Mission-Critical Development with Open Source Software: Lessons Learned

Jeffrey S. Norris, *Jet Propulsion Laboratory*

By the time this article is published, NASA's Mars Exploration Rovers will be nearing the end of their seven-month journey to the red planet. Once Spirit and Opportunity are safely on the ground, mission operators at the NASA Jet Propulsion Laboratory will use a suite of software tools to analyze data acquired by the rovers and direct their activities. The development of one of these tools, the Science Activity Planner,[1] relied extensively on open source software components.

This article describes the lessons learned from the development of SAP and discusses how open source software can play an integral role in mission-critical software development both inside and outside NASA.

Using open source software components in mission-critical projects can not only keep the projects within budget but can also result in more robust and flexible tools.

Science Activity Planner

SAP is the primary science downlink analysis and uplink planning tool for NASA's Mars Exploration Rover (MER) Mission (http://mars.jpl.nasa.gov/mer). During each sol (Martian day) of mission operations, scientists and engineers use SAP to visualize the data acquired by the rover on previous sols and develop a plan of activities for the rover to accomplish on the next sol. Other operations tools then refine the plan and transmit it to the spacecraft. NASA classifies SAP as a mission-critical application because a failure in SAP could jeopardize an entire sol of operations.

Engineers also use SAP to operate research rovers in the Jet Propulsion Laboratory's Mars Technology Program and are developing a version of SAP for use in NASA's 2009 Mars Science Laboratory mission. Finally, SAP has played a major role in NASA's public outreach efforts: versions of the SAP software have been freely available for download by the general public since the 1997 Mars Pathfinder Mission. The public version of the MER version of SAP can be downloaded at http://wits.sdsu.edu.

SAP provides a broad range of capabilities, including 2D and 3D data visualization, image

0740-7459/04/$20.00 © 2004 IEEE

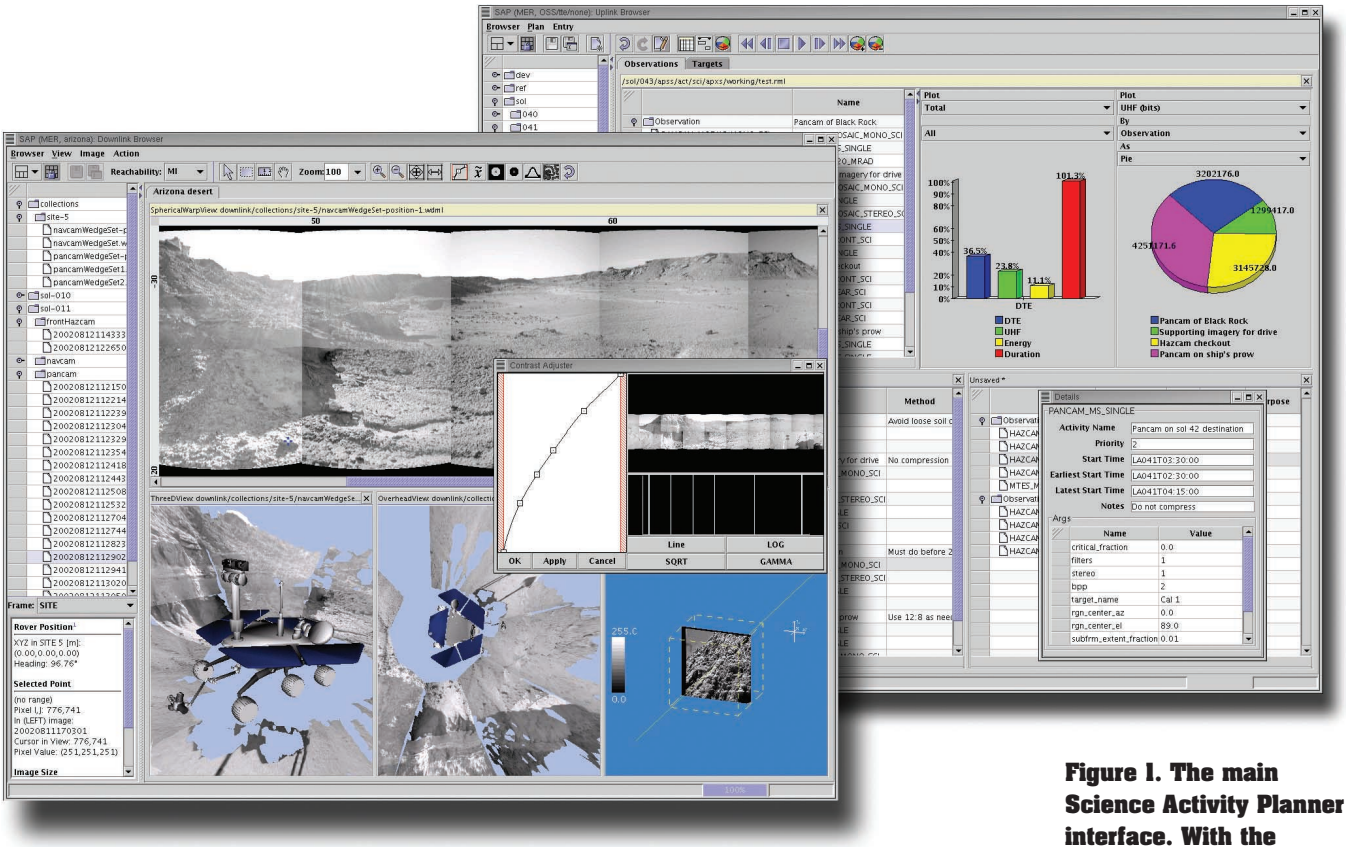

Figure 1. The main Science Activity Planner interface. With the downlink browser (foreground), users view and analyze images and other data in various ways. With the uplink browser (background), users build, simulate, and validate plans for execution by the rover.

processing, resource analysis, distributed operations, and rover simulation. Figure 1 shows some of the features of the main SAP interface.

Spirit and Opportunity carry the most sophisticated set of scientific instruments ever sent to the surface of Mars, and these instruments demanded major advances in nearly every part of SAP's downlink visualization and uplink planning architecture. SAP needed to be able to visualize data sets an order of magnitude larger than it had encountered in previous missions while letting users analyze the data in fundamentally new ways. The complexity of modeling these instruments' performance required the development of a new simulation system within the tool as well. These challenges called for a complete redesign and redevelopment of SAP for the MER mission.

The project's ambitious goals, coupled with a modest development budget, required a departure from the traditional in-house development model used for most mission-critical software (and for previous versions of SAP). Early on, the SAP development team adopted a design philosophy that sought to satisfy as many of the requirements on our tool as possible with open source components developed and maintained outside JPL. This decision affected nearly every part of SAP's redevelopment, from design to delivery.

Developing SAP with open source software

We began development of SAP with an intensive period of requirements definition in cooperation with the scientists and engineers who would be the tool's primary users. Requirements definition included many Web "shopping trips" to search for open source components that fit customer requests. Identifying such components let us more confidently assess a requirement's cost than if we had to consider developing it ourselves. In a few instances, we even recommended changes to a requirement when we felt that by doing so we could satisfy it with an open source component. Our customers responded positively to these suggestions, especially when they realized that we could use the resources saved to develop more features.

During the early stages of SAP's design, we avoided asking "How can we develop this capability?" and instead asked "How can we avoid developing this capability?" By focusing

Open Ops

The Science Activity Planner uses the following eight open source components for critical mission operations tasks, ranging from writing rover activity plans to calculating resource usage:

- *Castor* (http://castor.exolab.org), a data-binding framework that lets SAP move data between XML files, Java objects, and SQL databases. Castor is the core of SAP's input and output functions.
- *Java Expression Parser* (JEP, www.singularsys.com/jep), a system for parsing and evaluating mathematical expressions that SAP uses to process resource formulas from the mission activity dictionary.
- *Xerces-J* (http://xml.apache.org/xerces-j/index.html), a validating XML parser that SAP uses with Castor to read and write all official mission formats.
- *MySQL* (www.mysql.com), a database application SAP uses to synchronize data between multiple program instances.
- *MySQL Connector/J* (www.mysql.com/products/connector-j/index.html), a middleware component that converts Java Database Connectivity calls in SAP to the SQL network protocol.
- *HSQL Database Engine* (http://hsqldb.sourceforge.net), a Java SQL database embedded in SAP to give the application a local database when a shared MySQL database isn't available.
- *Virtual Reality Modeling Language*, or VRML97 (www.web3d.org/technicalinfo/specifications/vrml97), a geometry loader SAP uses for 3D spacecraft models.
- *Skaringa* (http://skaringa.sourceforge.net), a data-binding framework similar to Castor. SAP uses its time-formatting functions.

on developing as little as possible in-house, we could save our precious development resources for the most deserving tasks.

Selecting the final set of open source components was a complex process, as I discuss later in the article. The "Open Ops" sidebar describes the suite of open source software SAP uses today. Most of these components serve critical roles in the SAP application, and none are peripheral to SAP's mission.

We not only incorporated open source in the SAP application, but also relied heavily on open source development tools. In addition to the familiar GNU Emacs editor and the Concurrent Versions System (CVS), our team used JUnit for unit testing, Xalan-J for XSL translation, and JavaCC to generate a parser for image arithmetic expressions.

SAP's development saw more than its share of setbacks. Budget overruns elsewhere in the mission required two deep budget cuts amounting to nearly 50 percent of the total SAP development budget. An evolving operations process and unstable requirements required several late-stage redesigns.

SAP's reliance on open source let it readily adapt to many of these challenges. For instance, one redesign demanded eliminating a SAP capability that an open source component was to satisfy. Because we hadn't invested any SAP development resources or procurement funds in that capability, its elimination had little overall impact. It might seem counterintuitive, but our experience indicates that using open source can often make a project more nimble because its resources are concentrated on the system's core architecture instead of specific features.

Although SAP has been cleared for release to the general public in binary form, obtaining clearance to release its source code is a far more daunting prospect given its mission-critical status. This posed a problem for the use of two components that were released under the restrictive GNU General Public License, which requires all applications linked to the code to also be open source. Fortunately, in both cases the open source suppliers let us purchase a less restrictive license for a small fee, and tossed in priority technical support as part of the deal.

Development results

Without exception, we consider the use of open source components during SAP's development a success. The quality of the open source components we used was excellent. In fact, overall they were of better quality than two commercial components we purchased for thousands of dollars.

We also encountered fewer bugs in the open source components. Hoping it would let us diagnose problems in the commercial components more effectively, we paid additional fees for access to their source code. With one exception, the source code for the open source components was better documented and easier to understand than the source code for the commercial components. Admittedly, this sample size is quite small, but perhaps the greater exposure of open source code encourages developers to write better code.

The most striking difference between our experiences developing with commercial and open source components was the open source developers' responsiveness. When we contacted open source developers about a problem in their component, they responded immediately with workaround suggestions and kept us informed as they worked to correct the issue. In one case, they diagnosed the problem, fixed it, and released a corrected version in less than a day.

When we encountered a problem in a com-

mercial component, we often had difficulty even getting in contact with the developers. It became clear that the most we could hope for was for the problem to be fixed in the commercial product's next release, which was often months away.

Because the commercial components' life cycle was so long, we were less motivated to take the extra time to submit bug reports—we often fixed the problem in our copy of the source code and continued developing. When we discovered and fixed a problem in an open source component, however, we submitted the fix to the developers because we could download an updated version in just a few days. The reward for contributing fixes to open source projects might explain the consistently higher quality we encountered in these projects.

It might seem that open source would decrease the system's overall reliability, but our experience indicates that this is incorrect. Consider first the current test strategy for the SAP components developed in-house. We test these components to varying degrees depending on several factors. The most tested portions have custom-built automated unit test suites and simple interactive applications with which developers can test interactions between components. Finally, system-level and user-acceptance testing evaluate each component's performance in the complete system.

Many of the open source components we selected included detailed unit- and component-level tests that met or exceeded the testing level in most other portions of SAP. In addition, all the components we selected have large, active user groups that constantly use the open source, report bugs, and submit fixes. In a way, each of these users acts as a component-level tester for SAP and for every other user of the open source project. Obviously, none of the in-house portions of SAP has thousands of users, and no one outside the SAP development team has reviewed the source code for these in-house components. We therefore believe that introducing open source into SAP has improved the application's overall stability.

The development of the SAP data synchronization system illustrates many of the positive features of open source development. The initial SAP design included a powerful data synchronization application that the SAP developers working on the MER mission and a researcher in JPL's technology program were to develop. Unfortunately, after development had already begun, simultaneous budget cuts in the mission and the technology program decimated the resources available on both sides of the cooperation. We diverted additional resources from the mission budget in an attempt to compensate for the loss in technology funding, but the resources available proved insufficient to develop a usable system.

Over a year into development, we had to decide whether to spend our quickly diminishing resources in an attempt to complete our current system or change to a less complicated design in the hopes of recovering some of our losses. After careful study, we discarded our in-house system for a new implementation based on four open source components (Castor-JDO, MySQL, MySQL Connector/J, and HSQLDB). Whereas our in-house effort had required several person-months to partly complete, the new implementation was both more stable and more functional in less than one month. In the end, we delivered the synchronization capability to our customers ahead of schedule despite the time lost on the first implementation.

This example also calls into question the seemingly logical concern regarding open source's quality and stability. Whereas team members can tailor an in-house component precisely to their own requirements, an in-house system developed hastily will rarely approach the quality of a mature, widely used open source project. Suggesting that we could develop a synchronization server in a few months that would rival the performance and reliability of MySQL, which has been in development for nearly 20 years and is used by thousands of companies around the world, would have been ridiculous. Furthermore, none of our developers had experience developing high-performance, robust server software. We were more than happy to trust this portion of our software to open source developers.

An in-house system developed hastily will rarely approach the quality of a mature, widely used open source project.

Suggestions for future mission-critical projects

We compiled our experiences developing SAP into a developer's guide for those considering using open source in their mission-critical application. In addition to discussing how to evaluate an open source component's suitability for inclusion in a mission-critical application, the guide suggests strategies for working with open source development teams.

Keep in Touch!

Poul-Henning Kamp

In 1994, I wrote a one-way password scrambler, based on the MD5 algorithm, to avoid entangling FreeBSD in the International Traffic in Arms Regulations (ITAR) crypto dual-use export rules (see www.freebsd.org/cgi/cvsweb.cgi/src/lib/libcrypt/crypt.c#rev1.2). I released the source code under the "beerware" license, which lets people use the code however they want as long as they don't remove the license clause or claim they wrote the software; the beer "fee" is entirely discretional.

This relatively small piece of software has since had a solid open source career:

- It's been FreeBSD's default password protection algorithm since release 2.0 in 1994.
- NetBSD (http://www.netbsd.org) and OpenBSD (www.openbsd.org) adopted it, and OpenBSD extended the basic concept using stronger algorithms. Both projects imported the version from FreeBSD and have the algorithm's history in their version control systems.
- The GNU C Library (www.gnu.org/software/libc/libc.html) adopted the algorithm, but rather than ask my permission to relicense the implementation, they rewrote it to get it under the GNU license. The unfortunate side effect is that their implementation has no back reference to me or to the original FreeBSD implementation.
- Cisco Systems uses the algorithm to protect the highest-privilege password in their IOS router software: "enable." A Cisco employee confirmed that they imported the implementation verbatim from FreeBSD.
- The RIPE (Réseaux IP Européens; www.ripe.net/ripe) regional Internet registry uses the algorithm to authenticate database update requests. I believe they also use my source code unchanged.

Vulnerabilities

So what does this algorithm protect today?

The Cisco routers alone probably number in the low tens of millions, ranging from telco optical monsters to consumer broadband terminals. These routers carry the passwords to the accounts that control the Internet's configuration.

Interestingly, the "security by diversity" argument doesn't help us much here. Juniper is Cisco's only competition in the backbone router market. Its high-level processor runs JunOS, which is derived directly from the FreeBSD operating system and thus is likely to use the same algorithm.

NetCraft reported in July 2003 that FreeBSD powers almost 2 million sites with almost 4 million hostnames.[1] This number includes only the publicly accessible Web servers its automated survey found, however. No one has even tried to estimate how many FreeBSD machines run in other applications, from firewalls to turn-key components. For example, I was recently told that document-processing systems running FreeBSD handle approximately 10 percent of all business-to-business financial transactions (by both number and value) in the US.

Practically all Linux systems include GNU Libc, and I'd estimate that some millions of these systems worldwide—from home computers to IBM mainframes—default to the MD5 algorithm.

Password-protected Web pages often rely on the underlying operating system's password scrambler, and contain all sorts of nonpublic information. Most is probably porn, but anything from credit card numbers to health information to military secrets makes up the remaining fraction.

One-line summary: If I goofed in the algorithm or its implementation, all hell can break loose.

Before I started writing this piece, I hadn't fully realized what had happened to the piece of code I churned out, sitting

Evaluating open source software

Evaluating open source components for a mission-critical application is a delicate task. As we've discussed, open source components can accelerate development and cut costs, but the consequences of selecting the wrong component can erase these benefits. It's important to consider several characteristics when evaluating open source components.

Maturity. How well established is the open source project? Clearly, a mission-critical application is no place for someone's untried, untested code. In addition, a new project is more likely to unexpectedly shift focus or disband altogether than an older, established project. Developers of mature open source projects have also demonstrated their ability to make releases, handle bug reports, and support users.

To evaluate a project's maturity, consider both the amount of time the development team has invested and the number of releases the team has made. Projects in alpha or beta stages of development are usually poor choices for mission-critical applications. For projects that don't clearly state their stage of development, consider the state of the project relative to the development team's stated goals. If many features still await development, the project is probably not mature enough for your project.

Software reliability is vital in mission-critical development. Every piece of open source you adopt must become part of your software's overall test plan. Open source components that include automated unit and component-level tests make this task much easier. Without these features you might need to write automated acceptance tests to determine

on the floor while my son learned to crawl. And honestly it scares me more than a little. Fortunately, I have no reason to believe that any problem exists with either the algorithm or the implementation, and given that MD5 is pretty strong, it's unlikely that any will ever be found.

What if ... ?

But imagine I did goof, and I find a serious flaw in the algorithm. I believe in full disclosure of software security problems. I also believe in giving "the good guys" a fair head start whenever possible. But could I provide this?

Because I know they use the algorithm, I could contact security people at FreeBSD, Cisco, GNU, and RIPE before CERT (originally the Computer Emergency Response Team) discloses the problem to the public. Within hours, or at most days, security advisories and patches covering more than 90 percent of the deployed systems would be distributed and installed.

But what about the users I don't know about? How many organizations have imported my algorithm into products, turn-key solutions, or in-house applications? Not knowing who they are, I couldn't warn those customers before the public disclosure. They would find out via the official CERT advisory at the same time as the bad guys. And that's assuming they realize that the advisory is relevant to them.

The situation is not much better in reverse: Assume instead one of the algorithm's users found a problem and tried to alert me. For reasons aptly described by Alan Greenspan as "irrational exuberance,"[2] the email address I used in the original source code license no longer exists, so the user's first attempt to contact me would almost certainly fail.

My name "Googles" well, so the user should be able to find me; however, an open source author named Bob Smith would not be as lucky. Moreover, a user starting from the GNU Libc reimplemented version wouldn't even have my name to use in a search.

A new risk

Software development, or more precisely software maintenance, must find a way to mitigate this new risk, brought on by the open source phenomenon. (Given that the legal connection between the software supplier and the software consumer consists mostly of text amounting to "You can't sue me, ever!" the legal department, for once, can't mitigate the risk.)

I don't think any general solution to this problem exists; users in each case will have to weigh the risks and the resources to find an acceptable compromise. Consequently, I won't offer a solution.

I do have one idea, however. Send the author an email saying, "Please treat this as confidential information: FroBoz Dam Construction used your software in our flood control dam #3 project. Thank you very much!"

Open source authors are driven by honor and recognition; thus, I predict that more often than not, authors will archive such email in a vanity folder from which they can and will cull return addresses should the need arise.

So keep in touch (just in case).

References

1. "Nearly Two Million Active Sites Running FreeBSD," *Netcraft News*, 12 July 2003, http://news.netcraft.com/archives/2003/07/12/nearly_2_million_active_sites_running_freebsd.html.
2. S. Reier, "Five Years Later, Greenspan's 'Irrational Exuberance' Alert Rings True," *Int'l Herald Tribune*, 1 Dec. 2001, www.iht.com/articles/40648.htm.

Poul-Henning Kamp owns a consulting company and is a major contributor to the FreeBSD operating system and a gaggle of other open source projects. His research interests are infrastructure design in Unix kernels and precision computer time-keeping. The computing business has effectively distracted Kamp from his studies; at the current rate he might finish a BSc in 2038, just in time for the Unix 231 time_t rollover. Contact him at Herluf Trollesvej 3, DK-4200 Slagelse, Denmark; phk@phk.freebsd.dk.

whether it's safe to integrate a new version of the component. The presence of a cohesive, well-thought-out test plan is an excellent indication of the open source project's maturity.

Longevity. How long will the open source project survive? Ideally, all the open source components you select will be consistently improved and supported throughout the life of your application. If the open source project shuts down, your team will have to support the component in-house—clearly a situation that a mission-critical project should try to avoid.

How, then, do you evaluate an open source project's longevity?

A mature project is more likely to survive, so start by considering the project's maturity using the metrics described earlier. Next, check the dates of the last releases, news posts, and messages in user forums and mailing lists to assess developer and user activity on the project. Some open source projects are like ghost towns—they appear to be established, quality projects when in reality the developers and users have long since left the code to rot. SourceForge.net makes it much easier to assess the level of activity on a project by assigning projects an *activity percentile* based on several factors, such as commits to the source repository, forum use, and bug tracker facility use.

Finally, look for a large, diverse development team. If an open source project is largely the work of one developer, its life depends on that developer's willingness to continue the project. Also be careful if all the developers work for the same company or university. If their organization suddenly changes priorities, the project might not survive.

Why should we spend taxpayer dollars developing something that already exists?

Flexibility. How well will the open source project respond to your project's changing needs? Even if an open source component seems absolutely perfect when you pick it, you'll likely discover a bug or want a new feature eventually. Some open source teams are receptive to user suggestions while others have a more rigid conception of what their project should become. If the team is inflexible, depending on their project is much riskier.

When evaluating an open source project's flexibility, first make sure that you're planning to use the software the way it was intended to be used and the way most other users are using it. If you stray too far from the beaten path, the developers might not want to help you when you run into problems.

Next, spend some time reading the users' forum or mailing list. Does the development team answer user questions in a timely fashion? Does it seem open to suggestions? Don't underestimate the importance of an active, interested user community. Users of an open source component are often a valuable source of support.

Finally, get a feeling for the development team's willingness to support your work by sending an email to a couple of the developers introducing yourself and describing how you'd like to use their system.

Working with open source developers

Having chosen and integrated an open source component into your program, you might consider your interaction with the open source team complete. However, in the most productive arrangements with open source developers, this is only the beginning. If the open source component will be part of your application's core, your goal shouldn't be a simple "code drop." Injecting thousands of lines of code that someone else wrote into a mission-critical project is a risky proposition. Instead, you should aim to establish a long-term working relationship with the open source team. Poul-Henning Kamp further addresses this issue in the "Keep in Touch" sidebar accompanying this article (see p. 46).

The relationship between an open source developer and the project's users is complicated. Clearly it differs from the relationship between vendor and customer because the open source user is not paying the developer. It also differs from the relationship between coworkers because the open source developer might have very different goals for the component than the user does. The relationship is most similar to a casual partnership between departments at a university. The open source developer and user work toward their own goals, and the partnership flourishes as long as the relationship benefits both parties.

At first glance, the open source user might seem to be receiving all the benefits in this relationship. What, then, motivates an open source developer to write thousands of lines of code and give it all away? Several of the ones we work with indicated that the desire to see their creations used by lots of people in exciting and interesting ways and to improve as developers through the increased exposure of their work are primary motivations. Therefore, one of the best payments you can give an open source developer is simply to send them an email periodically telling them how you are using their product. If you are doing something exciting with their work, try to include them in that excitement.

Looking ahead

SAP is not the only mission-critical piece of software at NASA that was developed using open source. In fact, SAP is not the only piece of software within the MER mission to use this strategy. Open source development is building momentum at the agency. This movement is a natural outgrowth of one simple question: Why should we spend taxpayer dollars developing something that already exists?—a question that led NASA to embrace commercial off-the-shelf hardware as a viable alternative to in-house hardware development many years ago.

Development with COTS software components is a natural step on the way to open source development. NASA has formally examined COTS-based software development,[2] which is becoming more prevalent each year. Development with open source is a natural extension of COTS-based software development; however, it might take longer to catch on because there is no analogue to open source in the hardware world. Try to imagine a computer manufacturer giving free laptops to anyone who asks for one and you'll begin to understand why some people have a tough time comprehending how open source works. The "Openly Skeptical" sidebar addresses some common questions about incorporating open source into a project.

If open source development becomes commonplace at NASA, the agency might take a

"giant leap" toward the idea of releasing much of the software that NASA develops to the open source community. As long as issues of national security are properly addressed, it's hard to imagine how this wouldn't ultimately benefit NASA, the US, and the rest of the world. A recent report from NASA Ames Research Center indicates a growing interest in this concept within NASA,[3] but it will likely be some time before any official steps are taken to make open source the norm in our agency.

The SAP development team chose components from among existing open source projects. If we were unable to find a component to suit a particular need, we developed that component ourselves. Future projects need not limit themselves in this way. The open source community is home to some of the most talented developers in the world, most of whom get involved with projects merely because they enjoy writing software and like to see their work used in interesting ways. Even if the entire project wasn't open source, a project team could still pick mission-critical components that were likely to be useful and interesting to people outside their organization and found open source projects to jumpstart their development. A project could easily double or triple the size of their development team this way with little cost to their organization.

Open source is already revolutionizing software development in many areas, and the realm of mission-critical development is likely to follow. Fundamentally, open source offers an attractive third option to the build vs. buy question. Where "build" offers flexibility and "buy" offers accelerated development, incorporating an open source component might offer the best of both worlds. Open source is "buy" without having to spend anything and "build" without having to develop anything—it's hard to imagine a better deal.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

Openly Skeptical

If you choose to incorporate open source components in your mission-critical application, be prepared for some hefty skepticism from your customers, management, and even yourself. Here are some common questions and answers.

- *What if the open source developers quit the project?* If the project is really dead and your application depends on it, you'll have to maintain the code in-house. Some people might think you'd avoid this risk by developing in-house to begin with, but what if the component's chief developer leaves your company without finishing the work? You can't avoid this risk—you can only mitigate it by getting to know the developers you trust with critical parts of your system.
- *If this component is so valuable, why is it free?* Many people seem to think that free means worthless. Some skeptics even prefer that you buy a component than get it for free. Remind these people that open source developers are deriving reward through means other than direct sales of their product.
- *How do you know the product is of sufficient quality?* For starters, spend a few hours evaluating the open source product—from reading user comments to experimenting with the component on your own. Also consider your development team's specialties relative to the open source team's. If your developers are simulation experts, can they write a better encryption system in a few months than an open source team of cryptography experts wrote in a year?
- *What about licensing issues?* Depending on what you're planning to do with your application, you might need to be careful here. A full discussion of open source licensing issues is outside this article's scope (see Michel Ruffin and Christof Ebert's article "Using Open Source Software in Product Development: A Primer" elsewhere in this issue), but keep in mind that many open source projects under restrictive licenses will grant you a much more liberal license for a nominal fee. You can often begin developing with the software immediately and pay the fee only when you're certain that the product matches your needs.

About the Author

Jeffrey S. Norris is a senior computer scientist in the Telerobotics Research and Applications Group of the Mobility Systems Concept Development Section at the Jet Propulsion Laboratory, California Institute of Technology, and development team lead for the 2003 Mars Exploration Rover Mission Science Activity Planner. His research interests include collaborative, immersive operations for Mars rovers and landers, science data visualization, and advanced human-computer interfaces. He received his bachelor's and master's degrees in electrical engineering and computer science from MIT and is working toward a PhD in computer science at the University of Southern California. Contact him at Mail Stop 264-422, Jet Propulsion Lab., Calif. Inst. of Technology, 4800 Oak Grove Dr., Pasadena, CA 91109-8099; jeffrey.norris@jpl.nasa.gov.

References

1. P.G. Backes et al., "The Science Activity Planner for the Mars Exploration Rover Mission: FIDO Field Test Results," *Proc. 2003 IEEE Aerospace Conf.*, IEEE Press, vol. 8, 2003, pp. 3525–3540.
2. M. Morisio et al., "Investigating and Improving a COTS-Based Software Development Process," *Proc. 22nd Int'l Conf. Software Eng.* (ICSE 2000), ACM Press, 2000.
3. P. Moran, *Developing an Open Source Option for Nasa Software*, tech. report NAS-03-009, NASA, 21 Apr. 2003.