# Development of a Weather Forecasting Code: A Case Study

**Richard Kendall, David Fisher, and Dale Henderson,** *Software Engineering Institute*

**Jeffrey C. Carver,** *Mississippi State University*

**Andrew Mark, Douglass Post, and Clifford E. Rhoades Jr.,** *US Department of Defense High Performance Computing Modernization Program*

**Susan Squires,** *Sun Microsystems*

A case study of a weather-forecasting code aimed to investigate the code development challenges, understand the development tools used, and document the findings for other developers.

Computational science is increasingly providing insight into scientific phenomena that have previously been studied only experimentally, observationally, or theoretically. Codes, the software projects written for computational science, generally have three main differences from traditional software projects, such as those from the commercial software domain. First, because these projects often perform new scientific investigations, the requirements can't be known in advance and must evolve over time. Second, the main driving force for these projects is producing correct, important scientific advances rather than ensuring software quality through formalized software engineering processes. Finally, these projects' developers tend to be domain scientists rather than software engineers, so they're less likely to use any heavy software development processes.[1–3]

To better understand the impact these characteristics have on software development practices in the computational-science domain and to document lessons learned for similar projects' benefit, we performed a series of case studies of computational-science code development projects sponsored by the DARPA High Productivity Computing Systems program. Here, we describe the sixth case study (after Falcon,[4] Hawk,[5] Condor,[6] Eagle,[7] and Nene[8]).

The common objectives for all case studies were

- identifying critical success factors,
- identifying issues that hardware and software vendors must address to make the code development process more productive,
- developing a reference body of case studies for the computational science and engineering community, and
- documenting the lessons learned from analysis and personal team interviews.

For this case study, we followed the same methodology that we used in the previous case studies.[2] Here, we provide only a basic overview of the methodology to give context for understanding the results. After we identified a suitable project, the team completed a questionnaire. We used the responses to plan an on-site interview, after which we iterated the results with the code team to ensure correctness.

At the highest level, the Osprey code team has some control over the requirements through the proposal process.

## Code characteristics

Osprey is one component of a large weather-forecasting suite that combines the interactions of large-scale atmospheric models with large-scale oceanographic models. Osprey's current code—which has evolved from other codes that date back to 1974—has been used operationally for 10 years. Development of the present version began in 1988 as a research project that aimed to extend the predecessor code's capabilities. At the highest level, the Osprey code team has some control over the requirements through the proposal process. That is, the team can make direct proposals to potential sponsors about features to implement in the code. This process is how the team acquired funding for code-directed goals such as the implementation of the Message Passing Interface (MPI) and nested grids. The sponsors, which have increased in number in recent years, often set the requirements through directed research and development grants (that is, unsolicited grants). Work tends to be funded in ever-smaller increments (relative to constant costs), and individual code developers now work in multiple topic areas rather than specializing in one topic.

Osprey is distributed via the Web. It has no license fee, and the team has deployed a home-grown license manager to track the code's distribution. Osprey has been downloaded hundreds of times by institutions worldwide. Some institutions have hundreds of users.

Because the Osprey code is one component in a larger system of systems, its architecture lets it both send and receive information from the other system components. A framework supports one-way and two-way coupling of Osprey to other system components, using an exchange grid to facilitate data exchanges between components.

With the exception of approximately 300 lines of C code, Osprey is written in a Fortran subset (which we describe in the next section). There are approximately 150 KLOC in Fortran, of which approximately 50 KLOC are comments. The driving motivation behind choosing Fortran was the need for portability and ease of development and maintenance (similar to our findings in the Falcon, Condor, and Nene case studies). After considering these needs, Osprey determined that Fortran was the best language available when development began. Although university computer science departments no longer routinely teach Fortran, new developers can master its fundamentals in a week (using Fortran 77) to a few months (Fortran 90). In contrast, C++ or other modern, higher-level languages typically take much longer to master. Furthermore, using an all-Fortran code eliminates complicated make and link operations and other problems associated with using multiple programming languages. The project leader hasn't encountered any limitations in Fortran (augmented with the MPI) that would cause him to seriously reconsider the choice. Moreover, programming-language evaluation has been a distraction in the past, so the team is reluctant to invest resources in this area unless it has identified a specific need, such as the one that led to implementing MPI.

Essentially one version of the Osprey program library executes on SGI, IBM, HP, and Linux platforms. A locally built preprocessor configures the code for each supported platform. Compiler flags configure the code to run using different parallel-programming libraries (such as MPI and OpenMP).

Parallelization is a key priority because the team must use parallel computation and processing to achieve the performance level needed to obtain the expected scientific results. Conversely, actual demonstrations of parallel performance haven't reached the level that the Osprey development team believes is feasible. This shortfall results from the fact that, to maintain portability, the code isn't highly tuned for any specific parallel platform; however, some optimization occurs for each platform. In general, however, the team avoids the use of libraries and machine-dependent features. Earlier attempts at hardware-based customization led to cluttered, unmanageable code. The emphasis now is on general, flexible, and scalable code.

## Code project and team

Osprey's core team consists of about 10 developers and a few outside consultants and postdoctoral researchers. The core team is located at a single site along with most of the development teams for the other components in the larger system of systems. However, other sites also contribute components. Most of the core team members have worked together for the past 10 years. The team leaders have preferred to recruit staff members who already have domain knowledge. Moreover, the team has generally found individuals with computer science backgrounds to be helpful for specific coding tasks or applications, but it hasn't utilized computer scientists as part of the primary code development team. The team has had difficulties interfacing with pure computer scientists that have had little exposure to large weather or oceanography codes. As an example of the difficulties the team has experienced working with computer scientists, the Osprey team leader cited a computer scientist who wanted to make every executable line of the code into a subroutine to make the code easier to debug.

The Osprey team describes itself as follows:

- Team members have a common professional background.
- Team management processes are largely informal and collegial.
- There is strong peer pressure to contribute and strong team member dedication to the project.
- The team views code validation as especially important.

The Osprey team indicated that code maintenance, portability, and speed-to-solution are the main drivers of development, with ease of maintenance being paramount. They also emphasized flexibility, by which they mean

- the code must run on many platforms,
- the code must port easily to the next generation of hardware, and
- the code can't be tailored to any particular platform.

This definition of flexibility represents a conscious attempt to manage an important technological risk. The Osprey developers aren't encouraged to use all of the constructs that Fortran supports. The practice within the Osprey development community is to use only features that are well tested and reliable. The team follows a style guide and uses the structure and coding practices of the code itself to train new team members.

Osprey and the developers of the other system-of-systems components are colocated and under a common management structure. This colocation, along with the Osprey development team's relatively small size, has limited the degree of formality needed to manage the team. The Osprey team doesn't adhere to any formal code development methodology (such as CMM, ISO, or even formal agile development processes). However, the team has adopted repeatable coding practices and employs coding-style requirements and standards. From the formal software engineering viewpoint, this project is underconstrained—as have been all but one of the other computational science projects that our team has examined in case studies. This approach is successful because, for the most part, the Osprey team can set its own milestones with the sponsors' approval; thus, the iron triangle (that is, the scope, resources, and schedule) isn't violated. Although the team doesn't identify with formal development processes, it follows an agile philosophical approach (www.agilealliance.com) in the sense that it emphasizes practices over process. Table 1 summarizes the extent to which the Osprey team uses various development practices.

Of the 20 or so software development tracking metrics[9] cited in the software engineering literature, the Osprey team employed

- lines of code,
- code performance,
- degree of performance optimization,
- parallel scaling,
- number of users, and
- computer time used for code development.

LOC is a general measure of the code's growth. There is growing pressure from sponsors for science metrics, not code development metrics.

Osprey's approach to configuration management is similar to that of the other projects we've studied. A part-time source code librarian uses Subversion (http://subversion.tigris.org) to manage the code. The code librarian logs the bugs. No formal issue-tracking system has been deployed, nor does one appear to be necessary. The code librarian requires a prologue detailing a subroutine's purpose before it can be committed to the code library. Verification tests are also required.

## Code life cycle and workflow management

Much of the Osprey program library's code is in maintenance. Because Osprey is undergoing continuous enhancement, there's always new development and testing. At any given time, code modules in the program library range from brand new to 20 years old. This project's workflow is typical of the other projects that we've studied, consisting of the following steps:

- question formulation,
- development approach formulation,
- code development,
- testing (validation and verification),
- production,
- analysis, and
- hypothesis formulation.

(This isn't customary software engineering language—such as requirements gathering or specifications formulation—but it's more descriptive of the actual workflow management process the computational-science projects we studied employed.)

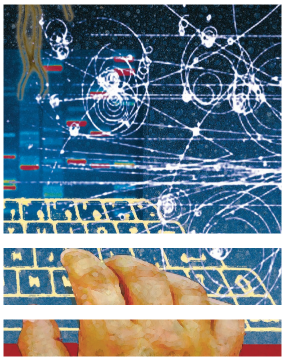


**Table 1**

**The Osprey team's development practices**

| Practice | Description | Followed | Comments |
|---|---|---|---|
| Collective ownership | Allow anyone to change any code anywhere in the system at any time. | Partially | Followed to a small degree; a limited set of developers allowed to make changes |
| Configuration management | Establish and maintain the integrity of work products using configuration identification and control. | Yes | A key role in this project |
| Continuous integration | Integrate and build the system many times a day, each time a task is completed. | Partially | Performed weekly rather than daily |
| Feature-driven development | Establish an overall architecture and feature list, then design by feature and build by feature. | Yes | Project development driven by new features |
| Frequent delivery/ small releases | Have many releases with short time spans; implement the highest-priority functions first. | No | Driven by task—no explicit goal to have small releases, although this often occurs |
| Onsite customer | Include a real, live user on the team who is available full time to answer questions. | Yes | Code's developers are also users |
| Organizational process definition | Follow an organization-wide process. | No | No formal organization-wide process |
| Organizational training | Develop team members' skills and knowledge so that they can perform their roles effectively. | No | Nothing formal in place |
| Pair programming | Work side by side with another programmer at one computer, collaborating on the design, algorithm, and code. | No | Not found to be useful |
| Planning game | Quickly determine the scope of the next release with business priorities and technical estimates. | No | Not used |
| Peer reviews | Review peers' software artifacts (requirements, design, code) to improve quality. | Partially | Informal code reviews (but not requirements or designs) used |
| Process and product quality assurance | Objectively evaluate adherence to process descriptions and resolve noncompliance. | Partially | Long-term roadmap for Osprey, but no formal monitoring of adherence to the development strategy |
| Project monitoring and control | Provide an understanding of the project's progress so that appropriate corrective actions can be taken if progress deviates from plan. | Yes | Internal and external reviews throughout the year; local groups meeting weekly to discuss issues |
| Project planning | Establish and maintain plans that define project activities. | Yes | Planning done at the beginning of each fiscal year as part of the funding cycle |
| Refactoring | Restructure software to remove duplication, improve communication, simplify, or add flexibility. | Yes | Used to improve performance and make it easier to change the code in the future |
| Requirements development | Produce, analyze, and verify customer, project, and product requirements. | Yes | A formal process followed as part of proposal generation |
| Requirements management | Manage the project's requirements and identify inconsistencies with the project plan. | Yes | Internal milestones set by each subteam |
| Retrospective | Perform a postiteration review of the effectiveness of the work performed, methods used, and estimates. | Yes | Performed at the annual review |
| Risk management | Identify potential problems and adequately handle them. | Partially | Emphasis on testing and benchmarking; conscious strategy for managing technology risks; no formal management of other risks |
| Simple design | Design only what is being developed, with little planning for the future. | Partially | There's a long-term plan beyond the current funding; the design is documented only in the code, so features that aren't coded haven't been formally designed |
| Tacit knowledge | Maintain and update project knowledge in participants' heads rather than in documents. | Yes | Tacit knowledge in the scientific discipline is important |
| Test-driven development | Write module or method tests before and during coding. | Partially | Testing integrated with development |

The development path through these steps is iterative. The team developed physics prototypes for testing in the Osprey code framework. The team wrote these prototypes as candidates for direct inclusion into the code, so there's also an aspect of spiral development. Unlike some other projects we've studied, these prototypes aren't written in a higher-level language such as Matlab. However, the Osprey team sometimes uses Matlab to study parameterization issues. There is some refactoring of old modules as capabilities are replaced or enhanced. The project has no mandated release schedule; releases occur when new capabilities or bug fixes are ready. Last year there were 10 releases, but some were small fixes. The downloadable version is two years old (that is, well behind the most current version).

As we've observed in our other case studies, most sponsors don't directly fund code maintenance. It's not entirely clear that development is funded directly either—the funding is usually directed toward the science, not the code.

The team does testing both during and at the end of development. Team members can't submit new capabilities for inclusion in the Osprey program library unless the capabilities pass verification tests. The developers do substantial testing before submitting capabilities to the configuration-management process. Team members share quality assurance. After internal testing, the code is transferred to a primary customer for additional, independent testing prior to release. For major changes, an independent panel with broad stakeholder representation performs a series of tests before the transition to production.

A goal of our case studies has been to document the tools the code development teams use (see Figure 1 for a summary of these for Osprey). The use of particular tools, such as TotalView, hasn't been mandated and is often a matter of personal preference. IDL is used primarily by customers, not developers. In general, developers prefer open source tools.

## Lessons learned

All our case studies have extracted important lessons for the computational science and engineering code development community. These lessons can guide software engineers seeking to better understand how to successfully work with this type of code project. In this case study, we present two types of lessons learned. First, we describe the lessons the Osprey team members learned. Then we combine those lessons with our observations from the questionnaire and interviews into the context of our previous studies' findings.

| | |
|---|---|
| **Code development environment** | |
| Compilers | Fortran, C |
| Scripts | Perl |
| Debuggers | DX, TotalView |
| Performance analysis: profilers | prof |
| **Execution environment** | |
| Element/grid generation | GrADS |
| Visualization | IDL, Matlab |
| Data analysis | IDL, NCAR command language |
| **Code development process tools** | |
| Configuration management | Subversion |
| Bug tracking | No formal tools deployed |
| Code documentation | Users' manual, code documentation, Web site |
| **Support libraries** | |
| Computational mathematics | BLAS |
| Parallel-programming libraries | MPI, MPI-2, OpenMP |

**Figure 1. Osprey's life-cycle management tools.**

First, the Osprey team members articulated the following lessons learned:

- If it's important, do it right.
- Listen to the customer.
- Data assimilation becomes increasingly important as the model grows in sophistication.
- Configuration management is essential.
- Physical parameterizations remain a necessary evil, but you should avoid them if physical models are feasible.
- Validation and verification are crucial to improving and accepting scientific codes.
- Strive for performance, but not at all costs.
- Build flexibility into the system.

In our previous case studies of computational science codes, we made nine observations about the software development environments used.[2] The lessons the Osprey team learned and our observations from the questionnaire and interviews provided support for seven of those earlier observations. Here, we describe them in more detail along with the supporting information from the case study.

### A code project's primary language is constant over the project's long lifetime

These projects tend to last multiple years (even decades), as evidenced by the fact that some of the Osprey project's code dates back 30 years. Therefore, you might expect that the programming language would evolve over time as more modern languages are developed and standardized. Conversely, in Osprey and the other codes we studied, the basic language didn't change over time. The

**The Osprey team still uses Fortran because it's easy to learn compared with modern languages.**

Osprey team still uses Fortran because it's easy to learn compared with modern languages. This is true also because the team has yet to encounter a weather-modeling requirement that couldn't be programmed efficiently, both in terms of programmer effort and execution time, using Fortran.

### Use of higher-level languages is low

The Osprey project uses high-level languages such as Matlab very sparingly, if at all. In some other cases, we've seen Matlab used as a prototyping language almost akin to a statement of requirements. But none of the projects we studied extensively used such languages.

### Risk management is important

These types of projects' long lives present risks that aren't always present in other types of software. In our previous studies, teams viewed externally developed tools and libraries as risky because of the likelihood of the vendor going out of business during the project's life cycle. For the Osprey project, lack of flexibility was an important risk. To address this risk, they designed and implemented the code so that it wasn't closely tied to any particular platform and could be easily ported to new platforms as they became available.

### Performance isn't the only important nonfunctional requirement

All the code projects we studied used high-performance supercomputers, so performance (that is, speed of execution) was an implied goal. In Osprey, performance is an important goal, but only to the extent that it improves the code's scientific output. In other words, scientific goals drive the need to achieve a particular performance level. Osprey's main focus is to support scientific research, not to demonstrate computer science ability or even achieve impressive performance numbers. Although it isn't the main driver, code performance is still important to the Osprey team because its customers require real-time applications that support operational functions. In addition, as discussed earlier, maintainability and portability are also essential to the Osprey code's success.

### Agile approaches are better suited than more traditional methodologies

In the projects we studied, including Osprey, scientific-code teams valued agility over formalized processes. That is, they usually avoided rigid software management approaches, but planning and adoption of useful software practices *are* important to the success of these projects. We've found that development teams can function well with a very lightweight process as long as teams are small, perform adequate planning, and have good communication among team members. Although Osprey and the other code teams we studied didn't use one of the formal agile methodologies (such as Scrum or XP), they did operate their team according to the mind-set proposed in the Agile Manifesto (http://agilemanifesto.org). The choice to operate this way emerged out of necessity rather than a conscious decision to follow one specific agile approach over another.

### Multidisciplinary teams are important

The complex nature of the scientific domains for which computational science is effective necessitate that much of the code be written by domain experts (scientists). It's simply too difficult to teach a computer scientist or software engineer the technical details of a scientific domain for which a PhD is needed to even understand the problem to be solved. In many cases, including Osprey, these teams have found it effective to use computer scientists' expertise and ability to perform specific coding tasks that don't require extensive domain knowledge.

### Project success or failure depends on keeping customers and sponsors happy

As in any type of project, its longevity depends on its success among its customers. One unique aspect of the computational science projects we studied, including Osprey, is that the customer and the users aren't always the same group of people. So, to be successful, a project must keep both the users (other scientists) and the customers (the sponsors or funding agencies) satisfied. If the code project meets the technical goals the customers set, but isn't seen as useful by the end users, then it won't be viewed as a success over the long term.

We hope that these lessons learned will be useful to other scientific-code developers. We realize that the Osprey project might be unique and might not represent experiences in other domains. Nonetheless, we believe that many of these lessons are applicable to a broad range of computational science projects.

**Acknowledgments**

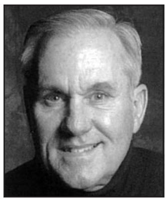
We thank the Osprey team members for participating in this case study. Air Force grant FA8750-05-1-0100 supported this research in part.

Osprey is a pseudonym; we have masked or omitted any details that might reveal the code project's identity.

## References

1. J.C. Carver, “Post-Workshop Report for the Third International Workshop on Software Engineering for High Performance Computing Applications (SEHPC 07),” *ACM SIGSOFT Software Eng. Notes*, vol. 32, no. 5, 2007, pp. 38–43.
2. J.C. Carver et al., “Software Development Environments for Scientific and Engineering Software: A Series of Case Studies,” *Proc. 29th Int’l Conf. Software Eng.*, IEEE CS Press, 2007, pp. 550–559.
3. L. Hochstein and V.R. Basili, “The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development,” *Computer*, vol. 41, no. 3, 2008, pp. 50–58.
4. D.E. Post, R.P. Kendall, and E. Whitney, “Case Study of the Falcon Project,” *Proc. 2nd Int’l Workshop Software Eng. for High Performance Computing Systems Applications*, ACM Press, 2005, pp. 22–26.
5. R.P. Kendall et al., *Case Study of the Hawk Code Project*, tech. report LA-UR-05-9011, Los Alamos Nat’l Lab, 2005.
6. R.P. Kendall et al., *Case Study of the Condor Code Project*, tech. report LA-UR-05-9291, Los Alamos Nat’l Lab, 2005.
7. R.P. Kendall et al., *Case Study of the Eagle Code Project*, tech. report LA-UR-06-1092, Los Alamos Nat’l Lab, 2006.
8. R.P. Kendall, D. Post, and A. Mark, *Case Study of the NENE Code Project*, tech. note CMUI/SEI-2006-TN-044, Software Eng. Inst., Jan. 2007.
9. E.E. Mills, *Software Metrics*, Software Eng. Inst. Curriculum Module, SEI-CM-12-1.1, 1988, www.sei.cmu.edu/pub/education/cm12.pdf.

## About the Authors

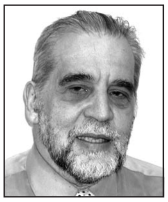


**Richard Kendall** is a visiting scientist at the Software Engineering Institute and retired CIO of Los Alamos National Laboratory. His research interests include computational methods in partial differential equations, computational physics of oil and gas exploration, computer security, software engineering, and software assurance processes. Kendall received his PhD in mathematics from Rice University. He’s a member of the Society of Petroleum Engineers and the IEEE. Contact him at rkendall@sei.cmu.edu.

**Andrew Mark** is the program manager for DoD software applications in the DoD High Performance Computing Modernization Program. His academic interests include continuum mechanics, and his professional interests include the development of integrated software tool suites for design, testing, and analysis of DoD materiel acquisition programs. Mark received his PhD in applied mechanics from Drexel University. He’s a member of the American Institute of Aeronautics and Astronautics and American Society of Mechanical Engineers. Contact him at amark@hpcmo.hpc.mil.

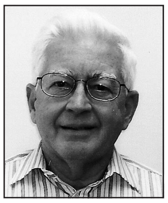
**Jeffrey C. Carver** is an assistant professor in the Computer Science and Engineering Department at Mississippi State University. Beginning 16 August, he’ll be an assistant professor in the Department of Computer Science at the University of Alabama. His research interests include empirical software engineering, software engineering for computational science, software architecture, and requirements engineering. Carver received his PhD from the University of Maryland. He’s a member of the IEEE Computer Society, ACM, American Society for Engineering Education, and International Software Engineering Research Network. Contact him at carver@cse.msstate.edu until 16 Aug., and at carver@cs.ua.edu after that date.

**Douglass Post** is the chief scientist of the DoD High Performance Computing Modernization Program. He is a permanent member of the senior technical staff of the Software Engineering Institute. At the HPCMP, he also manages the DoD Create program to develop large-scale computational engineering tools for designing ships, airplanes, and RF antennas. His research interests include the development of computational engineering and science tools and the associated software engineering issues. Post received his PhD in physics from Stanford University. He’s a fellow of the IEEE, American Physical Society, and American Nuclear Society. Contact him at post@ieee.org.



**David Fisher** is the chief engineer for the Create program and project manager for Create infrastructure in the Department of Defense High Performance Computing Modernization Program. He completed the work described in this article while he was at the Software Engineering Institute. His interests include computational science and engineering, high-performance computing, software engineering of complex systems, emergent behavior, and software development infrastructure. Fisher received his PhD in computer science from Carnegie Mellon University. Contact him at david.a.fisher@hpcmo.hpc.mil.

**Clifford E. Rhoades Jr.** is the technical director of the Maui High Performance Computing Center under an Intergovernmental Personnel Act assignment from the Software Engineering Institute. His research interests include computational physics, high-performance computing algorithms, and software engineering. Rhoades received his PhD in physics from Princeton University. He was one of the first five American Physical Society fellows elected in computational physics. He’s a member of the ACM, the Air Force Association, the American Institute of Aeronautics and Astronautics, the American Physical Society, the IEEE, and the IEEE Computer Society. Contact him at crhoades@hpcmo.hpc.mil.



**Dale Henderson** is retired from the Los Alamos National Laboratory after a long career in basic and applied research, with some focus on large-scale computation and computer simulation. Henderson received his PhD from Cornell University in applied physics. He’s a member of the American Physical Society. Contact him at denise-dale@newmexico.com.

**Susan Squires** is executive director of user research at Tactics, a user and customer research consultancy firm that specializes in ethnography to gain insight in connecting what people do to what they say. She completed the work described in this article while she was at Sun. She’s a practicing anthropologist with experience in customer research, strategic planning, and program management. Squires received her PhD in anthropology from Boston University. She is a fellow of the Society for Applied Anthropology. Contact her at susan.squires@acelere.net.