

# PATTERNS IN SCIENTIFIC SOFTWARE: AN INTRODUCTION

*Scientific programmers have generally avoided object-oriented approaches because of their heavy computational overhead. But the benefits of using patterns for scientific problems can outweigh their costs.*

Patterns are a well-understood methodology for object-oriented software architecture, especially for business applications. But what relevance, if any, do they have to scientific software? If they are relevant, in what way?

These are important questions, because scientific programmers have often shied away from object-oriented languages and approaches. Object-oriented languages have significant computational overhead, and scientific software development focuses on efficient mathematical algorithms rather than on data structures. Put simply, the intent of much scientific computing is fast and accurate number crunching. These needs contrast strongly with those of business applications, which require databases, user interfaces, and representation of business documents and transactions through complex data structures. It is a distinction as old as Fortran and Cobol. And thus, although most business applications have migrated over the past two decades to C++ and Java, the bulk of scientific software is still written in C and Fortran.

This article introduces the concept of object-oriented software patterns and discusses how they can be applied to scientific software problems. After a brief explanation of what patterns are and why they can be relevant to scientific software, I'll explore the application of patterns to dynamic-systems simulation, such as molecular dynamics, and identify four design patterns that emerge in modeling such systems. To illustrate how to reuse a general pattern for a specific problem, I'll then apply one of the dynamic simulation patterns to the different problem of hydrodynamic chemistry tracers.

## What are patterns?

Software patterns are an extension of object-oriented methods of analysis and design. Just as object-oriented programming built on earlier software ideas by unifying data structures and functions, so patterns expand the scope further by seeking out sets of interrelated classes and objects. Patterns are object systems that recur again and again in software architecture and implementation. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides introduced patterns to the computer software world in *Design Patterns: Elements of Reusable Object-Oriented Software and Design*.<sup>1</sup>

What constitutes a pattern is not always

**Table 1. Three categories of design patterns.**

Pattern type	Type features	Examples
Creational	Describes different ways of constructing objects	Object factories
Structural	Describes relations of classes and objects in a software system	Object compositions, interfaces
Behavioral	Describes how objects behave and interact; dynamics	Events, messages, iterators

sharply bounded. On the one hand, patterns are not primitive data structures like linked lists or hash tables, nor are they mathematical algorithms such as differential equation solvers or matrix inverters. On the other, neither are patterns application-specific architectures. Patterns are more abstract than that: They are *standard models* of a structure or process that can be applied to specific cases in a consistent way. Patterns thus enable the reuse not only of code, but also of the results of one object analysis or design in other contexts. Many of Gamma's design patterns, for example, arose from solutions in building graphical user interfaces.

*Design Patterns* identified 23 different class and object patterns, divided into *creational*, *structural*, and *behavioral* types (see Table 1). We can also classify patterns based on whether they pertain to *analysis* or *design*. Design patterns crystallize standard ways of implementing a software system; analysis patterns are conceptual models of elements of the actual systems the software represents. Martin Fowler's book *Analysis Patterns* presents several patterns directly applicable to scientific software; these include measurements, quantities with units, observation processes, and hypotheses.<sup>2</sup> In general, strong similarities exist between analysis patterns and older approaches to conceptual system modeling.<sup>3,4</sup> Conversely, any entity that often recurs in conceptual models of a physical system makes a good candidate to abstract as an analysis pattern.

Patterns, as systems of classes, are often communicated using the Unified Modeling Language. UML provides a convenient graphical representation of class attributes and methods, along with class inheritance, containment, and interaction relations.<sup>5,6</sup>

### Patterns in scientific software

The relevance of patterns to scientific software grows with system size and the corresponding needs for software to be more reliable, scalable, extensible, reusable, and maintainable. Patterns help lead to better code architecture by providing standard and proven structures. Ideally, a

software system would be constructed entirely of object components. Patterns achieve this goal by defining functional interfaces, by abstraction, and above all by decoupling a system's objects. (Component systems such as Sun's Enterprise Java Beans and Microsoft's COM are widely used in enterprise business computing. Remote object standards such as Java RMI, DCOM, and Corba provide means of decoupling objects at a performance price; these advantages and disadvantages are even more marked for emerging technologies such as the XML Simple Object Access Protocol, SOAP. Any scientific component scheme is up against the trade-off between flexibility and performance.)

Object decoupling not only leads to flexibility, but is essential for componentization. Decoupling, isolation, delegation, and abstraction are common threads that unite nearly all design patterns, whether through (creational) object factories, (structural) adapters, or (behavioral) messaging schemes. Libraries of mathematical functions are an old scientific software approach to meeting this need. Object-oriented software patterns go beyond function libraries, however, by building objects to represent entire physical systems, whether in an abstract form (such as a grid) or a more concrete form (such as an atmospheric grid for weather modeling).

Patterns also solve another common problem in scientific software development: division of labor. Scientific software is usually written by the "domain expert"—the physicist, chemist, or engineer who has the specific knowledge needed for the model, simulation, or algorithms. Although often inevitable, this is not desirable for large systems. Correct algorithms do not guarantee good software architecture or good coding practice. Conversely, writing good scientific code is difficult for a software engineer who lacks knowledge of the scientific principles the program should embody. Patterns and object-oriented analysis overcome this dilemma by introducing a division of labor between domain experts and programmers. They let the domain expert express information about the domain in a programming language-independent form, such

as UML. Software developers with little or no domain expertise can then implement the design.

All these benefits come at a price in performance, of course. The key to using patterns in science and engineering software is a judicious balancing of ideal object designs with computational speed.

### Identifying scientific software patterns

Pattern methodology use has two facets. First, we can apply or adapt an existing pattern. Second, we can identify new patterns—new class or object systems—that are specific to scientific computing problems. Here, I'll emphasize identifying new patterns. I do not, of course, provide an exhaustive catalog of scientific software patterns, but a few representative examples showing pattern methods and their utility.

The point of departure of nearly all scientific software is the need to model or simulate a physical system consisting of one or more entities, each having a state vector of relevant properties. The entities have relations—for example, fields—that govern either the system's time evolution and the corresponding change of state vectors (for dynamic systems), or the iterative convergence of the state vectors to a desired answer (for static solutions).

#### Patterns in dynamic systems

Let's focus on dynamic-systems simulation. The most important distinction here for patterns is between discrete and continuum systems. Discrete systems have a conceptual model of discrete objects with some set of relations that provides an equation of state or motion. Molecular dynamics simulations are the most obvious example, but scientists apply many of the same approaches used to model the dynamics of interacting atoms to celestial mechanics and the many-body problem in general.<sup>7-9</sup> Essentially, all these reduce to a conceptual model of mass points with locations and velocities, whose dynamics are governed by some combination of force fields (electromagnetic, gravitational, Van der Waals forces, and so on)—in other words, *particle mechanics*. Molecular dynamics thus yields our first relevant analysis pattern.

Continuum systems, such as fluids and fields, as well as quantum wave packets and extended elastic objects, can be modeled (except in very simple cases) only by approximation on a lattice of points or grid of cells. In other words, to

model the continuum, we must reduce it to a discrete form through a virtual set of objects and relations. Thus, the *grid* or *mesh pattern*, as we will call it, is a second analysis pattern.

Turning from these abstract analysis patterns to model implementation—that is, to possible design patterns—we see that a single conceptual model can in fact have different implementations, depending on the problem and how the model is used. For example, R.W. Hockney and J.W. Eastwood in *Computer Simulation Using Particles* pointed out that although a system of interacting particles can be modeled directly (the *particle-particle* method), it can also be modeled as an interaction between particles and a field grid.<sup>10</sup> This *particle-mesh* approach can be advantageous in cases where the force on any single particle varies smoothly and changes much more slowly than the particle motion. This can be the case when there are many particles, as in galactic and plasma simulations. The model calculates the force field on a grid at larger time intervals than the update cycle for individual particles, leading to a large gain in computational speed. We can combine the particle-particle and particle-mesh methods into a *hybrid* that splits forces into long-range, slowly varying forces on a grid with directly calculated short-range particle-particle forces. In other words, the conceptual model and analysis pattern of particle dynamics yields three distinct design patterns (see Table 2).

The one-to-many relation between conceptual models and design patterns also holds for continuum systems, because the PM and P<sup>3</sup>M patterns can be used in addition to grids.

#### Applying the particle-mesh pattern: Hydrodynamics

As an example of how to apply one of these patterns to a different problem, consider a specific case in hydrodynamics that models the fluid in a manner similar to the particle-mesh pattern. The problem is to calculate the dynamics of air with the chemistry of all molecular species included as part of the model. The model must account for trace species such as ozone or nitric oxide as well as ions; these are necessary in such cases as pollution models, prediction of atmospheric optical emissions, detailed models of lightning, and studies of chemistry in the highly disturbed atmosphere.

The straightforward approach to the problem is to calculate the hydrodynamics and chemistry together on a grid. However, if the model accounts for the 35+ chemical species that might

Table 2. Four dynamic simulation patterns.

Design pattern	Analysis pattern (conceptual model)	Description	Classes in pattern [and state vector]
Particle–particle (PP)	Particle mechanics	Point particles interacting through force relations (such as gravity and electromagnetism)	<u>Particle</u> [mass, position, velocity; other relevant force constants (for example, electric charge)], <u>Interaction</u> [force and equation of motion]
Particle–mesh (PM)	Particle or continuum mechanics	Point particles with a grid mesh containing the force fields; mesh values periodically updated from particle and external forces	<u>Particle</u> [same as for PP], <u>Mesh</u> [array of MeshElements], <u>MeshElement</u> [position; field value, such as electric field], <u>Interaction</u> [force and equation of motion]
Particle–particle–particle–mesh (P <sup>3</sup> M)	Particle or continuum mechanics	Hybrid of PP and PM patterns; forces derived both from field grid and particle interactions	Combination of PP and PM classes
Mesh or grid (M)	Continuum mechanics	Continuum discretized on spatial grid of cells or lattice: fluid mechanics and so on	<u>Grid</u> [array of GridElements], <u>GridElement</u> [position; other]

be relevant, the calculation will be prohibitively expensive in computer resources for a grid of any realistic spatial extent and resolution. To make matters worse, much of the chemistry can change on a faster timescale than the hydrodynamics, which dramatically slows down the calculation.

If all the air chemistry were strongly coupled with the hydrodynamics (as in combustion), we would have a difficult, perhaps insurmountable, problem. Fortunately, however, such strong coupling is not typical. Depending on the temperature and density, only a few of the atomic and molecular species contribute significantly to the hydrodynamics.

It makes sense, then, to split the problem into two pieces: a gridded hydrodynamic model with little or no chemistry and a detailed chemistry portion modeled as discrete tracer particles carried passively along by the hydrodynamic flow field. Figure 1 shows the hydrodynamic grid and chemical tracer approach. Each tracer particle represents an air parcel with its full number of chemical species. In essence, the problem divides into two different systems with two different sets of state vectors, the hydrodynamic system providing a set of external conditions for the chemical tracers.

The calculations are much more efficient, because the simulation of air motion need know nothing of the detailed chemistry and can usually be carried out for much larger time steps. We can, in fact, perform it completely separately from the detailed chemistry tracer calculations. Given the hydrodynamic data—the grid of densities, temperatures, and velocities at a series of

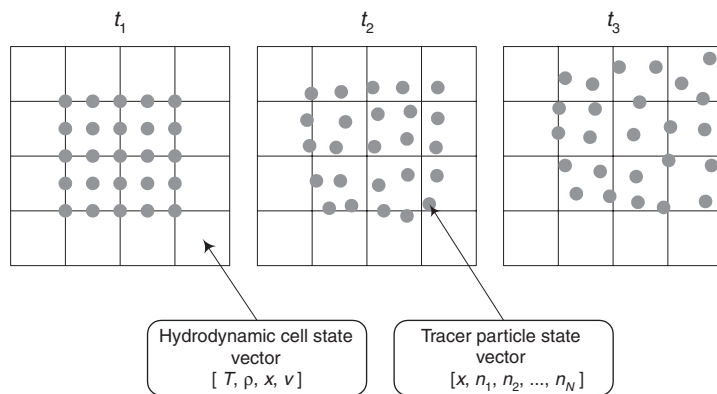
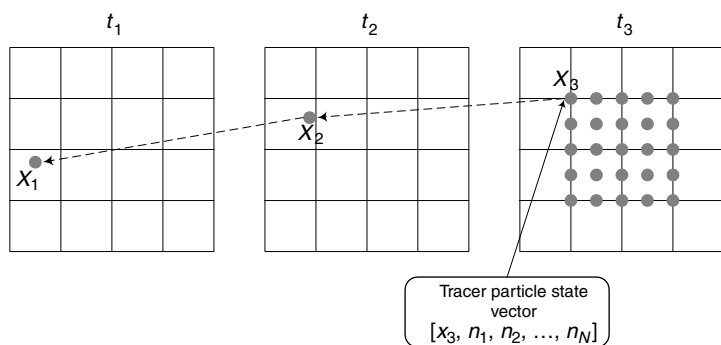


Figure 1. Hydrodynamic grid and chemical tracer particles at a series of times. In the state vectors,  $x$  is position,  $v$  is velocity,  $T$  is temperature,  $\rho$  is mass density, and the  $n_i$  are number densities of the chemical species of interest.

times—calculating the chemistry for the tracers is straightforward. Each tracer is instantiated at a certain location in the grid, and from this location it gets its initial conditions, both hydrodynamic and chemical. Integrating the tracers forward in time is then easy, because the velocity field passively carries them along. Also, at each time step, we obtain the density and temperature from the hydrodynamic grid. The model can then calculate the chemistry on the basis of stepping forward from the initial conditions, undergoing the changes of temperature and density. We can perform the detailed chemistry for each tracer with much finer time steps than the air motion grid.



**Figure 2. Reverse integration of tracer motion. Points of interest are integrated back to their original locations; the chemistry is then integrated forward based on the initial conditions and changes of density and temperature.**

The grid-tracer method is actually nothing more than an application of Hockney and Eastwood’s particle–mesh pattern. Thus, although this pattern’s original purpose was to deal efficiently with interacting particle simulations, it also applies to problems in the mechanics of continua. In addition, we can use several different design patterns to realize the conceptual model of a continuum hydrodynamic system—I chose the particle–mesh pattern here to address the needs of air chemistry.

Now let’s consider a variation on the grid-tracer theme, one that shows patterns’ utility more fully. For some problems, “forward integration” of tracers is not adequate. First, the air chemistry model must often inject many tracers into the grid to provide adequate resolution of results at later times. Even if we inject many tracers in a regular pattern at the initial time, they typically scatter to an irregular pattern. This means that we end up performing chemical calculations for a significant number of tracers that are never used. In addition, to get chemical properties at any particular point, the model must interpolate between several tracer values and search for the nearest tracers to interpolate from. The interpolation smooths the results and thus “smears out” any fine spatial structure in the chemistry, such as the results of shock waves or thin layers. Such fine structure of chemical species and temperatures can, however, be highly important—as it is, for example, in calculating atmospheric infrared emissions.

In such cases, a preferable method is to start with a set of locations of interest (such as a grid

of contour points at a particular time) and then perform a *reverse* integration to the initial tracer locations.<sup>11</sup> This approach, as Figure 2 shows, breaks the tracer calculation into two parts: a reverse integration of hydrodynamic properties that determines the path, density, and temperature for each hydrodynamic time step, and a subsequent forward integration of the detailed chemistry. Thus, we perform calculations only for the points of interest and avoid erroneously mixing chemical species between air parcels through interpolation of tracer values.

Thus, we now have two grid-tracer methods, forward and reverse integration; both are applications of the particle–mesh pattern. The only significant difference between them is the integration scheme, so unifying the two methods in a single pattern that has the following advantages is relatively easy:

- *Flexibility and modularity.* An initial version of the reverse integration model was implemented using traditional procedural coding practices. The problem was that, for each change in grid size, tracer patterns, chemical species and rates, and so on, the code had to be recompiled. In other words, the system was too tightly coupled and inflexible. Not merely a pattern approach, but any object approach, remedies this problem.
- *Incorporation of both integration schemes through abstraction.* Using an abstract pattern like grid-tracer (particle–mesh) lets a single model structure use several integration schemes by isolating them behind a standard interface. The actual method used can be chosen at runtime. The model at the high level need not know the details of the integration of either the equations of tracer motion, the hydrodynamics, or the tracer chemistry.
- *Object-oriented methods at low performance cost.* An object-oriented pattern approach would not necessarily render the system as a set of object components (grid cell objects, tracer objects, and so on), because this would be too computationally expensive. Rather, it would use the basic particle–mesh pattern as a template for an object factory. The object factory would then incorporate the specific parameters of each run of interest to generate the grids and tracers automatically. If still more computational speed were required, the object factory could write and compile source code based on the grid-tracer pattern for each case. Automated generation of code based on an ab-

stract pattern is a promising area for pattern use in scientific computing, one that uses the advantages of patterns without their performance drawbacks.

The grid-tracer approach, as an adaptation of the basic particle-mesh pattern, also shows how we can apply patterns to specific cases and reuse them. In the particle-mesh pattern presented by Hockney and Eastwood, the spatial mesh represented a field, which they used to derive the force acting on each particle in the simulation. Each particle had a state vector with mass, location, and velocity, plus other relevant properties such as electric charge. The field mesh was periodically updated from the particle properties (as well as any external fields). The grid-tracer method resembles the particle-mesh pattern on the abstract level. The differences are the particle state vector (location and chemical species), grid properties (density, temperature, and fluid velocity), and the integration method or equation of motion (tracers are passively carried along by a precalculated hydrodynamic flow field). There is also no “feedback” of the particle state into the field grid, as there can be in applications of the particle-mesh pattern. The implications of these differences are twofold:

1. To encompass the grid-tracer pattern on the abstract level, the basic particle-mesh pattern must isolate the particle state vector (other than location), the grid properties, and the integration of the equations of motion as separate objects. The pattern only specifies an interface for these, not the contents.
2. Applying the pattern means supplying implementations of the particle state vector, grid properties, and equations of motion with the integration method.

The particle-mesh pattern is not specific to any computer language. It could be implemented in C++, Java, or any other object-oriented language. Indeed, the particle-mesh pattern was first coded in Fortran.<sup>10</sup>

pattern. By grasping common systems of objects or classes at the abstract level, we can obtain a pattern that both better organizes existing software and can help solve entirely new problems. ■

## References

1. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
2. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, Mass., 1997, Ch. 3.
3. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Upper Saddle River, N.J., 1997, p. 903ff.
4. G. Booch, *Object-Oriented Analysis and Design*, Addison-Wesley, Reading, Mass., 1998, pp. 12–25.
5. S.S. Alhir, *UML in a Nutshell*, O'Reilly, Sebastopol, Calif., 1998.
6. M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, Reading, Mass., 1997.
7. J.M. Thijsen, *Computational Physics*, Cambridge Univ. Press, New York, 1999, pp. 175–193.
8. N.J. Giordano, *Computational Physics*, Prentice-Hall, Upper Saddle River, 1997, pp. 81ff and 233ff.
9. D.W. Heermann, *Computer Simulation Methods in Theoretical Physics*, Springer-Verlag, New York, 1990.
10. R.W. Hockney and J.W. Eastwood, *Computer Simulation Using Particles*, Adam Hilger, Philadelphia, 1998, pp. 6–23.
11. C. Blilie, *The Phenomenology Reverse Integration Method*, tech. report PRI-SB-89-R011, Physical Research, Inc., Santa Barbara, Calif., 1989.

**Charles Blilie** is a software architect with Computer Data Strategies, Inc. He has a PhD in physics and has done research in such areas as nuclear structure modeling, air chemistry, and infrared sensors. In addition, he has extensive experience in designing and developing software systems for application integration using Java, C++, and Python, designing XML messages for electronic commerce, and using methods of object-oriented analysis. Contact him at [cbllie@cds-inc.com](mailto:cbllie@cds-inc.com).

**T**he process of identifying and applying new patterns in scientific computing is analogous to what we've done here with the particle-mesh

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.