

# Bottom-up, Open, and Concurrent:

A Perspective on Development Processes for Scientific Software Systems

Jing Guo\* and Peter Lyster†

Data Assimilation Office, Code 910.3, NASA/GSFC

November 28, 2000

## Abstract

Development processes for scientific software systems are often characterized as *ad hoc*, if not chaotic. However, there are some common characteristics in these processes, including, *a*) development of software from the bottom-up with clear objectives, but with little up-front planning or formal design; *b*) open modification with little apparent central control; *c*) concurrent changes involving groups of loosely connected developers. Some of these practices may be properly challenged when the systems become large. However, maintaining certain aspects of them may be desirable, or even critical to the survival of some projects that rely heavily on the incorporation of cutting edge domain sciences and computational technologies. This paper will discuss these issues, and relate them to the development process of the NASA Data Assimilation Office's (DAO) Physical-space Statistical Analysis System (PSAS). Efforts have been made to learn from successful development experiences and to define software engineering practices that will sustain long-term development of the PSAS.

## 1 Introduction

Scientific software systems are characterized not only by their complexity but also by the sometime radical development efforts achieved by apparently undisciplined software engineering practices. There are often only small groups of elite developers who are privileged to make direct contributions to the major development stages of the software, such as, requirements analysis, design, implementation, and maintenance. Development processes for this class of systems are in general not well understood. Attempts to implement formal software engineering processes in these projects are sometimes challenged by the core developers. These formal processes are perceived to be inefficient for the rapid concurrent development efforts that are necessary to meet the scientific and technological requirements of the software products. We will discuss some of these issues in relation to the development of the Physical-space Statistical Analysis System (PSAS)[2] at NASA/Goddard Space Flight Center's Data Assimilation Office (DAO).

The PSAS is a complex scientific computing system designed to solve large-scale linear systems of equations defined by error covariance matrices with  $\sim 10^4 - 10^6$  unknowns. The PSAS was developed from scratch involving a total of less than a dozen developers over a ten year period in  $\sim 20$  man-years. The software core is a programming environment supporting advanced covariance modeling. Despite its own complexity, the PSAS

---

\*mailto:jguo@dao.gsfc.nasa.gov. Also affiliated: SAIC/General Sciences Corporation, Laurel, MD

†mailto:lys@dao.gsfc.nasa.gov. Also affiliated: University of Maryland Earth System Science Interdisciplinary Center (ESSIC), College Park, MD

is just one of several complex subsystem components of the DAO's Data Assimilation Systems.

The developers of the PSAS include scientists with advanced degrees in different fields, such as mathematics, meteorology, and physics, with strong background in scientific computing but differing levels of experience in operational software and system development. They were assembled from DAO, other organizations in NASA, and through outside hiring. Often, they shared a common interest in developing good software for scientific software systems, but had little or no previous training in software engineering before they joined the project.

The first version, circa 1994, was more than 10 thousand lines of code, while today the PSAS is about 90 thousand lines of code. The software has grown through several important development cycles and has continuously supported the DAO's scientific research efforts and its data assimilation systems. Through these development cycles, the consensus among developers has been to improve the development processes whenever and wherever they can. Much of the debate has surrounded the early attempts to follow a formal software engineering process for overall system development, and how much this process should be applied to the development efforts of scientific components such as the PSAS.

A practical, and perhaps less risky, approach to process improvement is to follow the initial process as the baseline, and improve it as necessary. However, looked at from the outside, the initial process appears to be nothing but *ad hoc* and relying on "heroic" efforts. Despite many successes of similar processes for large-scale scientific software systems, it remains open to such criticism. However, to experienced scientific software developers, a pattern of the processes, although generally unnamed, does exist in day-to-day practices. This pattern seems to be repeatable and produces quality scientific software. Also, this pattern is in fact well understood by developers in various scientific fields such that projects are often safely passed from one qualified developer to another through rather limited communication and documentation.

The purpose of this paper is to describe certain common characteristics of actual development processes, with a developer's perspective on why they are natural and important to scientific software. We will also discuss efforts to incorporate this understanding into software engineering processes that are used by the PSAS developers.

## 2 Characteristics of Actual Processes

In this paper, three common characteristics of actual processes for the development of scientific software systems are discussed. They are, *a*) development of software from the bottom-up with clear objectives, but with little up-front planning or formal design; *b*) open modification with little apparent central control; and *c*) concurrent changes involving groups of loosely connected developers. The details will be explained below.

### 2.1 Bottom-up

The development of a scientific software system is often started with clear scientific objectives, but very informal, if any, up-front requirements analysis, project planning, or software design at the high-level. Scientific software developers often start to write code based on a general description of equations, before many high-level issues considered critical to software engineers have been formally specified. This approach of developing code as soon as possible and delaying or even avoiding top-down up-front system wide activities is referred in this paper as *bottom-up*.

Despite possible reasons why this *bottom-up* approach could be problematic for project management, there are reasons why this approach is natural to a scientific software development process:

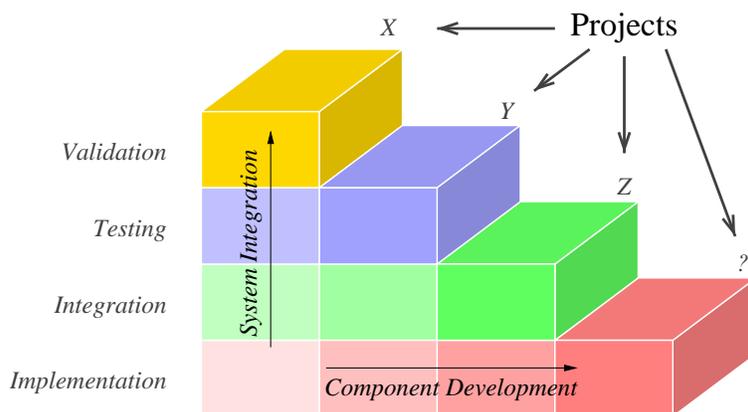


Figure 1: A *component development* process is driven by near- to long-term scientific and operational requirements, and requires constant innovation to incorporate cutting edge science and computational technologies. A *system integration* process is driven by individual projects, and focuses on realizing system integration, testing, and validation using available system components.

- Contrary to appearances, the up-front scientific objective of the project represents abstract yet unambiguously concrete requirements to scientific software developers. The scientific requirements may be subject to some detailed clarifications. However, rewriting the requirements in a language understandable by average software engineers could be very challenging. In some cases, it may even open the door to ambiguous or misleading interpretations.
- A system is constructed by interdependent software components. It can not be built if its components are not available. When a scientific requirement becomes well understood, the primary risk for a system development project would be any unavailability of its components. Therefore, it is just a common sense risk mitigation to rapidly construct (*i.e.* to code) a working prototype to allow further scientific and software developments, before any commitment to costly formal implementation efforts.
- A software component may support a number of system development efforts sequentially or concurrently during its life-cycle. This may be through single or multiple releases with similar but evolving requirements. Therefore, component developers must have a strategic view of the future direction of the component, while customizing the releases to meet the requirements of different projects.

In short, this *bottom-up* approach is *component developments first, system integration later*. Although it seems to be just a “common sense”, this approach suggests two separate dimensions in scientific software development processes. One dimension is for *component development* that is driven by near- to long-term strategic scientific and operational requirements of the organization, and focuses on supporting innovations in the domain science. Another dimension is for *system integration*, driven by individual projects and focuses on realizing integration, testing, and validation (Fig. 2.1).

By *bottom-up*, we do not exclude system wide requirements analysis, project planning, and software designs, but merely emphasize that different types of development may be better managed by following different process models with different objectives.

For example, despite many projects may wish otherwise, a *system integration* project can only be reliably supported on a reasonable schedule using components with reliable release time. Therefore, a *system integration* project should be advised to focus on understanding the baseline system as well as describing expected changes based on

available or near-term component releases. It should adapt a process that serves its purposes well, and avoid wasting its efforts on managing dynamic development processes of all its components through a top-down approach. On another hand, *component developments* may better serve an organization and its near- and long-term future system development projects by adapting a process that will promote innovations addressing critical scientific and operational requirements.

For most scientific software, the developers have to constantly seek improvements to the software, to meet the growing demands by its user community. Through the process, except the basic scientific and operational requirements, all aspects of software, including data structures, interfaces, algorithms, and even the architectures, could be altered whenever it is necessary. A software solution for a specific problem may finish its whole life-cycle from being proposed to being either accepted or rejected in minutes, hours, or days, depending on its implementation costs. In this context, software engineering practices to be applied to this class of component developments but with improper time scales, may be helpful neither to the quality of the software nor to the performance of a development team. In particular, high-level practices that unnecessarily increase the level of communications should be avoid.

## 2.2 Open

Changes to scientific software are often made directly by scientific developers without a formal change management process. This practice is often in conflict with software engineering efforts to enforce a centralized change management mechanism both for the resource management of a project and to ensure the software integrity of a system. *For a scientific software system, we believe that not all aspects of software change management should be centralized – in fact, reliable and more efficient decentralized mechanism are possible.* In another words, scientific software development should be made more *open*.

By *open* we do not mean, for example, placing code on an ftp site such that scientists in other parts of world can use the code for free. Nor are we referring to the generation of standards for application programming interfaces such that some software developers can develop standard conforming software components and others can develop applications using them. These are important but separate issues.

By *open*, we intend to allow, to invite, and to guide, users and other developers to contribute to the PSAS software, sometime without the consent of the main developers, in the forms of bug fixes, code improvement, customization, or major developments.

The reasons for *open* PSAS development are: First, as a component, the PSAS must support the DAO's system development efforts by customizing the software to meet the requirements of different system integration projects. However, for various reasons, the main developers are often not available or the best candidates for the jobs. Second, the user community of the PSAS software is an extremely valuable resource for its development as well as for its quality assurance. Third, no state-of-the-art scientific software system can be fully developed without helps from broader scientific community.

Based on these reasons, as well as the experience of the PSAS developers with software version management, including observations on the software development practice advocated by *Open Source*[4], the following practices have been consciously followed in the recent PSAS development activities to create an *open* environment:

- The principle of “release early and often”[4] is adapted to provide an effective *peer-review* mechanism that allows more people to review and to test the code informally before a decision may be needed to include a change or a new development in a release.
- System integrators are allowed to modify the code and commit the changes back to the software repository with or without main developers' consent, since the integrators have a larger responsibility to make software components to work together in the targeted systems.

- Other users or developers are also encouraged to contribute to the development of the PSAS through its software repository, from simple bug fixes to major developments, or some controversial new features.
- Except for obvious cases, the PSAS developers have tried to be inclusive to different solutions brought to the system by different developers by improving the architecture of the software and assisting developers on improving their software designs. By doing so, sensitive decisions could be deferred to a later stage of the system development and be made by system integrators or other users.

The central concept of this *open* approach is that users and developers are encouraged to make decisions on the implementation of a change, based on their own scientific as well as personal managerial judgment. It is often the case that only developers and users have the detailed knowledge about a software module in a complicated scientific software system – this knowledge having been achieved after extensive research or through much trial-and-error. Also, a change can often be implemented and tested quickly requiring little or no additional resource, if the decision has been made by the developer who is in the right place and at the right time to make the change. After a change is committed the new code will automatically be subject to code *peer-review* in the broader development environment. This practice is based on the concept described as “Linus’s Law” by Eric Raymond, or “Given enough eyeballs, all bugs are shallow” [4].

A common concern with the *open* approach for software change is the possibility of losing control. We should point out that with the *open* approach:

- Higher level controls over software changes are still practiced, except that the focus of higher level controls is on evaluating releases with changes already successfully implemented, not on evaluating all changes, including some that may never be implemented.
- Detailed decisions are made by people who have the detailed knowledge about the problems and the solutions, not by people who may be at higher levels but do not know enough details for a sound judgment. This should give people more confidence on the overall decision.
- Many aspects of code inspection and testing are now embedded in the process.
- Fragmented resources as well as developers with various skills of different levels can be more efficiently used.
- With a proper software version management, all changes can be retracted and reviewed down to the code level if necessary.

One scenario of a functioning team involves roles as developers who are given the responsibility to decide and implement *changes* in a software component. Another role is the *curator* who has separate responsibility for the *releases* of the component. The decisions to be made by the *curator* will be simplified and focused: for example, on whether certain changes meet relevant requirements; if a release should include certain changes; or if additional tests and resources will be needed to finalize the release.

An *open* development process requires a shared software repository. When properly managed, this combination is expected to be a powerful mechanism to encourage users and developers from different paths to work on the same scientific software in a collaborative way. The authors’ experience with the PSAS development has been positive towards this expectation. This expectation has also been further inspired by the success of the Linux operating system development [4]. However, details of the approach, as well as the possible metrics measuring the success of the approach, remain to be worked out.

## 2.3 Concurrent

It is very common for different scientific software developers to work on the same software at the same time, to address different problems, from the same or even different

software baselines. This practice creates a *concurrent* development pattern, which allows multiple developments to proceed in parallel. Developers often follow this development pattern unconsciously by working from copies of the same software, unawarely make changes requiring some major work before their developments can be integrated back into the same software. In fact,

- most development activities taking place are *concurrent*, and poorly managed;
- The more *open* the development is, the more *concurrent* these development activities become;
- Without a good mechanism to coherently organize these activities, *concurrent* developments will become redundant at the best, divergent as a normal result, and conflicting at its worse but not uncommon.

To address these issues, the popular Concurrent Version System (CVS)[3] has been used as the tool to support the PSAS software version management activities. Particularly, its version management model has been adapted as a guidance to understand the life-cycles of concurrently developed software. A code change, either small or large, will not be considered done if it has not been committed to the software repository.

A key issue for managing *concurrent* development is how to resolve conflicts between developments. Instead of avoiding conflict by either through a traditional *lock-modify-unlock* version control mechanism or *branching* the developments, the CVS' *copy-modify-merge* model[3] forces *concurrent* developers to face conflicts when and where conflicts occur. Changes are not allowed to be committed by the CVS until all appending conflicts are somehow resolved. On another hand, the CVS shows the code conflicts only when they physically occurred. If managed properly, conflicts between *concurrent* developments may occur much less than people imagined.

Additionally, conflicts between *concurrent* development efforts can be reduced by a more modular software design. This approach in turn allows the development to be more *open*. It also emphasizes a need for a knowledgeable *curator* in a natural conflict resolving process. The *curator* will determine the releases of the component, and issue a "decree" only if an agreement can not be reached between conflicting developers.

One interesting thing should be noted. Combining *open* and *concurrent*, developers are given more decision (making changes) and implementation (merging changes) responsibilities over the software. When this happens, developers become much more careful and creative to ensure that changes are consistent and graceful.

### 3 Summary

To many scientific software developers, the processes involved in the successful development of scientific software systems should be understood as fairly typical and repeatable. In this paper, we tried to identify some common characteristics of these processes based on our experience with the development of the PSAS, and described these characteristics as *bottom-up*, *open*, and *concurrent*. We believe that these characteristics are natural because of the complex, exploratory, and collaborative nature of scientific software developments. We also believe that these characteristics are critical to the success of scientific software system development projects of the similar class, since in a competitive scientific development environment, rapid innovations are critical to make scientific software systems acceptable by its user community, despite of the high risks involved.

On another hand, processes practiced by many scientific software development are often considered *ad hoc* by people from other disciplines of software system development. Many efforts were even made to implement "right" processes with mixed results.

Instead of trivializing the actual processes practiced by many scientific software developers, the authors believe that one should try to understand and apply successful

practices in these processes, then improve the processes by improving these practices not replacing them unnaturally, since these characteristics do not have to conflict with good software engineering practices. Nevertheless, requiring *bottom-up*, *open*, and *concurrent* may present challenges to the engineering management of similar developments, because few studies are available in this area. We are pleased to see that some studies are becoming available recently in this direction. For example, a study by Ambrosiano and Peterson[1] suggested that the process model for scientific research software development is not an engineering workflow, but an exploratory workflow. We hope that this paper represents our efforts of improving software engineering processes for sustainable scientific software development, and provides a useful perspective on development processes from a developers' point of view, based on our very preliminary observations.

## References

- [1] Ambrosiano, J., and M. Peterson, 2000, Research Software Development Based on Exploratory Workflows: The Exploratory Process Model (ExP), Los Alamos National Laboratory, LA-UR-00-3697.
- [2] Cohn, S.E., A. da Silva, J. Guo, M. Sienkiewicz, and D. Lamich, 1998, Assessing the Effect of Data Selection with the DAO Physical-Space Statistical Analysis System *Mon. Wea. Rev.*, **126**, 2913-2926.
- [3] Fogel, K., 1999, *Open Source Development With CVS*, <http://cvsbook.red-bean.com/>, The Coriolis Group (<http://www.coriolis.com>)
- [4] Raymond, E.S., 2000, The Cathedral and the Bazaar, <http://www.tuxedo.org/esr/writings/cathedral-bazaar/cathedral-bazaar/>