Under TheHood

BY RONALD J. NOWLING, TREVOR M. CICKOVSKI

Prototype to Release: Software Engineering for Scientific Software

Having engineered several scientific software applications for public consumption, the authors know from experience that the process offers unique challenges. Typically, the algorithms being implemented are complex; the process involves numerous developers with various backgrounds and skill sets; and it all takes place in a fast-paced environment where new methods must be prototyped and tested regularly.

Set Clear Objectives

Our experience suggests that these challenges can be overcome by establishing a clearly defined set of objectives for the engineering process. First, there must be an intention to produce stable software releases for public usage on a regular or semiregular basis. Second, the process has to allow software designers to implement new experimental methods without disrupting the release schedule or introducing bugs or destabilizing the software. Third, the process must allow for regular, rigorous validation and testing of the software to prevent the accidental introduction of bugs and ensure ongoing confidence in the correctness of the software under continuous development. And finally, the process needs to be amenable to involvement by multiple developers with varying backgrounds and skill sets as well as different goals and priorities.

To fulfill these objectives, we recommend and describe here a development methodology that we've found to work: the iterative model. Here, we also survey some tools that we've found useful for implementing that process.

Other development methods and tools do exist, but many people don't use any process at all. We hope this article will give people some new ideas, and encourage them to both reflect on their process and research the options.

Iterate Using Split-stream Engineering

A key step is to split the engineering process into two concurrent "streams"—stable and unstable. The stable version of the code forms the basis of regular releases; it is the trusted code base known to be relatively bug-free. Each release of the stable version is guided by a development plan with well-documented features and goals. Goals should be narrow enough to be accomplished in the given timeframe and significant enough to warrant a release. And the plan should include sufficient time for testing to ensure that the release will be stable and trustworthy. If a desired feature or goal has unexpected large-scale consequences or might otherwise hold up release, it can be dropped or moved to the unstable stream. Otherwise, we follow the development plan to its completion.

Meanwhile, the unstable version serves as a basis for developing prototypes and experimental features, as well as for making large-scale changes. Occasionally, it may be necessary to "rebase" the unstable branch to the stable branch to keep them in sync. As various changes mature and are shown to be correct through rigorous validation and testing, developers can create a copy of the stable branch to work on as an extra unstable branch. Once the feature is solidified, tested, and stable, developers can merge it into the stable stream as part of a future release. If conflicts develop between the stable and unstable streams, developers resolve them in the unstable branch. This is a continual process.

Goals should be narrow enough to be accomplished in the given timeframe and significant enough to warrant a release. And the plan should include sufficient time for testing to ensure that the release will be stable and trustworthy.

Use Appropriate Tools

Hosting and release platforms such as Simtk.org, SourceForge or GitHub can be used to facilitate development and to disseminate releases to the public. The appropriate platform should be chosen based on what team members are familiar with as well as available functionality.

Use of a versioning and revision control system (RCS, e.g., Subversion, Git or Mercurial) is essential as it tracks the history of a project's source code. As programmers reach various milestones, they can commit code to the repository, which takes a snapshot of the current state of the source code and determines line-by-line what changes from the previous version have occurred. Such systems also allow programmers to develop features independently from the work of other programmers, synchronize any changes, and resolve conflicts.

RCS systems tend to follow one of two models: (1) centralized, where all commits immediately appear in a cen-

tralized repository and (2) decentralized, where all commits are made to a local copy and are later copied to the centralized repository at the programmer's convenience. The decision of which RCS to use can be based on a balance among such factors as developer comfort, preference for centralized or decentralized repositories, and availability of support by the chosen hosting platform.

Test and Validate

In many scientific software domains, software testing and validation are also crucial. Testing refers to automatically identifying any changes in output. Validation is the process of determining if an output is correct, mathematically or scientifically speaking. Here, we do not further discuss approaches to validation because they typically depend on the specific scientific question being addressed. However, the details of testing merit some discussion.

Complex scientific algorithms are not only sensitive to small bugs but also susceptible to their introduction. Testing can serve as an "early warning" system for accidental bugs, help localize bugs to a specific function, and give developers a chance to think through the correctness of any changes.

Testing can be done at two levels: unit and system. Unit testing assesses an individual unit of code (such as a function or class) by providing a bit of input data and comparing the output against expectations. Unit tests are often written in conjunction with the code being tested and they are used early and often to check code as it's being written. Because they test parts independently, the developer should design the software as independent modules and reduce concrete dependencies using techniques such as abstract interfaces. In one approach, test-driven development (TDD), the programmer actually writes the unit tests before writing the implementation. Unit testing can often be automated and made easier through frameworks such as CppUnit (C++), JUnit (Java), unittest (Python), or FUnit or FRUIT (Fortran).

System-level tests check the integrity of the entire software system. These do not help localize bugs but can be used to determine proper integration of components, ensure dynamical run-time behavior has not changed, and enable comparisons with other software packages. Moreover, they are capable of testing higher-level results such as an entire simulation.

Document Assumptions

Developer documentation (which is distinct from user documentation) is another key part of the development process. It can help prevent mistakes by keeping everyone aware of assumptions and expectations. Basic forms of documentation include the use of descriptive variable names; clearly identified units; and other commentary. Document generators such as Doxygen (C++, others), Javadoc (Java), or Pydoc and Sphinx (Python), can automatically compile all API documentation assuming properly formatted comments. A more advanced documentation method, the design by contract methodology, provides a mechanism for specifying formal requirements for input (pre-conditions), guarantees for output (post-conditions), and maintained properties (class invariants) and can be enforced programmatically through language features such as assertions (which are manually specified by the programmer) and in more automated ways through preprocessors such as GNU Nana (C/C++) and Contract4J (Java).

Producing high quality scientific software starts with developing a culture or mindset that emphasizes quality. Here, we've discussed how requirements and a clear process can make a difference; described several types of helpful tools; and offered suggestions for good practices. We hope that these software engineering principles are helpful to others facing the challenges of preparing scientific software for public use. □

DETAILS

RJ Nowling is a PhD student at the University of Notre Dame in South Bend, Indiana, and has experience with several scientific software projects. He worked with the CONNJUR group to implement and release a file converter for NMR data and a visual programming integration environment for NMR data processing software; MimoSA, a database-driven system for annotating minimotifs from literature; DARWIN, a cataloging software for dolphins that uses semi-automated computer vision techniques; and ProtoMol, a molecular dynamics package designed for prototyping new algorithms. For the last two years, RJ has TA'd for Programming Paradigms, a required course for juniors that includes software development practices (unit testing, design by contract, design patterns). RJ recently visited Stanford as a Simbios OpenMM Visiting Scholar and was awarded a GAANN Fellowship.

Trevor Cickovski is an assistant professor at Eckerd College in St. Petersburg, Florida, where he teaches courses such as Data Structures, Compilers, and Operating Systems. Trevor has worked on ProtoMol; MDLab, a Python environment built on top of ProtoMol; CompuCell3D, a modeling environment and PDE solver for simulating cellular behavior; and GPU DePiCT, a GPU implementation of software for designing degenerate DNA primers.

Most recently, RJ and Trevor have worked on OpenMM FBM, a GPU implementation of the Flexible Block Method, a fast approximation method for the diagonalization of matrices in the context of normal mode analysis of proteins.