t = - ( u.x + v.y )
    """ )

m = \
model( shallow, params, leapfrog, grid9x9, init )

for i in xrange(1000):
    m.integrate()

```

Although end users will focus their attention on elaborating and sublooping the `for` loop in this code, the magic occurs in the model object's instantiation. Among the various transformations that occur during instantiation, PyNSol generates the following code:

```

v_t = lambda p, : - csq * h['cur'].y()
h_t = lambda u, v, : - u['cur'].x() - \
v['cur'].y()
u_t = lambda p, : - csq * h['cur'].x()

def loopfn(self,dt):
    self.dtime += dt
    v['inc'] = v_t(p)
    h['inc'] = h_t(u,v)
    u['inc'] = u_t(h)

```

PyNSol then evaluates and dynamically binds this code to

the model object's `integrate` method.

Another step in the model instantiation process is to differentiate the prognostic fields (in this case, `u`, `v`, and `p`) from the process declaration, and then bind to objects inheriting from a grid object. These objects then inherit space-derivative function methods from a grid.

In the simplest case (a first-order-centered difference on a toroidal grid), the method, drawing on Python's `numpy` module, looks like this:

```

def x(S):
    shl=concatenate((self.data[:,1:],
                     zeros((self.shape[0],1)),1)
    shr=concatenate((zeros((self.shape[0],1)),
                     self.data[:, :-1]),1)
    d = shl - shr
    d[0,:] = d[-2,:]
    d[-1,:] = d[1,:]
    d[:,0] = d[:, -2]
    d[:, -1] = d[:, 1]
    return type(self)(None,expr/2.)

```

These objects are also assigned to `u`, `v`, and `p` in the global namespace, so they're now available for the user at the Python interactive prompt.

This object structure thus builds the executable syntactically at object instantiation time and passes the code to the language compiler and interpreter for runtime evaluation. The automatically generated code then invokes these methods.

```

do 1 ix_x = 2, XMAX-1
do 1 iy_y = 2, YMAX-1
  inc(ix_y,ix_x)=inc(ix_y,ix_x)+(u(ix_y,ix_x+1)-u(ix_y,ix_x-1))/2.
1 continue

do 2 ix_x = 1, XMAX
do 2 iy_y = 1, YMAX
if (ix_x .eq. 1) then
  inc(ix_y,ix_x) = inc(ix_y,XMAX-1)
endif
if (ix_x .eq. XMAX) then
  inc(ix_y,ix_x) = inc(ix_y,2)
endif
if (ix_y .eq. 1) then
  inc(ix_y,ix_x) = inc(YMAX-1,ix_x)
endif
if (ix_x .eq. YMAX) then
  inc(ix_y,ix_x) = inc(2,ix_x)
endif
2 continue

```

Figure 2. The resulting code generated by the method in Figure 1.

PyNSol, in a sense, understands the code well enough to generate new, more detailed code from it. Besides its obvious advantages, nothing constrains this output code to Python, and certainly nothing constrains it to Python alone. The intention—and PyNSol’s principal innovation—is to have it both ways. In addition to binding runtime methods for derivatives such as `u.x()`, for example, the architecture can just as easily bind an additional method, say, `u.x_codegen()`. Indeed, the next choice for PyNSol’s code-generation target will be a classical language that predates object-oriented constructs, Fortran 77. Fortran 77, though old-fashioned, is an ideal language for source-to-source translation tools—in fact, a mature source-translation tool for it already exists for translating algorithms on regular grids to parallel platforms using the message-passing interface (MPI). The US National Center for Atmospheric Research’s weather forecasting model, MM5, which many countries use in both operational meteorology and in research settings, deploys such a tool, called the Fortran Loop and Index Converter (FLIC).<sup>6</sup> FLIC can bring automatic parallelization to PyNSol’s target audience, which is very interested in parallel computing platforms on regular (logically rectangular) grids.

As an example of how the whole process works, we could have the code-generation routine shown in Figure 1 for the same toroidal grid method as I described earlier. Although Figure 1 looks dramatically different from the `x()` method, it achieves the same thing by assembling a text string that contains the Fortran 77-equivalent calculation (see Figure 2).

Admittedly, working in this fashion takes some getting

used to, but this approach is increasingly popular in commercial settings.<sup>7</sup> (The code in Figure 2 isn’t sufficient for PyNSol’s purposes because we still need a second automatic transformation to support various time-differencing schemes. I use it here to give the flavor of the transformations that the PyNSol architecture can support.)

At first glance, this may appear silly. Surely it would be easier to write Fortran directly than to write the code that generates it. Although true, this is beside the point: the generator code only needs to be written once, and it can be reused in many contexts. Even in the simple shallow-water example alone, we would evoke this code-generation method three times. Automatic code generation is valuable for repetitive code fragments, not for cases that come up once.

## Architecture

The differences between using PyNSol’s approach and those of various other domain-specific languages that address partial differential equations is an interesting topic, but they’re beyond the scope of this article. Rather, I focus on how useful work is done with an unmistakably object-oriented strategy, but in which no objects whatsoever exist at runtime.

Figure 3 is a UML class diagram that represents the PyNSol structure. A PyNSol implementation consists of one or more models, each of which represents the equations of evolution of a physical continuum, and each model instance consists of one or more processes. Enhancing the user’s ability to control the interweaving of the processes using multiple time steps and schemes is a principal goal of the archi-

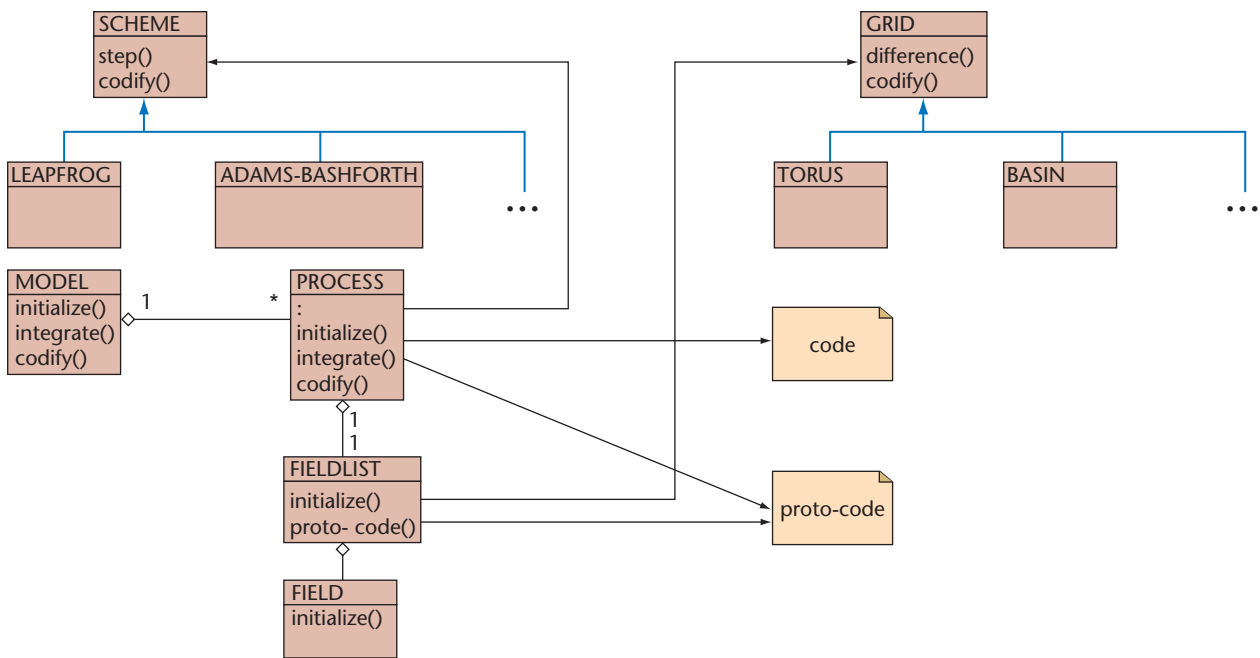


Figure 3. PyNSol's class hierarchy.

ture. Processes in turn have a `fieldlist`, the latter being composed of several objects of class `field`.

The abstract `scheme` class (which specifies time-differencing) provides an interface, but no implementation. Specific time-stepping schemes must inherit from it to implement appropriate methods and structures. So far, PyNSol supports only explicit time-stepping schemes, and, although implementing implicit schemes will present a challenge, this isn't an especially risky or speculative extension.

Similarly, the `grid` class is abstract. Although the `scheme` class abstracts the time-differencing scheme, the `grid` class abstracts both the spatial discretization of the field and the space-differencing schemes (corresponding to various spatial derivatives). The torus case is implemented as a method that dynamically creates torus classes of given dimensions.

The system state components that represent the model continuum's physical properties are instances of a `field` class. Transformation rules are associated with fields and allow them to be updated. The actual data is stored in an object of a class that inherits from a generalized abstract `grid` class. The non-abstract inheriting class is responsible for the actual implementation of the system's spatial numerical methods.

Instantiating a PyNSol process requires a choice of specific `scheme` and `grid` classes, a definition of the process equations, and an initialization of the fields. PyNSol generates any interactively callable integration routines as Python code at this instantiation time. It then evaluates and attaches this code as a method to the process. (A model consists of one or more such processes, woven together via explicit user code.)

A fully instantiated PyNSol system has two modes of operation. In *interactive mode*, the Python interactive interpreter invokes the generated method directly from a Python script or from the prompt, letting the scientist interact with the code. In *production mode*, the process's `codify` method is invoked, starting a process similar to the one that generated the interactive method for a target compiled language. This code is saved to a source file, which can be built and run using conventional methods.

Although we could, in principle, achieve source-to-source translation to a parallel architecture entirely in Python, we plan to implement the first version of this feature offline via FLIC. We'll use Python as a conventional glue language and simply invoke the operating system to perform the final code transformation. This planned use of FLIC drove our choice of Fortran 77 as the target language.

### Extending PyNSol

PyNSol's design disentangles the differencing and optimization methods from the physical model as far as the end user is concerned. It's the software's job to reweave these formal expressions into an executable. The code thus assembled isn't necessarily built in an object-oriented language, but the re-assembly itself, achieved via a series of semantic transformations, is implemented with an object-oriented strategy.

PyNSol's architectural design begins with a set of abstract classes for the model specification's required components. Implementation classes inherit from these abstract classes and then instantiate instance objects that implement the required transformation strategies. The sequence of transfor-

mations turns an expressive formal specification language (built as an extension to Python) into not only an interpretively evaluated and interactively available method, but also into compilable and automatically parallelizable code.

PyNSol has essentially three different use cases. Every user is in some sense a programmer, so let's call them end users, numerical analysts, and architects.

The system's end users aren't expected to be highly trained software professionals, thus they don't need to know about object hierarchy, abstract classes, code generation, or even very much about numerical methods. The objective here is to keep them focused on the area in which they have expertise, such as environmental physics or chemistry. A modest grasp of Python—the sort picked up in a week—is all an application scientist needs to use the system to support meaningful research that invokes well-tuned high-performance multiprocessor simulations.

Although PyNSol is an interactive, domain-specific language to the end user, it presents itself as an extensible framework to the numerical analyst.<sup>8</sup> In this framework, the numerical analyst must implement time- and space-differencing schemes; such analysts, who tend to be more versed in software methodologies, will also need an understanding of the interfaces that the grid and scheme objects support. They should have a modest familiarity with the target compiled language.

The third class of user, the system architect, must understand how all these pieces play together. A deep familiarity with object methodologies—as well as an understanding of the sorts of problem domains to which PyNSol can usefully be applied—is essential here. An example of an important architectural improvement for PyNSol's practical utility is the ability to support implicit time-differencing schemes, which could necessitate an intermediate XML-based language and some interesting algebraic transformations on top of it. Other useful modules could feasibly include a visualization module and operator overloading modules for automatic tangent and adjoint model generation. The system architect is responsible for the growth and utility of PyNSol itself. If the architect is successful, many of the details that normally interfere with scientific modeling of continua will be hidden from other users, whose interests are typically scientific or numerical.

### Why the Object Advantage Fails Us

The object advantage is in logically decoupled abstractions. An object-oriented system's decoupled pieces are on the winning side of a combinatorial explosion: a tightly coupled system is dramatically harder to design and maintain than

one that we can effectively break down into conceptually simple parts.

In many types of physical models, though, it isn't obvious that the real world will cooperate with our desire for the sort of decoupling that object techniques are meant to facilitate. In these cases, a conventional effort to divide and conquer the executable with runtime objects yields little—if any—advantage.

The patterns literature often states that new object-oriented programmers take the concept of “object as noun, method as verb” somewhat too literally.<sup>9</sup> These newbies have a tendency to identify the system's “objects” as physical objects. Scientific programmers make an interesting variation on this mistake: no atmospheric continuum modeler would have a “cloud” object, which is probably the first “atmospheric object” that pops into the mind of a nonspecialist.

Instead, the scientist's idea of “objects” tends to correspond to the physical process in the simulation. The atmospheric scientist, for example, tries to design one runtime object for radiation, one for convection, one for boundary layer friction, and so on. This approach identifies as objects the pieces that the scientist is most interested in swapping in and out. The trouble is that nature doesn't cooperate by making these phenomena especially “object-y.” The ideal object takes responsibility for its own data, is passed a very brief message, modifies its data according to that message, and then returns brief messages to other objects. On the other hand, physical phenomena in complex environmental systems are very tightly coupled and unavoidably share a vast amount of state—in fact, the entire physical state of the continuum under study. The object abstraction doesn't fit very well.

That being the case, can any object-oriented abstractions be brought to bear to simplify the modeling of environmental physics and chemistry? Indeed, PyNSol demonstrates that there are.

### Objects Are for People, not for Computers

PyNSol uses an intensively object-oriented strategy to implement a model that has absolutely no object infrastructure at runtime. Is this really all that surprising, though? A piece of code must accomplish two things: communicate with the coder and instruct the machine. The purpose of object-oriented code is to clarify the structure for the system's human builder; the machine itself has no need or use for such clarification.

In many environments, codes interact with other codes that can't be specified at build time. In such cases, leaving the object interfaces in place at runtime makes perfect sense: the collaborating code simply has to be built to a mutually agreed

## Café Dubois

### Haney Frightens the Warden

Everyone has their own favorite cartoons. Two of mine were in *Mad Magazine* when I was young. One was an extended parody of Perry Mason: “The Day Perry Masonment Lost a Case.” Week after week on the popular TV show, Perry’s cross-examinations shattered poor District Attorney Hamilton Burger’s airtight cases, forcing one murderer after another to burst into tears and confess. In the *Mad Magazine* version, the DA tricks Perry himself into committing a murder at home plate in Yankee Stadium during a day game. Perry is on trial, looking as cocky as usual, until the DA announces he has 45,000 witnesses that he plans to call in alphabetical order. It’s only when the first witness is called that Perry grasps just how much trouble he’s in: “Call Aaron A. Aardvark.”

In my other favorite *Mad Magazine* cartoon, a one-pager shows a patient in bed with medical charts and such. The doctor hands the guy a sheet of paper, and he reels back in shock. The title says it all: “Great Moments in Medicine: Presenting the Bill.”

In this spirit, I wish I had a cartoon I could call “Great Moments in Scientific Programming,” but instead, let me



paint you a word picture of the greatest moment in the Scientific Programming department.

From pages 552 to 557 in the November/December 1996 issue of *Computers in Physics*, we published an article by Scott Haney about using expression templates in C++.

This was a good moment, for sure—it was clear for the first time that performance and abstract expression in C++ weren’t implacable enemies. The idea that C++ really could be used for scientific programs was coming of age. Yes, a good moment.

But the truly great moment occurred on 19 December 1996, when we received a letter addressed to us by a prison warden:

*US Department of Justice  
Federal Bureau of Prisons  
Federal Correctional Institution  
<location withheld>*

*Dear Sirs,*

*Your magazine entitled “Computers in Physics” was received at this institution addressed to <name withheld>. In accordance with Bureau of Prison Program Statement 5266.07, “Incoming Publications,” the warden may reject material sent to an inmate if it is determined detrimental to the security, good order, or discipline of the institution. I have determined that the November/December 1996 issue of this publication will not be delivered.*

*Specifically, pages 552 through 557 contain explicit programming language. It is felt publications such as this could*

upon interface. In typical scientific programming, though, experimental codes simply don’t encounter unexpected collaborating executables. Usually, the scientist has decided what pieces will interact, so new ones don’t just attach themselves to a run. In such cases, runtime objects achieve nothing; rather, these objects are for us, and whether they continue to exist as such at runtime is a matter of convenience.

So how do we discover useful design- and build-time objects? The answer is by modeling the process of model design itself, which is where the benefits of abstraction and encapsulation really lie. The process of continuum modeling is often a series of loosely coupled and highly specified transformations. Although the final result is very complex, the initial formulation and each of the individual transformations are much less so. This process simply cries out for an object-oriented decomposition.

If you find yourself doing a series of transformations from a terse mathematical representation to a verbose, fussy, and awk-

ward final implementation, consider applying object strategies not to the code, but to the process of creating that code. You might end up with a code-generation strategy. To decide whether the resulting generated code should be object-oriented, consider whether the sorts of flexibility and encapsulation provided by polymorphism and inheritance really capture the nature of the runtime problem. If your executable isn’t going to unexpectedly encounter other executables at runtime, and you’re using a code-generation strategy, it’s probably best to consider limiting your object-orientation to build-time.

PyNSol is far from maturity, but it has already delivered some very useful lessons. It demonstrates that in some applications—and, in particular, in continuum models of natural processes—it’s worthwhile to apply object methods to decouple the design process rather than the executable. The objects of primary interest in such a strategy are the steps of the model implementation process, not the objects of the modeled physical world.

be used to seriously jeopardize institution security. I have determined this publication would be detrimental to the good order and security of this institution.

The letter went on to inform us how to appeal this decision within 20 days. Editor Lewis Holmes sent me a copy with the notation, "This is a new one on me!" I laughed until I cried. Once I recovered, I showed the letter to Haney, whose laughter was completely uncontrollable.

All these years I've wondered what <name withheld> had done to land himself in the federal pen, why he read *Computers in Physics*, whether explicit programming language was bad, and, if so, what would inexplicit programming language look like? Why was only this one issue rejected? Did the warden hate C++ and want the inmates to stick to Fortran? Was <name withheld> a trustee who made love to the warden's wife and, while she slept afterwards, hacked into the warden's bedroom computer to try to program in a door-unlocking worm, failing only because he made a mistake in the order of initializing a parent class's attributes? A whole hidden novel I can never read. You might ask yourself how you already guessed that the prison was in Florida. I hope you're still a loyal reader, <name withheld>, and living in better circumstances. If you're still trying to break out, try a FILE object.

### Skype

For the past year, I've been playing bridge on the Internet. I avoided playing bridge after I started college as I saw the finest mathematicians of my generation flunk out by playing it 24/7 in the student union; besides, I learned Go during my postdoc career, and it quickly became my favorite game. But for someone who's interrupted a lot by family,

Go requires a bit too much consecutive private time. By chance, I tried playing bridge on AOL and fell in love.


My favorite hangout now is [www.okbridge.com](http://www.okbridge.com). OK-Bridge has a Windows client as well as a Java one, so you can use it on Linux or a Mac. Actually, someone told me you can Telnet in, but that's so 20th century I haven't tried it. The current membership rate is US\$99 a year, but they have a free tryout period. (For Windows only, [www.online.bridgebase.com](http://www.online.bridgebase.com) has a quite-busy free alternative; ditto several other smaller sites.) Bridge can be a seriously competitive game, but it's also a social one; people like to chat with each other. And here Internet bridge has an annoying problem: seeing your partner type "wdp" just doesn't give the same satisfaction as hearing, "Well done, partner." And of course, the well-known problems of being unable to correctly perceive emotions in email apply in spades: "What was that bid, anyway?"

On the other hand, distance is no barrier on the Internet. I routinely play with people in many countries. A Bulgarian friend plays in the morning, when it's my evening; a regular partner lives in Toronto, and my teacher in Southern California.

Recently, we've been augmenting our lessons with Skype ([www.skype.com](http://www.skype.com)), a free voice-over-IP telephone/chat client program. Although you can buy cheap minutes to call real phones, calling another computer is free. Add a Logitech USB headset or something similar, and you're set. The quality isn't perfect, but it's pretty good. Do use a headset—words tend to break up or echo if you just use an open mic/speaker set. As it happens, conference calls can connect four users. Perfect!

You can test your headset by calling the user "echo123." And echo123 is sooo hot. A robot, but I can tell she loves me. Pathetic computer geek is such an ugly phrase.

The natural world's physical processes are under no obligation to cooperate with our methodologies by being loosely coupled and modular.

Objects in PyNSol are build-time scaffolding. They do their work in helping us think. By the time the processor, which doesn't get overwhelmed by detail, executes the model, objects are nowhere to be seen. 

### References

1. E. Dijkstra, "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982, pp. 60–66; [http://cs-exhibitions.uni-klu.ac.at/contentsep\\_main.php](http://cs-exhibitions.uni-klu.ac.at/contentsep_main.php).
2. R. Van Engelen, L. Wolters, and G. Cats, "Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling," *IEEE Computational Science & Eng.*, vol. 4, no. 3, 1997, pp. 22–31.
3. C. Elliott, "Modeling Interactive 3D and Multimedia Animation with an Embedded Language," *Proc. 1st Usenix Conf. Domain-Specific Languages*, Usenix Assoc., 1997.
4. S. Thibault, R. Marlet, and C. Consel., "A Domain-Specific Language for

Video Device Drivers: From Design to Implementation," *Proc. 1st Usenix Conf. Domain-Specific Languages*, Usenix Assoc., 1997; [www.usenix.org/publications/library/proceedings/dsl97/technical.html](http://www.usenix.org/publications/library/proceedings/dsl97/technical.html).

5. M. Lutz, *Programming Python*, O'Reilly Press, 2001.
6. J. Michalakes, "FLIC: A Translator for Same-Source Parallel Implementation of Regular Grid Applications," tech. memo ANL/MCS-TM-223, Argonne Nat'l Laboratory, Division of Mathematics and Computer Science, Feb. 1997; <http://www-unix.mcs.anl.gov/~michalak/papers.html>.
7. J. Herrington, *Code Generation in Action*, Manning Publications, 2003.
8. J. Carey and B. Carlson, *Framework Process Patterns: Lessons Learned Developing Application Frameworks*, Addison-Wesley, 2002.
9. A. Shalloway and J. Trott, *Design Patterns Explained*, Addison-Wesley, 2005.

**Michael Tobis** is a postdoctoral research associate in the Department of Geophysical Sciences at the University of Chicago. His technical interests include software engineering and high-performance climate modeling. Tobis has a PhD in atmospheric and oceanic sciences from the University of Wisconsin, Madison. He is a member of the IEEE Computer Society and the IEEE Engineering Management Society. Contact him at [tobis@mailbag.com](mailto:tobis@mailbag.com).