Re-Using Open Source Software in Your Software Delivery

Karen McRitchie, Galorath Incorporated
Rick Spiewak, The MITRE Corporation

Abstract. Open source software is generally available with few restrictions (depending on license) to be reused by other developers. Reuse of software presents both potential cost and schedule savings and corresponding risks to both cost and schedule. The key defining characteristic which distinguishes between risk and reward in this scenario is the quality of the software to be re-used. Well-established metrics and best practices in software development can be applied and assessed when examining open-source software for potential re-use.

Introduction

Open source software can be effectively incorporated into larger software systems. It is important to understand the origin, quality and completeness of such software. While the reuse of software can be cost-effective, it does involve cost. These costs can be taken into account using standard measurement and estimation tools based on the completeness of the package and the structure of the source code. This will help to avoid unexpected cost and schedule problems caused by incomplete or problematic source code acquired via open source. An example using the SEER for Software [1] modeling tool is used to illustrate this.

Open Source Software Defined

The definition of Open Source software is generally attributed to the Open Source Initiative [2]. In general terms, it requires the following (abbreviated) features:

- Free redistribution – no fees or royalties
- Source code (in the preferred form for modification)
- Derived works are permitted
- Limited restrictions on distribution of modified source code
- No discrimination as to persons or field of use
- License included without requiring re-licensing: not specific to a product, not restricting other software, and not technology-specific

This means that open source software is generally available to be re-used by anyone who requires the capability which it represents, and for any purpose.

Note, however, that this definition does not include any form of warranty or support for open source software. Responsibility for meeting any and all requirements, whether technical or in the area of reliability, maintainability and availability, rests with the user of open source software. This caveat also applies to any security considerations.

Licenses for open source software vary, and need to be examined to make sure that there are no unacceptable terms and conditions relative to your intended use. Take care to ensure that these do not conflict with the need to maintain modified source code without distributing it or require contributing it back to the original project if this is relevant.

Open Source Projects

There are a number of well-known, widely used open source projects. These serve as examples of the fact that open source development can produce viable, reusable products. Key instances include:

- Operating systems based on the Linux kernel (several variants)
- The Apache web server
- The Eclipse software development environment
- The NetBeans software development environment
- The Java language
- The MySQL database
- The Git version control system
- The JUnit unit testing framework for Java

These and other tools are often used to develop other open source software, which is commonly available for download or contribution on web sites such as GitHub [3] or SourceForge [4]. The overall effect is the creation of a software ecosystem, in which many developers contribute in a synergistic fashion to related projects. While this ecosystem is particularly useful to one-off or research projects which simply need a particular function in order to reach an end objective, incorporating open source into products which need to be delivered and sustained as part of a program of record requires additional consideration.

Acquiring Open Source Software

While there is no direct cost involved in acquiring open source software beyond the time and effort involved in locating and downloading it, this doesn't make it free of cost to use. There are a number of elements that factor into the ability to re-use software. These are outlined in Elements of Reusable Software, below. Missing elements are likely to require additional work including additional software development on the part of developers reusing the software.

As pointed out by Capers Jones [5], "Reuse of code, specifications, and other material is also a two-edged sword. If the materials approach zero-defect levels and are well developed, then they offer the best ROI of any known technology. But if the reused pieces are buggy and poorly developed … software reuse has the worst negative ROI of any known technology."

This means that open source software being considered for reuse should be assessed according to the standards used for

new or reused internal development by the adopting organization. Because the original developers may not be available to answer questions or provide direct support, in some respects it is more important to ensure that the initial quality meets these standards.

Additional considerations which can improve reusability include [6]:

- An active online community providing support forums
- Availability of training from authors or third parties
- Availability of paid support from authors or third parties
- Source code licensing which is appropriate for the intended use

In addition to the above attributes of open source software, it is important to distinguish the form in which this software is acquired. Some repositories, such as NuGet [7], are oriented towards ease of use and keeping current with referenced projects. This can mean providing pre-compiled components together with their dependencies. In some cases these components represent released and supported products from major companies. These are not in the same category as open source, and don't need to be treated in the same way. In cases where pre-compiled components are actually related to open source, the selection criteria may vary. If the pre-compiled components are released products these may be the appropriate choice unless there is a good reason to modify the source.

When acquiring open source software for a program of record, an approach which provides compiled components should only be used for products from an approved supplier. In this case, they should likely be downloaded separately and configuration managed in lieu of obtaining them from the repository at build time. Source repositories such as GitHub or CodePlex [8] can be used to acquire source code. Be sure to distinguish among the various types of available components and select your acquisition methods accordingly.

Applying Open Source Software

Applying open source software in a program of record requires that the development team assess the completeness and quality of the open source software under consideration using the same standards as are required for new or internally reused development. This means that the developers need to be able to incorporate the open source software into their development as if it were part of their original work.

Understanding and Inspection

It is necessary to understand open source software as well as test it to the same standards used for internal development. This should include appropriate inspection and testing using the steps suggested by Capers Jones [9] and listed in Defect Removal, below. This can serve as an important measure of the defect potential and quality of the software.

Not all of these steps can be readily applied to software acquired as open source. On the inspection side particular attention should be paid to the code, any test cases and static analysis. Automated tools are readily available to aid in this task, providing the ability to measure elements such as:

- Conformance to best practices in coding – measured by static analysis tools, generally specific to particular language(s) or development environments. Examples include Cppcheck [10] for C and C++, FindBugsTM [11] for Java and FxCop [12] or the equivalent Microsoft Visual Studio Code Analysis [13] tool for Microsoft .NET, C and C++. These tools report on potential errors in code and identify the potential consequences. The number of potential errors can be compared to standards as enumerated [14] by Capers Jones in terms of potential defect density.

While static analysis does not completely detect all types of defects, it will provide a good starting point for judging code quality. When analyzing open source, you may find it necessary to exclude pre-determined rules in areas such as spelling and naming conventions. Analysis of code which was not originally written with these conventions in mind can produce voluminous errors which will tend to mask the potentially serious defects. Another type of error which may require manual inspection to validate is a common rule against catching general exceptions. Some static analysis tools will flag this even if the exception is then processed further. Manual inspection can distinguish cases where exceptions are ignored from those whose processing doesn't conform to the analysis tool's implementation.

- Complexity analysis [15] (most commonly Cyclomatic complexity) is measured by a variety of tools, and can highlight the potential for errors due to excessive complexity.

The common rule of thumb used as a test of excessive complexity is that when Cyclomatic complexity exceeds a range of 10 to 15 that the software routine should be refactored in order to reduce this metric [16]. Other investigators have found that the probability of a routine or module being fault-prone increases dramatically starting around a measurement of 38, and approaches a near certainty at 74 and up. Note: manual inspection can mitigate this, as (for example) lengthy "switch" statements will raise the complexity measurement. By ensuring appropriate breaks, not all high-complexity routines will require re-factoring.

- A freeware application such as SourceMonitor [17], can be used to analyze source code for "quality and quantity." This tool includes calculation of Cyclomatic complexity.
- Code reviews should be performed on open source components which display either a large number of errors flagged by static analysis or high complexity numbers.

If automated unit tests are included they should be run against the acquired code. If they are not included provision should be made for developing them on at least an as-needed basis, including allocating additional resources for this purpose. For example, if a defect is found in a particular routine it is recommended that a test be written which fails (showing the defect) followed by fixing the defect and re-running the test to show success. If a particular section of code is found to be unusually prone to defects, automated unit tests should be written to exercise the public interfaces, and used to verify that incorrect results are corrected.

As the authors have shown in a previous article [18], development of automated unit tests (AUT) in place of traditional manual unit testing does not add cost during the development

cycle. However, this model doesn't apply to AUT developed after the fact. For this reason, these tests should be developed with an eye towards appropriate return on investment. In addition to the consideration above regarding defect prone code, routines whose correct operation is deemed critical to the application should be outfitted with AUT as well. One of the unheralded benefits of this type of test is that it serves as a working example of how a developer can successfully implement the underlying functions in new code, which can be a productivity enhancement.

Each of these aspects can be taken into account in modeling the potential cost of re-using the open source software under consideration.

Security Considerations

Additional inspection steps may be needed which specifically address potential security issues, classified as weaknesses or vulnerabilities [19]. This generally requires the use of specialized tools such as HP's Fortify, klocwork Insight or Coverity Code Advisor. These are static analysis tools specifically built to inspect for potential security problems. All of these support the languages most commonly found in open source software such as C++, Java and C#.

If architecture specifications and diagrams or other information is not available with open source software it may be advisable to allocate resources to generate this by using available tools which include reverse-engineering capabilities. Examples include Enterprise Architect (Sparx Systems) and Imagix 4D (Imagix Corporation).

Cost Modeling

Code reviews should be conducted on selected code samples both for the purposes of inspection and for familiarization. Developers who intend to incorporate open source software should expect to become familiar with the code in order to effectively use, test and troubleshoot systems using the open source code.

In assessing the impact of incorporating open source software, an overall estimate of resource requirements can be created by applying a model using SEER for Software.

Identify What Needs to be Costed

Open source is no different from any other estimate in that you need to have an understanding of the scope of work involved. Traditionally for software projects, this involves sizing up the software to be built and using a parametric model or productivity factors to project cost.

Open Source Cost Modeling Checklist

- Obtain code count
- Identify build configuration assumptions
- Identify source files/modules requiring manual code review. (If your estimate is being done prior to static code analysis, assume 5%-20% will require review.)
- Review results of static code analysis to identify potentially problematic modules.
- Are unit tests built-in? If, not make sure to include these as needed in your estimate
- Identify modules requiring testing
- Include size for features that need to be added or modified to meet overall requirements
- Review assumptions on productivity drivers, including experience or lack thereof for any OSS packages used
- Review allocation into roles and activities
- Assess aggregate risk to the overall effort and schedule

Cost Modeling Example

This section has an example showing the use of OpenVPN, an open source VPN package that can be used to create connections to private networks. SEER for Software will be used to model and compute effort associated with the OSS package. SEER for Software is a commercial software estimating solution that can be used for a wide range of software development and maintenance projects, published by Galorath Incorporated [20]. The general approach will be to use the OSS package size as existing code and generate assumptions related to the review/rework and testing associated with the open source.

OpenVPN

OpenVPN is a full-featured open source SSL VPN solution that accommodates a wide range of configurations, including remote access, site-to-site VPNs, Wi-Fi security, and enterprise-scale remote access solutions. Starting with the fundamental premise that complexity is the enemy of security, OpenVPN offers a cost-effective, lightweight alternative to other VPN technologies that is targeted for the small/medium business and enterprise markets [21].

Using the USC Unified Code Count (UCC) tool [22], a full count of all source files was generated for OpenVPN, as shown in Table 1 – OpenVPN SLOC Count. The vast majority are the C++ files, at over 200K Source Lines of Code (SLOC). It is important to point out that for cost modeling purposes Logical SLOC should be used and not physical lines of code which can be considerably higher.

Language Name	Number of Files	Physical LOC	Logical SLOC
Bash	10	7349	5908
C_CPP	172	62167	38789
JavaScript	1	73	48
Perl	2	47	42
Total	185	69636	44787

Table 2 – Rework Assumptions

Parameters	Function Based Sizing	Project Monitor & Control Snapshots	Labor Category Allocation	
COMPONENT: OpenVPN (Open Source)	Least	Likely	Most	Note
Pre-exists, designed for reuse	366	492	966	
Pre-existing lines of code	44,787	44,787	44,787	
Lines to be deleted in pre-exstg	0	0	0	
Redesign required	0.00%	0.00%	0.00%	
Reimplementation required	1.03%	1.03%	4.15%	Based on 33 - 133 functions requiring manual review and potential change.
Retest required	1.60%	2.40%	3.20%	10%-20% of the features/fouctions will be use and will have to go through formal testing, including test driver development

OpenVPN Cost Modeling Assumptions

- The total SLOC size will be entered as pre-existing code.
- OpenVPN supports multiple platforms and configuration, however only Windows will be supported for this implementation.
- OpenVPN will have to be validated for use and will undergo static code analysis to identify high risk modules that will require manual code review. For this example, 33 of 2020 functions had an indicated risk of High or Very High (based on Cyclomatic complexity evaluation). An additional 100 functions have indicated a medium risk. Based on this, between 33 and 133 functions (of a total 2020, or 1.63% - 6. 58%) will require manual code review and automated unit test insertion.
- In addition, static analysis was run against the C and C++ files using Cppcheck. 3 errors and 7 warnings were found. This is a low number by any measure, and indicates that the manual inspections and code reviews conducted by the developers have been reasonably effective [23].
- The application will make use of 10% - 20% of the overall features and functions and require regression test.

Because OpenVPN has not been used by the developing organization, test drivers will need to be developed.

Sizing the Rework

The above assumptions are mapped into SEER for Software inputs with the aid of the SEER for Software Rework Percentages workbook to derive rework percentages that reflect the review and test effort associated with the OSS.

Reimplementation captures the effort associated with non-automated code reviews and unit test efforts. The retest effort captures effort for running regression tests as well as developing test drivers needed for such tests. It is assumed that the overall design of the OSS will remain intact, thus no redesign. The input assumptions are expressed as a range. This is especially important when static code analysis has not been done and the extent of the review is unknown. The computed rework assumptions into the size inputs are reflected in Table 2 – Rework Assumptions:

The overall estimate is coupled with any custom or non-OSS software that needs to be developed. In this case, a component for new code for secure components was added to the estimate.

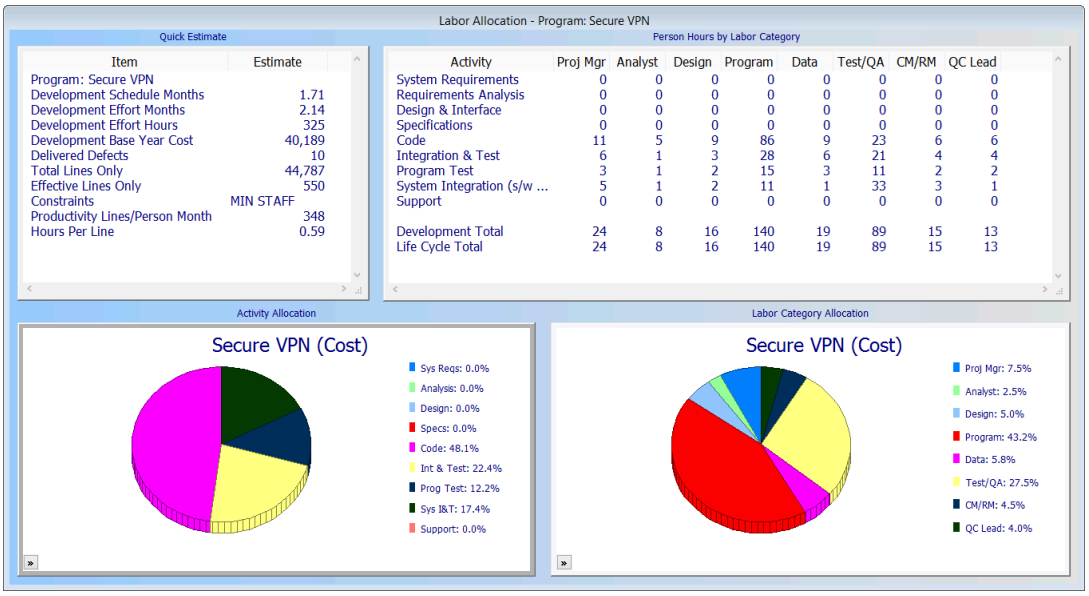

Figure 1 – Overall Estimate

Other Cost Driver Considerations

The default parameter settings are a good start for most input drivers. However some were tuned to reflect the specific situations. The following adjustments were made to default parameter settings:

- With respect to adopting the open source software, no special requirements effort is needed, so that effort component is turned off. (Requirements Definition Formality = Vlo)
- No rehosting to alternate platforms is required, the OSS packages provide multi-platform support. (Rehost from Development to Target = Nom)
- The OpenVPN has no special UI (Special Display Requirements = Nom)
- OpenVPN has added security requirements commensurate of Common Criteria EAL 1 (Security Requirements = Nom+)

The overall effort to adopt this open source packages comes to 325 hours, with a schedule of around two months (assuming effort occurs in parallel). Looking at the allocation into labor roles and activities, it is clear that this is effort is all in the code and testing phases.

While the above estimate provides a good overall planning figure, evaluation of the risks in terms of time and effort should be taken into account. Running a Monte Carlo simulation of the range of possible outcomes, it becomes clear that even at a most likely scenario, more hours and schedule should be considered, as seen in Figure 2 – Monte Carlo Simulation. In planning for contingency, using a higher confidence level such as the 70% or 80% is often prudent.

This cost modeling example focuses on the effort to adopt an open source package and should be considered part of an overall system effort. Other efforts, such as training, maintenance and deployment should also be part of a system total cost of ownership analysis.

Elements of Reusable Software

For software to be reusable by other developers, whether for modification or incorporation into larger systems, there are elements that should be included in order to reduce the cost associated with the adoption of the software. These include the following:

- Programmer's reference manual with examples for any components with public interfaces.
- Interface definitions
- List of all software components with the following information:

Purpose and function

Interfaces provided

Language/version for each module

- Complete source code:

Interface Definition Language files

Web Services Description Language files

Other source code as projects/solutions suitable for compilation/build in the Integrated Development Environment (IDE) or build/make program appropriate to the source type.

XML Schema and Schematron files

Database schema definitions as applicable

- Enterprise Architect or other Unified Modeling Language (UML) source where available
- Use cases (text and diagrams) – diagrams are included in UML design files in many cases
- Class diagrams where applicable
- Dependency diagrams if created or available
- Complete list of any third-party components with version numbers
- List of commercial and public domain software required to build the software, and the recommended order of installing these on the build machine
- Distribution package and source code of public domain components used in the software

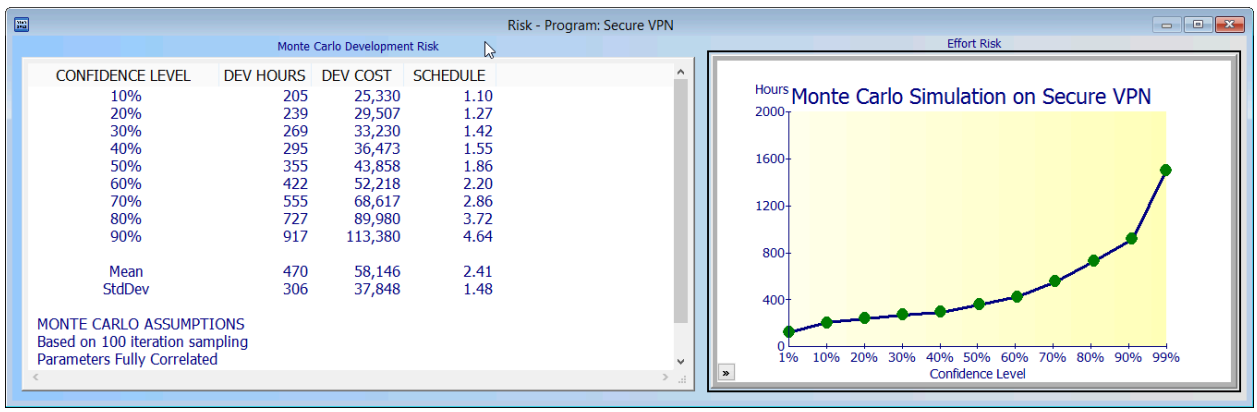

Figure 2 – Monte Carlo Simulation

- Contact information for any outside dependencies
- Build procedures, including documentation for building all components from source code

Detailed instruction on setting any necessary environment variables on the build machine, e.g., IDE options, system path.

Build procedures must be executable in a standalone development environment without requiring access to the developer's configuration management or source code control system.

No pre-installation of pre-built components included in the delivered software can be required other than any build order dependencies for components built from source as part of the build procedure.

- Test procedures – including any automated unit tests with source code, test scripts
- Installable versions of executable code, with and without debug information/symbols.
- Source for installation scripts and procedures.

While the above requirements can be specified for new or contracted development, open source software won't always include them. To the extent that these are missing or incomplete, the organization reusing the software may incur additional development or sustainment costs as a result. This must be taken into account, and decisions made as to whether to mitigate the risk associated with missing elements by assigning additional work ahead of time or to accept and account for the associated risk.

Pros and Cons of Using Open Source Software

Very often, the topic of pros and cons for open source are comparisons to using COTS software for a particular purpose. In this case, we need to consider the pros and cons in comparison to either new or continued custom software development.

Pros

- Acquisition cost: Initial acquisition cost is usually zero for source code.

Some software source code is provided free, but there is a cost for documentation and training. This is generally much lower than the cost of development.

- Maintenance: Open source software, because it is used by many others, is updated as needed. These updates are then available for incorporation. Changes you make may be adopted and incorporated into future versions. You may also benefit from changes made by others.
- Testing: Because there are many developers and users, there is ongoing testing. By participating in user forums it is possible to be alerted to potential problems ahead of encountering them in use.
- Security: Vulnerabilities are often reported to the development community, and fixes may be available in a timely fashion.

Cons

- Obsolescence: While this is a known issue with COTS software, it also exists with open source. If the system under development has a longer development and sustainment lifecycle than the open source software, it may be necessary to make custom modifications which then make it more difficult to move to a later version. Careful use of configuration management and source code control systems and good development practices can mitigate this by making it easier to identify and isolate changes. Whenever changes are contemplated, it should be determined whether a newer version should be adopted which might incorporate the needed capability, or whether the proposed changes might qualify to be submitted back to the community.
- Maintenance: While there may be many developers and users, there is not always a single point of contact for defect reporting and fixing. It may fall to the organization using the software to identify a defect, provide a potential fix, and win acceptance from the development community for implementation in a newer version.
- Quality: Coding standards may not meet those of the overall project. Static analysis may not have been applied, and peer review and inspections may not have occurred.
- Testing: The degree and depth of testing may not meet the quality standards of the adopting organization. Provision must be made for incorporating additional testing and inspection steps (see Defect Removal, below).
- Security: Secure design and security testing are not always high priorities in open source development. Depending on the security level required, standard static analysis tools may need to be supplemented with security-specific tools to examine the software. This is an additional reason to ensure that source code is used, rather than compiled versions.
- Licensing: This can vary. Some licenses require returning changes to the community. Examples of licenses applicable to cases where a requirement to post changes back is not acceptable include the Microsoft Public License, MIT license and Apache 2 License.

Defect Removal

According to Capers Jones, as cited above, combining the following recommended methods "will achieve cumulative defect removal efficiency levels in excess of 95 percent for every software project and can achieve 99 percent for some projects [24]." These are divided into categories of Pre-test and Testing. Pre-test methods consist of inspection and analysis steps. Testing is categorized according to the stage of development:

Pre-test Defect Removal

- Requirements inspection
- Architecture inspection
- Design inspection
- Code inspection
- Test case inspection
- Automated static analysis

Testing Defect Removal

- Subroutine test
- Unit test
- New function test
- Security test

- Performance test
- Usability test
- System test
- Acceptance or beta test

Open source software should be examined with respect to the needs of the application, and appropriate inspection and test steps should be performed as part of the process of incorporating this software into a resulting system. To the extent that documents or automated tests exist which represent any of these steps, the required effort to ensure appropriate quality can be reduced.

Summary

Open source software can be effectively incorporated into larger software systems. However, it is important to understand the origin, quality and completeness of such software.

While the reuse of software can be cost-effective, it does involve cost. This cost can be estimated using standard measurement tools and commercial cost estimation tools based on the completeness of the package and the structure of the source code. This will help to avoid unexpected cost and schedule problems caused by incomplete or problematic source code acquired via open source. The potential need to maintain compatibility with the original source should be taken into account as both a possible cost and a possible cost savings.

Available tools can be used to assess the quality of open source software in order to determine the likely applicability of the software to a particular system, and allocate sufficient resources to apply it effectively.

Disclaimer:

REFERENCES

1. <http://galorath.com/products/software/SEER-Software-Cost-Estimation>
2. <http://opensource.org/osd>
3. <https://github.com/explore>
4. <http://sourceforge.net/>
5. Jones, Capers. Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies. McGraw-Hill, 2010. Chapter 2, p. 101
6. Discussion with David Smiley, OpenSource Connections
7. <https://www.nuget.org/>
8. <http://www.codeplex.com/>
9. Ibid, Chapter 9, pp. 618-619
10. <http://cppcheck.sourceforge.net/>
11. <http://findbugs.sourceforge.net/>
12. <http://www.microsoft.com/en-us/download/details.aspx?id=6544>
13. <https://msdn.microsoft.com/en-us/library/dd264897(v=vs.120).aspx>
14. Ibid, Table 9-8, p. 982
15. McCabe, Thomas J. "A Complexity Measure", IEEE Transactions On Software Engineering, Vol. SE-2, No.4, December 1976
16. <http://en.wikipedia.org/wiki/Cyclomatic_complexity>
17. Campwood Software, LLC - <http://www.campwoodsw.com/>
18. <http://www.crosstalkonline.org/storage/issue-archives/2008/200812/200812-Spiewak.pdf>
19. See the Common Vulnerabilities and Exposures list at <http://cve.mitre.org/>
20. <www.galorath.com>
21. <http://openvpn.net/index.php/open-source/245-community-open-source-software-overview.html>
22. <http://sunset.usc.edu/ucc_wp/>
23. Conversation with Samuli Seppänen, Community Manager, OpenVPN Technologies, Inc
24. Jones, Capers. "Methods Needed to Achieve > 99% Defect Removal Efficiency (DRE) for Software", April 9, 2014

ABOUT THE AUTHORS

Karen McRitchie is Vice President of Development at Galorath Incorporated. Ms. McRitchie is responsible for the design, development, implementation and validation of the parametric estimation relationships found in the SEER™ estimation products published by Galorath Incorporated. She has worked in all domains of cost estimation, but much of her focus has been on the software/application and information technology domains. Ms. McRitchie has participated in numerous estimation, data collection, and calibration efforts and has trained hundreds of cost analysts in the use, application, and calibration of SEER-SEM™, SEER-H™ and SEER-IT™. She has been active in the International Cost Estimating and Analysis Association (ICEAA) for many years was honored by ISPA in 2002 with the Parametrician of the Year award.

Rick Spiewak is a Lead Software Systems Engineer at The MITRE Corporation. He works as part of the Battle Management / Command and Control & Surveillance Group, concentrating on Mission Planning systems. Rick has been focusing on the software quality improvement process, and has spoken on this topic at a number of conferences as well as publishing in CrossTalk and MSDN magazines. He has been in the computer software industry for over 45 years. His experience includes developing software and managing software development for data acquisition systems, transaction processing, and data communications and networking. Rick has also taught computer architecture and data communications and networking at the graduate level. He studied quality management at Philip Crosby Associates.