



www.computer.org/itpro

CIO Corner

Tom Costello

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

IEEE  computer society

© 2011 IEEE. Reprinted with permission from IT Professional. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/web/publications/rights/index.html.

Open source software use is amassing a long list of benefits, among them higher software quality. Enterprises that are still discouraged by the task of converting existing software might find that modernization building blocks can make that task easier. Adapting or converting to OSS need not be overwhelming, as this case study shows.

Phillip Laplante, Anthony Gold, and Thomas Costello



Open Source Software: Is It Worth Converting?

In his famous treatise, Eric Raymond compares traditional software development to a select group of secretive individuals planning and building a cathedral and open source software (OSS) development to the collaborative market environment of a bazaar (E. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly Linux, 2001). Probably the most oft-cited quote from that work—"given enough eyes, all bugs are shallow"—is not simply an endorsement of OSS as a business strategy, but a stunning declaration that software could become defect free.

But is Raymond correct that the quality achievable with OSS is higher than that expected with closed source software? At least one study cites evidence that defects are located and fixed more rapidly in OSS than in closed source software (J.W. Paulson, G. Succi, and A. Eberlein, "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Trans. Software Engineering*, vol. 30, no. 4, 2004, pp. 246-256). The study also asserts that OSS is easier to administer and sim-

pler to deploy, has more influence on product direction, and enables more rapid troubleshooting and bug fixes. Software qualities such as maintainability, portability, and testability could arguably be better in OSS than in closed source software.

As the sidebar "Open Source Projects" describes, thousands of projects exist, yet many organizations are still reluctant to use OSS in deployable code. In large part, this hesitation stems from an inherent distrust of OSS quality. However, arguments are strong that OSS use *can* achieve quality levels beyond that possible with closed source software.

There are many ways to use OSS in the enterprise, ranging from a few components to single applications, suites of related applications, and complete enterprise solutions. With such a diversity of applications, OSS use is even more attractive, yet it is not a trivial undertaking. The logistics of moving to OSS need not be daunting, however, if the organization considers the entire modernization plan, considers what various application types require, and does not undertake too much conversion at once.

Inside

Open Source Projects A Defining Difference

Some of the Open Source Systems section was adapted from *What Every Engineer Should Know About Software Engineering*, by P. Laplante, Taylor & Francis Publishing, 2007.

THE CASE FOR QUALITY

Software quality, of course, means different things to different people. For our purposes, higher quality means enterprises can enhance certain software attributes by using OSS instead of closed source software. These attributes include security, ease of evolution, and the common “ilities”: maintainability, testability, reliability, understandability, and operability.

Security

Hoepman et al. argue that OSS is essential to building secure systems. Their main argument is that “opening the source allows an independent assessment of the exposure of a system, and the risk associated with using the system makes patching bugs easier and more likely and forces software developers to spend more effort on the quality of their code” (J.-H. Hoepman and B. Jacobs, “Increased Security Through Open Source,” *Communications of the*

ACM, vol. 50, no. 1, 2007, pp. 79-83). They further state that OSS is easier for multiple security teams, outside vendors, or independent agencies to assess.

Ease of evolution

Another hallmark of OSS is a vibrant community, and if the community is vibrant, the software is more likely to rapidly evolve new features and bug fixes. Feature innovation in OSS does not depend on a single entity—for example, a vendor—so the software product is easier to change and evolve. Open source also allows for code branching—the process by which a new community forms around an existing application code base to take the features and functions or underpinnings in a different direction than that of the original team. In effect, rather than forcing a product team to choose between two possible product directions, the code may evolve in two parallel or divergent paths, each taken by a different community team.

Open Source Projects

Open source use goes well beyond infrastructure software onto the desktop and into specific applications in many different problem domains. The Open Source Initiative (www.opensource.org) gives a complete definition of OSS. The thousands of open source projects include games, programming languages, tools, and enterprise-level applications. Clones of many well-known desktop and enterprise applications are also available in open source. Table A presents the different types of open source software projects, their objectives, control structure, and sample projects (K. Nakakoji et al., “Evolution Patterns of Open-Source Software Systems and Communities,” *Proc. Int’l Workshop Principles of Software Evolution*, ACM Press, May 2002, pp. 76-85).

Some of the best known OSS projects are the Linux operating system, the Mozilla Firefox Web browser, and the Apache Web server. Developers commonly use OSS tools—for example, the scripting languages Perl, Python, PHP, and Ruby; Ant and Maven for building applications; XUnit for testing; CVS and SubVersion for source code control; and Eclipse or NetBeans as integrated development environments. The business application layer has seen rapid growth with the availability of robust applications for CRM, accounting, ERP, BI/KM, document management, content management, corporate email, PBXs, and Web 2.0. Open source desktop environments and related end-user desktop applications have increased in both numbers and functionality.

Table A. Various kinds of open source software projects.

Type	Objective	Control style	Community structure	Examples
Exploration-oriented	Sharing innovation and knowledge	Cathedral-like central control	Project leader, many readers	GNU, Perl, Linux Kernel
Utility-oriented	Satisfying an individual need	Bazaar-like decentralized control	Many peripheral developers, peer support to passive users	Linux system (excluding the Kernel)
Service-oriented	Providing stable services	Council-like central control	Core members instead of a project leader, many passive users that develop systems for end users	Apache PostgreSQL

ilities

By its open nature, OSS enhances maintainability. Not only is the software more available for maintenance, but active OSS communities participate in and cultivate a high level of interest in the software's viability. In many cases, the code team is also part of the user population and has a vested interest in the product's functionality and growth.

Because OSS code is readily available, testability is easier, both functionally and structurally. And since open source communities are very comfortable with full disclosure of bugs (as opposed to hiding them or treating them as features), software users can more easily track defects and their workarounds. The argument that thorough testing is impossible because of the absence or incompleteness of software requirements (as is frequently the case in OSS projects) doesn't hold, since some testing approaches don't need such requirements (A. Elcock and P. Laplante, "Testing Without Requirements," *Innovations in Systems and Software Engineering: A NASA J.*, vol. 2, Dec. 2006, pp. 137-145).

For the same reasons that testability is easier, reliability is also higher with OSS. And OSS's open nature also means greater understandability, since there are no black boxes.

Finally, interoperability is much greater with OSS because projects almost always use shared components, and these components tend to comply with relevant international standards. Frequently these standards are the main form of requirements satisfaction for all externally furnished (including open source) components available. With the recent formation of the Open Solutions Alliance, there is an even stronger push to define OSS interoperability standards.

Documentation concerns

Most open source software projects do not include a great deal of such documentation as requirements specifications, design documents, test plans, or user manuals. But the common absence of that documentation does not necessarily indicate a lack of understandability or traceability. If this were the case, the same claim could be made for software developed using agile approaches.

REAPING THE QUALITY BENEFIT

Assuming OSS can enhance certain software qualities, an enterprise must ask two questions before deciding to use OSS to improve those qualities: Are we ready, and how do we do it? The answer to the first question is complex. Readiness standards are available in CapGemini's Open Source Maturity Model (www.seriouslyopen.org), Navica's

Open Source Maturity Model (www.navicasoft.com/pages/osmm.htm), and The Business Readiness Rating (www.openbr.com).

The answer to the second concern—how we use OSS to enhance quality—assumes that the organization is culturally ready to adopt OSS and understands what business areas would benefit from OSS use. To maximize quality, an enterprise must

- determine the benefits and risks of OSS,
- determine total costs, including switching and ongoing costs,
- analyze internal and external support resources,
- assess the impact on organizational performance, and
- assess the health of an OSS product's ecosystem—that is, the community of developers, testers, and users of the software—for its vigor, organization and resilience.

Because OSS code is readily available, testability is easier, both functionally and structurally.

An enterprise can begin integrating OSS in one of two ways: green-field adoption, which is to use OSS from the outset, or brown-field modernization, which is to convert existing software to OSS.

Green-field adoption

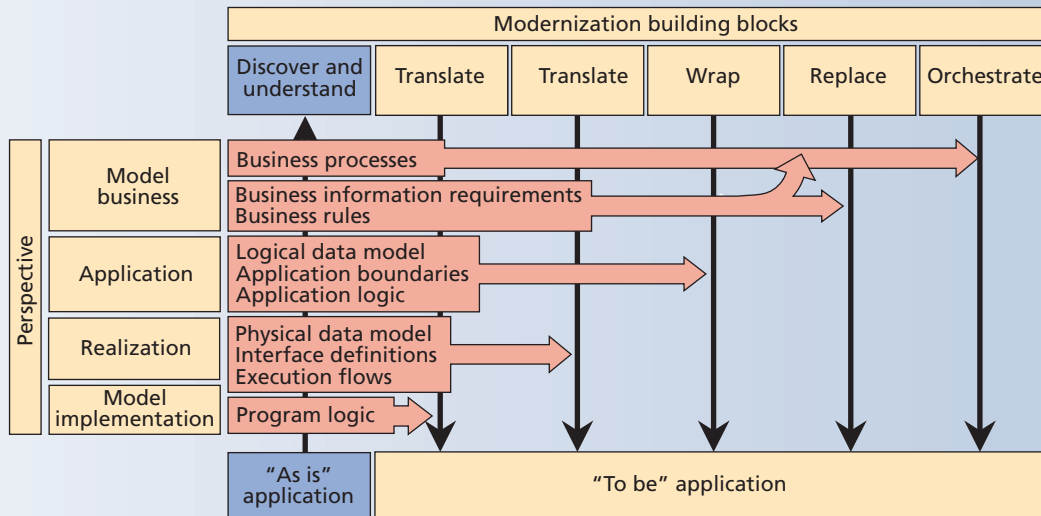
In green-field adoption, OSS is implemented either as a distro (short for distribution), as a stand-alone application, or as an integrated stack of related or integrated solutions. Distros incorporate the Linux kernel wrapped with a series of customized desktop and server maintenance applications and a load and install routine. A stand-alone solution can be any open source equivalent to commercial software (for example, a customer relationship management package), which could run either out-of-the box or customized. Because the open source code is available, the customization could include integration with other enterprise software, as well as extensions to the internal functions of the application itself.

A stack is a collection of open source applications that are interoperable and/or interconnected. It can be anything from the popular LAMP (Linux, Apache, MySQL, PHP) stack to a comprehensive business-in-a-box array of back-office applications. Several vendors conduct ongoing testing, security monitoring, patch management, and version upgrading of stacks and distros.

Brown-field modernization

Brown-field modernization is the process of understanding and evolving existing software assets within an architectural framework. Modernization can take many forms (W. Ulrich, *Legacy Systems: Transformation Strategies*, Prentice Hall, 2002) including

Figure 1. Brown-field modernization involves reengineering legacy software so that one or more qualities are significantly improved.



Achieving brown-field modernization by converting to OSS requires understanding and evolving software assets within an architectural framework. In the figure, business value increases with higher levels of abstraction. “Discover and understand” means gaining an understanding of the existing application and identifying the knowledge it contains.

- application portfolio management,
- application improvement,
- language-to-language conversion,
- platform migration,
- noninvasive application integration,
- services-oriented architecture transformation,
- data architecture migration,
- application and data architecture consolidation,
- data warehouse deployment,
- application package selection and deployment,
- reusable software assets and component reuse, and
- model-driven architecture transformation.

Unisys’s approach to brown-field modernization is to use certain building blocks in various combinations to create a target architecture. The building blocks include refactoring, translation, code wrapping, replacement, orchestration, and a sixth extraction, to be discussed later.

Refactoring—the first building block—is a process to preserve code transformation and results in modifying or cleaning up source code without changing the platform, language, or external behavior. Refactoring might be the only action taken if the code is only hard to maintain or change. However, refactoring might also be required to effectively wrap the application as a service and plug it into an orchestration.

Translating usually involves converting code into another language, frequently Cobol to Java. Automated

translation tools are available, but without subsequent refactoring, conversion might wind up going from spaghetti Cobol to spaghetti Java. Translation can also involve different platforms or data models.

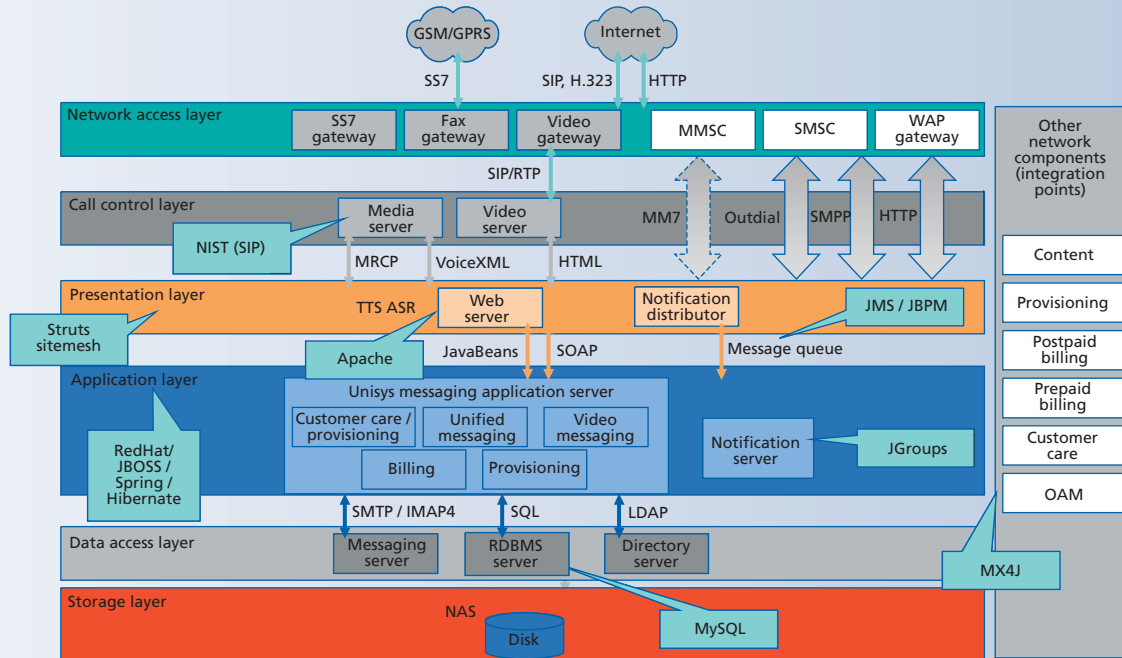
Wrapping surrounds existing applications with an interface layer to expose some or all of its functionality as a service so that it is easily reused in building new solutions. Sometimes wrapping is possible without refactoring the existing application; in other cases, much refactoring is needed.

Replacement means eventually decommissioning an application and replacing it with an entirely new solution, either designed or off the shelf. Even if the solution represents a novel vision of how to run the business, the enterprise must still know something about what the old application does.

Finally, *orchestration* requires integrating OSS-enabled legacy applications or newly created custom or packaged applications, components, or services with the help of the processes implemented using business-process enactment and monitoring services.

An enterprise deals with each application individually as part of its overall strategy for modernizing its application portfolio. As Figure 1 shows, different applications require different combinations of modernization scenarios. All six scenario categories in the figure require some understanding of the existing application, but the precise knowledge that an enterprise must discover depends on the particular scenario.

Figure 2. Architectural overview of a messaging application modernized for enhanced quality.



The application had millions of lines of legacy code.

For example, applications that need to be restructured in place might require only refactoring. Other applications that need to be wrapped as services require wrapping, but might also require refactoring and orchestration. Performing data migration to a relational database will require at least translation and refactoring. And all these applications require the enterprise to extract knowledge from the existing “as is” application—a process that becomes the sixth building block, extraction.

Each of the other five building blocks incorporates knowledge at a different abstraction level from the bottom (implementation) to the top (business). Focusing on knowledge abstraction levels is crucial to the modernization because an organization seldom has the luxury to look at each application in isolation and decide what is best for just that one piece. Rather, the view must be of the overall architecture—each piece as part of the larger plan to manage the application portfolio.

Using the building blocks: a case study

In 2006, Unisys undertook a project to modernize one of the world’s largest voice messaging applications. The strategy was to leverage 60 open source components to create a more easily maintainable and extensible and more reliable service-oriented architecture (SOA). The appli-

cation, which ran on a legacy mainframe, consisted of millions of lines of legacy code.

To achieve an SOA first, the Unisys project team transformed the application into modular subcomponents. The team then developed core software components using OSS and open standards, which they customized to interoperate with software from many other vendors. All six modernization building blocks were used to achieve the new architecture:

- The complexities of integration required refactoring code that handled processes associated with local billing systems and or local telephone networks.
- Much of the business logic that had been built over many years was translated into Java.
- Many parts of the messaging application were *wrapped* to expose them to services not running on the mainframe.
- To provide true green-field opportunities, some aspects of the mainframe application were *replaced* with a corresponding subcomponent running off the server
- As services migrated from the mainframe to the next generation architecture, we made sure that the user experience was orchestrated seamlessly.
- Finally, extraction was used throughout the process to aggregate business knowledge obtained from outside of

the messaging application. This knowledge was used as a check to ensure that the operational integrity of the messaging application was preserved with respect to the enterprise.

Figure 2 shows the overall architecture of the messaging system with some of the OSS components (in light blue). This software included Apache, MySQL database, Struts for building Servlet/JSP based Web applications, Sitemesh Web page layout framework, Java Message Service/Java Business Process Management engine for workflow management, JBOSS Java application server, Spring system for assembling Java components via configuration files, Hibernate database persistence layer, JGroups for multicast communication, and MX4J for Java management extensions. SIP is a signaling standard for telephony applications.

Customer feedback confirmed that the use of OSS dramatically reduced the time and cost of building the solution. The estimated time reduction was about several person years, and both the license and development costs decreased.

When using OSS for brown-field modernization, an organization must

- let the enterprise architecture drive strategy and apply this strategy across the entire software portfolio;
- use different modernization tactics—that is, apply a combination of the five modernization building blocks; and
- plan the modernization journey by understanding existing software so that the highest level of software quality can be achieved with the least effort.

Incorporating OSS into existing software is a viable strategy for improving certain software qualities. Both the green-field and brown-field approaches can enhance the quality of any enterprise software portfolio, although our work focused on modernization of existing software assets. ■

A Defining Difference

There is certainly room for healthy skepticism about open source software quality. One study of 200 OSS projects, for example, found that fewer than 20 percent had used associated test plans and only 40 percent used testing tools (L. Zhao and S. Elbaum, “A Survey on Quality Related Activities in Open Source,” *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 3, 2000, pp. 54-57).

This data points out a dramatic difference in the nature of product creation and testing between the proprietary and open source worlds. Proprietary applications typically have structured test plans based on stated user requirements or product plans for a known user base. Even agile development uses a methodology centered on test-driven design. Open source applications tend to rely more heavily on real-world use criteria for a user-base population where the array of features and functions might be broader than the coders envisioned (or could have predicted). A recent study concluded that the essential ingredients for quality included a large, sustainable community, available documentation, extremely modular code, rapid release cycles, code review by people outside the project community, and an appropriate community culture (M. Aberdour, “Achieving Quality in Open Source Software,” *IEEE Software*, vol. 24, no. 1, 2007, pp. 58-64).

Phillip Laplante is a professor of software engineering at Pennsylvania State University's Great Valley School of Graduate Professional Studies. Contact him at plaplante@psu.edu.

Anthony Gold is a vice president and general manager at Unisys Corp., where he created and leads the open source business unit, one of the world's largest such operations. Contact him at anthonygold@unisys.com.

Tom Costello is CEO of Upstreme Inc., where he is responsible for a variety of strategic and tactical services including open source strategies and delivery. Contact him at tcostello@upstreme.com.

For further information on this or any other computing topic, please visit our Digital Library at <http://www.computer.org/publications/dlib>.