Scientific Programming 20 (2012) 349–353
DOI 10.3233/SPR-2012-0348
IOS Press

Book Review

Dan Nagle
E-mail: danlnagle@me.com

Scientific Software Design: The Object-Oriented Way, by Damian Rouson, Jim Xia and Xiaofeng Xu (authors), Cambridge University Press, 2011, ISBN 978-0-521-88813-4, 404 pp., hardcover.

1. Who are we, anyway?

What is computational science? We have all heard the cheerleading. Computational science is the third great leg of science, to take its rightful place alongside observation and experiment, on the one hand, and theory, on the other. But what has computational science really got that makes it a third way? Is it not just applied theory, computing the consequences of an equation? Is it not simply mining observations or controlling experiments and recording the results? Is there really a body of knowledge, distinct from any field of application, that stands alone? Describing such a body of knowledge is where the book described in the following stands.

2. A body of craft

Scientific Software Design, subtitled *The Object-Oriented Way*, is 382 pages and includes a Preface, 12 chapters (in three sections), two appendices, a Bibliography and an Index. The authors are from Sandia National Laboratories, IBM Canada Labs, and General Motors, respectively. As the title indicates, this book is about software design, particularly the design of scientific software. The examples involve solving differential equations (both ODEs and PDEs). Computational Science departments, where they exist as separate entities, or where they are lumped together with other sciences in a department of this, that, and computational science, must work to explain the differences between themselves and the established Computer Science department (whether lumped together with this or that engineering). Perhaps this contributes to an emphasis on managing large datasets (for example, the output of a large telescope or particle collider), or solving differential equations on complex grids (for example, fluid flow in ever-more realistic settings), rather than on old fashioned software engineering. These authors set themselves the task of addressing this perceived gap.

We sally into the Preface. With the laws of physics in our current universe being what they are, high performance computing means highly-scalable computing. These authors mean to lead us on a track which, they claim, is nearly orthogonal. They want to discuss highly-scalable design. What is that? With more and more science being multi-science, advances are happening along the boundaries among various disciplines. So several experts are involved. As the code grows, it will become more complex, with no one author able to ride herd on all of it. One consequence is that the design must support clearly understood interfaces so interoperability will be effective in the long haul. The social side of this situation is that every expert in one topic is a non-expert in other topics, who nevertheless must be able to understand the overall computational scheme, even if only to integrate each expert's own contribution into the mix. And with science being done in the interdisciplinary gaps, mix it will be. We also understand that greater mix means computationally greater size and complexity. Not only must the scientists work together effectively, so must their code contributions. The contributions of the various disciplines are being combined, not displaced. From this perspective, we approach Part 1, *The Tao of Scientific OOP*.

Chapter 1, Development Costs and Complexity, starts us with the notions of multiscale and multiphysics. The authors deplore their perceived lack of discussion of software architecture in today's computational science education. They illustrate with an example. Start with the heat equation, a model of physical reality. From here, one derives a oft-stated problem of the heat flow along a conductor, in this example case the heat sink on a processor chip. Next, a discretization scheme is applied to produce equations suitable

for digital computation. And code may be developed to be executed on a digital computer to solve the original problem. Any student in a computational science program could go this far, perhaps as a homework problem. But, the authors remind us, the programmer's time is more valuable than the computer's time. A simple example with a postdoc and a departmental cluster puts realistic numbers on the money involved to prove the point. So the discussion takes us to code complexity to show us that, while the numerical complexity of the solution scheme can be estimated, the complexity of the code maintenance and code development, all too often, is not considered. In short, how many lines of code must be checked to find a bug? This is the main expense of fixing the bug. A simple solution of our heat problem provides the example. Language constructs, such as modules, pure functions, and argument intents, are helpful by limiting the lines of code that must be checked for any search. Here, the authors' penchant for mixing metaphors comes to the fore, Ravioli Tastes Better Than Spaghetti Code. Momentarily, we divert to a history of computer science. We start with assembly code, with its unbounded jumps. Then, fifty years ago, Fortran put some bounds on things, with defined loops and choice blocks (if-blocks did not arrive until Fortran 77, but that is a nitpick – loops were there from the beginning and anyway the argument is sound). Veering past whether jumps should be considered harmful, we sightsee along the structured programming vista before arriving at object-oriented programming. While keeping code to small code segments has not been shown to reduce the overall count of bugs, it does nevertheless reduce the amount of code needing to be checked when the symptoms of a bug are seen. The 90–10% rule, which the authors call the Pareto Principle and quote as 80–20%, is next. Thus, we may write most of our code to be readable by ourselves, as the bottleneck is a small portion of the overall software. And the kernel is likely supported by a tuned library procedure, so we will not ourselves write even all of the kernel. And writing modular, object-oriented code will help us parallelize our code when time comes to care about performance. The authors finish this first chapter with a presentation of their coding style, and show us that Fortran and C++, as done here anyway, appear rather similar.

So oriented, we venture into Chapter 2, The Object Oriented Way. We start with a Rosetta Stone, where names of concepts taken from Unified Modeling Language (UML) are related to their near-synonyms in C++ and Fortran. So the UML Abstract Data Type (ADT) is related to the C++ class and the Fortran derived type. It is a long list. Feeling bound to begin with "Hello, world!" we find a structured program of five lines that does the trick. It appears on the same page an object-oriented program of 33 lines to the same effect. What was done? Or rather, why? We have built a solid wall defending the data of each step of establishing data, actor and action. With this, modifications may be done with greater confidence. And the count of lines to check to find bugs can be substantially reduced. One could apply the techniques employed to wrap some old, yet trusted code to be used in a new program, without fear of contamination across the border. We take our object-oriented analysis, fortified with UML diagrams, and create an object-oriented design for our heat problem. We visit composition, aggregation and inheritance. Some advantages of inheritance are shown when we wish to modify our heat program to use more general heat distributions. That was easy! Which is the point. We encounter unit testing, a benefit of our data-isolating style. A short discussion of static inheritance versus dynamic inheritance brings us to a revisiting of the complexity analysis we saw earlier. And all this was done in Fortran, with public and private attributes, only clauses on use statements, and pure functions sporting intent-in arguments. The intermodule barrier holds, and can be seen to hold.

Of course, a scientific software project will grow, and this is what happens in Chapter 3, Scientific OOP. (Scientific OOP will be written SOOP, pronounced, I presume, soup.) We meet Abstract Data Type Calculus. Using C++ overloaded operators or Fortran user-defined operators, and C++ overloaded assignment operators or Fortran user-defined assignment, we can make our software resemble our blackboard notes. We explore forward Euler quadrature followed by a backward Euler quadrature. Once we have methods to compute a time derivative, either quadrature can be coded quickly, along with the confidence-giving test cases we now know we want. Object Oriented Analysis and Object Oriented Design are working. More integration schemes are easily added to our heat code. How easy is it? Well, how hard is it? And we are lead to design metrics. We meet the concepts of cohesion and coupling, followed by the notions of an ADT's afferent couplings and its efferent couplings. Hence, we visit complexity theory. Exact results are not available, but we make do with computed bounds. Next, we visit information theory, and ask how much must be known for two classes to interact? That is, how much must two programmers communicate to work together on the same program? All of which leads us to the Tao of SOOP.

Having mastered the basics, our journey continues into Part II, *SOOP to Nuts and Bolts* (yes, by now one has learned not to mind the mixed metaphors). So we find ourselves starting Chapter 4, Design Pattern Basics. Design patterns, we learn, originated in architecture. We take the lessons learned there to programming. We have a bit of a history lesson of how architectural ideas came into computer science. So following the path of architect Chris Alexander in his *A Pattern Language* books, and determined to use design patterns, we encounter the three problems that serve as examples in the remainder of Part II, the Lorenz Equations, whose chaos allows us to test our computed solutions, quantum vortices in superfluids described by the Biot–Savart law and Burgers equation. So prepared, we venture onwards.

The next chapter, Chapter 5, The Object Pattern, discusses the general object. It starts with a discussion of just how rigid to make the design of a large program. Which details to specify? If one is too lax, the code will lose understandability; if too tight, one might press needless constraints, hindering some efforts and wasting others. The object pattern takes its name from Java, where all entities are descendants of the object class. The object class holds codifications of properties we want all objects to have, such as construction and destruction. Our example for this chapter is the vortex problem described in the previous chapter. It is interesting due to the desire to have an object represent each vortex, which must be linked together to form the ring. Since the vortexes occur in a ring, and one would like a pointer from vortex to vortex, one has an interesting issue: how to add or remove vortexes? The data structure is not a list yet has a referent from the outside. Once one's code is traversing the ring, one has endless traversal. (Visualize a figure 6. Descending from the top along the riser, one is following an outside referent. Once in the loop at the bottom, one continues in endless cycles.) Pointers are used in either Fortran or C++, but the usual object constructor or destructor may not be used. Thus, one has a dangling pointer problem (where the object targeted by the pointer is no longer valid, yet the pointer remains). How to avoid the memory leak? The object pattern is used to supplement the language-supplied pointer to provide safe pointer manipulation. In either language, a more elegant solution could be employed, but the authors sacrifice for pedagogic purposes, keeping the solution simpler, and their point is clearly made.

Moving to Chapter 6, The Abstract Calculus Pattern, we see how to build our divisions between modules more clearly. The Lorenz problem, those three little scalar coupled differential equations demonstrating the attractor, will be our focus. This time, we write abstract classes to supply the integration services. Our user will write to the abstract classes. We then implement the abstract class with a Lorenz class containing the particulars. Our user does not even see our actual procedures, only their signatures. The solutions in either C++ or Fortran are strikingly similar. The biggest difference being that Fortran uses a default clause in a select type block where C++ attempts a dynamic cast within a try block, both languages guard to catch a class that cannot be integrated.

In Chapter 6, we supplied a service, integrating the Lorenz equations, without exposing the procedures that did so, we only exposed their signatures. Thus, we can change the procedure we supply during execution. That is what we do in Chapter 7, The Strategy and Surrogate Patterns. Now, we integrate the Lorenz equations, and we can choose an integrator from among a selection provided. As the design becomes more abstract, the C++ and Fortran expositions of the design are, surprisingly perhaps, converging. Here, the major difference is that the Fortran version must use an auxiliary class to implement the Surrogate Pattern, which is unneeded in C++ due to its forward attribute. On the other hand, the C++ version uses reference-counted pointers where Fortran can gain the same effect with its allocatable variables.

Pressing bravely into Chapter 8, The Puppeteer Pattern, we encounter the next level of complexity. Recall that we are addressing the complexity that arises from multi-physics simulations. Our weapon of choice is to safely encapsulate local data within the definitions of the software representing the abstractions involved. What happens when a non-linear problem requires data from within each abstraction to use an implicit method to advance to the next time step? We employ a Puppeteer, which acts the role of the broker who knows all the actors (considered to be the puppets), and allows them to cooperate without knowing each other. Again, the C++ implementation and the Fortran implementation are remarkably similar.

Chapter 9, Factory Patterns, takes the encapsulation process one step further. In the non-linear problem solved in Chapter 8, we had to allow objects of one class, the puppeteer class, to have access to data that we would like to be private within another class (the puppets). Can this be avoided? Indeed it can, and we learn that the factory pattern shows the way. Our example is solving Burgers equation. This time, all the classes in-

volved will be unknown to each other except via their parent classes. The parent classes expose the signatures of the procedures to be publicly known. The actual procedures are not known outside the child classes at all. Instances of a factory class can create new instances of the working classes as needed. Our separation techniques are functioning well. And we can advance to Part III, *Gumbo SOOP*.

Chapter 10, Formal Constraints introduces us to the Object Constraint Language (OCL). This language is aimed towards bridging the gap between working scientists, who are the subject matter experts of scientific computing, and experts in formal methods, the language whereby program correctness is to be expressed. Our example is a solution of Burgers equation expressed in Fortran. OCL lacks a good expression of arrays, so we must explore how to do so before proceeding to discuss memory management. Fortran specifies that the arguments of defined operators must not be changed by the pure procedure defining the operation, so if the results of nested operators are pointers, memory leaks can develop especially in complicated expressions with many partial results. This point, and Fortran's lack of a language-defined assertion mechanism, are the subject here. Solutions are provided, but the language could provide more help to good effect.

So we move into Chapter 11, Mixed-Language Programming. The context for the discussion is the Trilinos package. The goal is to link the Fortran binding with the C++ software. The strategy is to use Fortran's Interoperability with C to define the binding. Trilinos uses an object oriented design, so linking the of C++ to C to Fortran poses the issue that C does not directly support object oriented designs. Nevertheless, the authors describe a solution, which allows both the C++ side and the Fortran side to use object oriented features while communicating through C.

And so we arrive at the ultimate chapter, Chapter 12, Multiphysics Architectures. This brings us to the point where multiphysics meets multiprocessor. We profile our Burgers solver. Next, we dutifully salute Amdahl's Law one more time, this time with the greater detail of a multipart program. We review multithreading only to find refuge in directive-based OpenMP. Our next effort lands us in the war zone of message-passing, complete with conditional compilation, via *cpp*, to remove the library calls when undesired. We finally come to meet coarrays. Unlike library code, coarrays are fully integrated into the language, so they fit into derived types without jumping through hoops and fully participate in other common language features (allocation, input/output and so on). Our goal is to compute solutions to scientific problems, so we meet the multiphysics. Quantum turbulence, lattice-Boltzmann biofluid dynamics, particle dispersion in magnetohydrodynamics, and radar scattering in the atmospheric boundary layer serve as example problems. Each requires cooperation from a variety of disciplines. We arrive at the Morfeus Framework, an open-source project intended to aid our efforts at solving these and similarly complex calculations.

We have reached the Appendices. The first covers mathematics. The material would be at home in a first-year graduate course in Computational Science. The second introduces the reader to the Unified Modeling Language (UML). The authors make good use of the UML, the reader lacking familiarity with it could profitably read it before starting the chapters. Other readers might prefer to find a book on UML before venturing into the main text.

3. The skills of growth

So where have we been on this journey? The authors start us with the proposition that binding software into independent, self-contained and therefore reusable, segments is good for software development. Many professional programmers, making, for example, office software, would do little more than yawn in reaction. The authors have then proceeded to show us that, as the scientific complexity of our computational efforts grows, so the need for independence of scientific modules grows as well. This is due to the degree of detail to be included in our computations within one discipline, but also due to the interdisciplinary needs of a single analysis. The authors have also shown that, far from being at odds with the needs for parallel processing implied by the magnitude of operations counts and the size of datasets, properly handled independence of modules works synergistically with the forms of computation needed by parallel processors. This is indeed good and useful news. The techniques shown here can indeed form the inspiration for one's own design (or redesign). Indeed, there are several instances above where I wrote of what the programmer could or should do, when I could (should?) have written of the scientist instead. And that's the point: if computational science is a distinct discipline, the scientist must become a programmer.

New hardware today moves from commercial or entertainment uses towards scientific use. The "attack of

the killer micros" was first heard some 25 years ago. And today, 64-bit parallel hardware is developed for gamers and home viewing of movies. That's where the big money is, after all. Are software development techniques moving from commercial software developers to scientific software developers, too? The answer must be "yes" but certainly it's happening more slowly than one might like. Confidence in long-used and well-tested software, and a desire for continuity of published work conspire to keep the "if it ain't broke, don't fix it!" attitude alive and well. Computational Science is new to many campuses, so practitioners are still resolving just what the departmental identity is. Surely the focus must be on differential equations and large databases, with perhaps some graphics for understanding and for communication to colleagues. So perhaps issues of software engineering are still in the pending tray.

These authors have shown that increasing complexity, due to increased details of the calculation, increased variety of contributors to bring together an increased variety of expertise, and the increasingly complex demands of parallel processing for execution, must be addressed, and they have gone a long way at least towards showing us how to do so. Networked open source scientific software projects may further confuse things by leaving some inter-programmer communications in batch mode. Scientific software must be consciously designed to grow with a research program and the hardware that supports the research program. And how to do that is precisely what these authors in this book have shown.