An Open Source Approach to Developing Software in a Small Organization

Ken Martin and Bill Hoffman, *Kitware*

Kitware's software development approach borrows techniques from agile development and Extreme Programming and emphasizes long-term, ongoing projects.

How does a small group of software developers exploit the software engineering practices that large software organizations typically use? Small groups certainly need all the benefits that such practices provide but often lack the infrastructure and economies of scale that larger groups can leverage. A small software group typically doesn't have a dedicated QA team or testing farm, or possibly even a dedicated system administrator. The SEI report on process improvement for small settings provides a good introduction to these issues, statistics, and relevant lessons learned.[1]

Over the past eight years, Kitware has grown from having two developers to having more than 30. We produce a mixture of commercial closed-source and open source products. Some products are internal to Kitware, but others have collaborators located around the world. Likewise, some projects are small, with only one or two developers, but others have 30 or more. So, any development approach we use must have low enough overhead that we can use it with smaller projects but also be able to handle larger projects consisting of thousands of source files. Given all these requirements, plus the need to cross-develop on Windows, Linux, and Mac OSX, it's clear that any viable solution must be very flexible.

Our software development approach addresses some of these concerns. We've been refining our approach while working on projects such as the Visualization Toolkit[2] and the Insight Segmentation and Registration Toolkit (ITK),[3] widely used open source toolkits with thousands of source files and many distributed developers. We've also applied these techniques to ParaView, a scientific visualization application that's open source but has a smaller development community, and VolView, a closed-source, internal, medical volume-rendering application. This article discusses the specific tools and processes we've used, the costs and benefits, and the lessons we've learned.

Origins of our software process

In 1999, the US National Library of Medicine (NLM) at the National Institutes of Health (NIH) awarded a three-year contract to develop an open source registration and

0740-7459/07/$20.00 © 2007 IEEE

segmentation toolkit, which eventually came to be known as ITK.[3] The initial development team consisted of three commercial partners (GE Corporate R&D, Kitware, and MathSoft) and three academic partners (University of North Carolina, University of Tennessee, and University of Pennsylvania). This large, geographically separated group was tasked with a difficult software engineering problem: to develop a reusable cross-platform open source toolkit in C++ that would contain the state of the art in medical imaging algorithms and be useful for both research and development.

At the project's start, we surveyed the available tools to determine what we needed to develop and what we could use. We also studied recent software development trends. The emerging Extreme Programming[4] and test-driven development[5] methodologies provided many good ideas and helped us to develop our process. XP's testability requirement was of particular interest. ITK would have many developers and many platforms to support. For development to move forward at the correct pace, the developers would have to rely on the system working almost continuously. XP principles also encouraged the use of code standards and collective ownership. Six-sigma quality efforts led the team to use metrics such as daily code coverage analysis during testing. In many ways, our approach resembles the Agile Unified Process;[6] both are lightweight, iterative, incremental, and tool independent.

The process in detail

We organized the process into five major parts: communication and documentation, revision control, build management, testing, and release creation. The motivation behind this organization is looking at the software process from a steady-state perspective, and it's particularly suited for long-term projects. Communication, for example, is always happening with new requirements, design discussions, and end-user feedback, all of which occur concurrently on an ongoing basis.

We designed the process to be lightweight and allow for rapid development, with a requirement that the majority of the tools used had to be open source. For communication and documentation, the process initially used the Mailman mailing list software, Doxygen for automatic documentation of source code, and phpBugTracker for issue tracking (available at http://sourceforge.net). In recent years, we've added the Wiki,[7] another powerful communication tool. For revision control, we picked the Concurrent Versions System (CVS)[8] because of its easy-to-use, nonlocking client-server model. For the build process, at the start of the ITK project, no tool existed that could build complicated C++ projects on all of the supported platforms, so we developed a tool called CMake.[9] For automatic testing of software, the ITK team decided to develop another tool, eventually called Dart (http://svn.na-mic.org:8000/svn/Dart/trunk/Dart.pdf). Over the past eight years, we've also developed a process for using all these tools together to generate regular releases as well, and we've recently begun work on creating a new tool called CPack to aid in the release process. The process has proven to be flexible, and it adapts well to new tools and technologies as they appear.

Communication and documentation

Good communication and documentation are arguably the most important aspects of a software process. We use several techniques to encourage constructive communication and documentation during software development and maintenance.

Mailing lists. Mailing lists have proven to be an indispensable form of communication between software developers and users. The Mailman list manager is a simple, easy-to-manage, Web-based tool for creating and maintaining mailing lists. Each project normally has a developers' list and a users' list. The developers' list contains detailed discussions about design issues. The users' list provides a place for end users to pose questions about how to use the software. Users often help each other, although developers commonly subscribe to the users' list as well to counter any bad advice that might appear. In a project's early days (when the software has no users), this list isn't required. When the project reaches a certain maturity level, it makes sense to create a separate list for users, which Mailman lets you do easily.

Mailing lists are also beneficial because they result in searchable archives, which provide a source of documentation and can contain the most current information about how to use a project. So, email lists are dynamic communication channels that can react quickly as systems evolve in response to changes such as new operating system or compiler releases.

Good communication and documentation are arguably the most important aspects of a software process.

If programmers are developing documentation while writing code, it's far more likely that the documentation will stay current.

Wiki. An excellent adjunct to mailing lists is the Wiki, a Web page that the software community can edit quickly. This allows for the rapid documentation of discussions on a mailing list. For example, consider a situation in which the release of a new compiler results in a software compile failure unless a special compiler flag is added to the build process. The first lucky user to run into the problem would post it to the users' mailing list. Somebody in the community would then identify a workaround, post the fix to the list, and ask the original user to verify the solution. Once the user has verified the solution, somebody would update the Wiki with the fix, which then becomes part of the software documentation. We use MediaWiki, which is freely available as open source software.

Doxygen. As with XP, we've incorporated Literate Programming ideas into Kitware's development process. As Wayne Sewall explains, "In general, literate programs combine source and documentation in a single file. Literate programming tools then parse the file to produce either readable documentation or compilable source."[10]

The Doxygen tool employs a simple comment markup language in C++ or C source code and generates extensive source code documentation. Because the tool automatically generates the documentation from the source, it can be automatically updated each night to match the most recent copy of the code in the revision control system. Programmers generally don't like to write documentation, but they do like to write code. So, if they're developing documentation while writing the code, it's far more likely that the documentation will stay current and follow the code.

Issue tracker. Another powerful tool in the software process is the bug or issue tracker. An issue tracker lets developers and users report software issues and provides a place to store feature requests so they're not forgotten. We chose the Web-based phpBugTracker system because it's easy to install, has a clean user interface, and can easily handle multiple projects. As developers and users create issues in the system, the issues are given a unique ID and can be assigned a severity and priority. The issue tracker's administrator can give developers extra privileges, so that they can assign status and resolutions to issues. The system also lets you attach files to issues, providing a way to give reproducible test cases and even source patches. The issue's initiator and the developer assigned to it are notified via email when the issue's status changes. This allows for dialogue between the reporter and the developer.

However, having a good issue tracker isn't the end of the story. For the tool to remain useful, someone must manage it, and the team must get together periodically and "triage" the issues—that is, adjust their priorities and reassign them to the appropriate developers or categories. Triage normally occurs before a release and involves getting the entire team together for some form of real-time communication (either in the same room, or via a telephone or video conference). Without this attention, issue trackers can quickly become a list of things that never get done.

Revision control

All software projects, large and small, should use some sort of revision control. A revision control system lets developers track changes in the software over time. Many tools have been created to manage file versions. Most revision control software works by keeping a central repository for files under revision control. The central repository usually stores the files as a history of changes or deltas. Each time a developer changes a file, he or she can "commit" the change to the repository, making it available to other developers. This can be problematic when two people change a file at the same time. This problem has two basic solutions:

- limit access to a single developer, allowing changes to a file only after the developer obtains permission to modify it from a central server, or
- allow concurrent edits of the same file and provide a way to merge the changes automatically, marking conflicting changes if they occur.

In our software process, the concurrent edit mode works best. It allows flexibility in location, so developers can make edits while on a plane or when working from home. The most popular tool for this type of revision control is CVS, which is built on a client-server architecture. The server stores the files in the project, and the clients can connect to the server and check out copies of the software. The client can request any version of a file in the system. CVS can also tag and branch files. A tag is a symbolic name for a group of files that provides a project snapshot at a given point in time. Branches let multiple paths of development occur concurrently and independently. Changes committed to one branch don't affect the other branches. One special branch, which we call the "main tree" or trunk, is the default development path checked out if none is specified. A newer tool called Subversion has a few advantages over CVS and can be used in place of it. Notably, Subversion provides http/https server mode and lets you move and rename directories, and branching and tagging are computationally cheap.

When it's time to create a release for a software project, one developer is assigned to be the release manager. He or she tells all the developers to get all final edits into the repository, which is then locked down, with only the release manager having write access. The release manager then tests the system and, when the desired features are present, creates a branch for the release. The release manager then applies fixes only to the branch. CVS can be set up to block other developers from committing changes onto a branch. The rest of the developers can then move back to the main tree and start developing the next version of the software.

Build management

An important but often overlooked part of any software project is managing the build process. The build process takes a group of source code files and transforms them into end-user applications and libraries. A large software package frequently employs a suite of tools, consisting of compilers, custom generators, linkers, and archiving tools, for this purpose. Coordinating all of these tools consistently from computer to computer and developer to developer can be difficult and time-consuming. These tools and the way that they're used vary significantly from platform to platform, so managing the build process across Windows, Unix, and Macintoshes is difficult.

Before CMake, Kitware developers used autoconf on Unix platforms and a Visualization Toolkit-specific nmake makefile generator on Windows. The approach consisted of handcrafted makefiles including generated lists of source files. The process was error prone and restrictive, with only a few developers knowing how to make changes to the build.

CMake takes as input simple script files in the CMake language. CMake uses the input files to perform system introspection and describe how to construct executables, libraries, and custom targets. Unlike many cross-platform build tools, CMake is designed to be used in conjunction with the native build environment by generating build files for native build tools. This lets developers use the tools with which they're most proficient. This is important for small companies because training people to use standard tools can be costly and might generate resentment among developers who are forced to use a corporate-mandated build environment. Larger organizations might have entire teams devoted to maintaining the build process, but by automating much of the work involved with the build process, CMake lets smaller teams develop portable software. It also lets developers generate an executable in just a few lines of code that will compile on multiple platforms.

CMake has been in active development since 1999 and has grown into a powerful general-purpose tool with a growing user base. Recently, the K Desktop Environment community selected CMake to replace autoconf and Scons (KDE is one of the world's largest open source projects). CMake is now a mature, fully featured build tool that allows for full customization of the build process and can replace the dedicated build team found in many larger organizations. The CMake language's simple syntax lets any developer easily create the executables and libraries required by the project being developed.

By automating much of the work involved with the build process, the CMake build tool lets smaller teams develop portable software.

Testing

Testing is a key tool for producing and maintaining robust, valid software.

Types of tests. The tests for a software package can take many forms. At the most basic level are smoke tests, such as those that simply verify that the software compiles. Although this might seem simple, with the wide variety of platforms and configurations available, smoke tests catch more problems than any other type of test. Another form of smoke test is one that verifies that a test runs without crashing. This can be handy when the developer doesn't want to spend time creating complex tests but is willing to run some simple ones. These simple tests are often small example programs. Running them verifies not only that the build was

Although smoke tests might seem simple, they catch more problems than any other type of test.

successful, but that any required shared libraries can be loaded (for projects that use them) and that at least some of the code can be executed without crashing.

Moving beyond basic smoke tests leads to more specific tests such as regression, black-box, and white-box testing. When a regression test fails, a quick look at recent code changes can usually identify the culprit. Unfortunately, regression tests typically require more effort to create than other tests.

The final type of testing we discuss is software standard compliance testing. Whereas the other test types focus on determining if the code works properly, compliance testing tries to determine if the code adheres to the project's coding standards. For example, such testing could check to verify that all classes have implemented some key method or that all functions have a common prefix. The options for this type of test are limitless, and you can perform such testing in several ways.

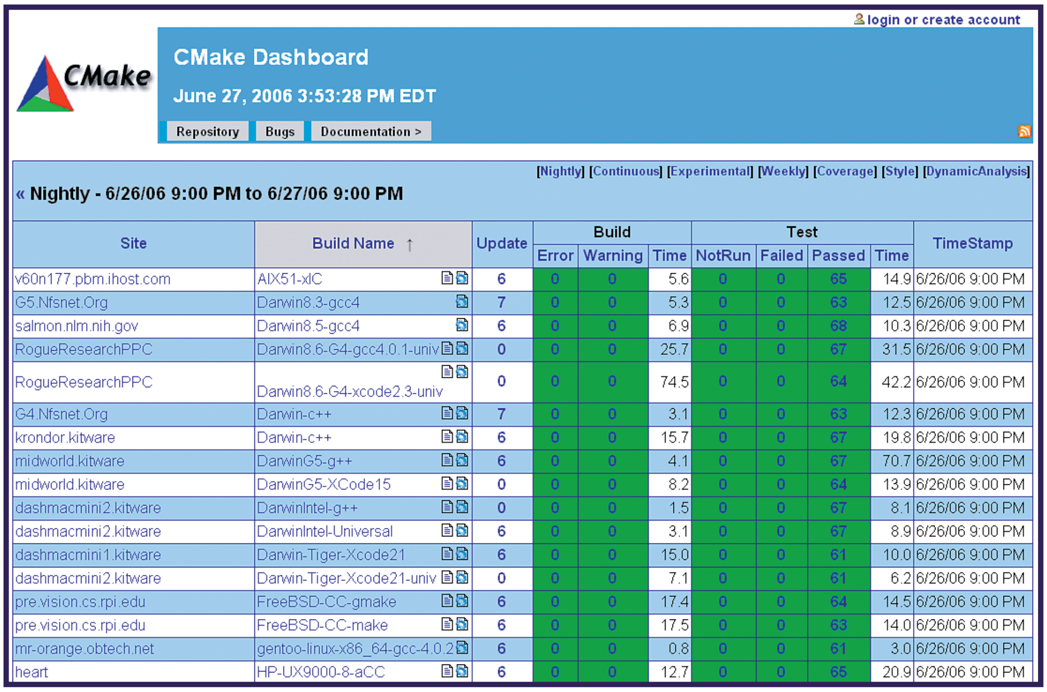
Our approach to testing. We provide a simple framework in which to add tests to projects and an easy mechanism to run those tests and view their results. For small or short-lived projects, testing is often limited to build and smoke tests. For larger, longer projects, we put more effort into testing and create specific regression tests to exercise key capabilities.

We leverage numerous tools to perform testing. The first is CTest, a command line program that Kitware develops and bundles with CMake. CTest reads in test description files typically generated by CMake and lets the user run all or some of the tests defined for a project. A typical development sequence is to modify the source code, compile, run CTest to verify that existing tests still pass, and then commit the changes to revision control. For projects with many tests that might take hours to run, we typically use a subset of tests selected by pattern matching or by using a specified stride to skip tests (for example, execute every 10th test). The goal is to provide some basic level of testing before committing the code without requiring the developer to wait for more than a few minutes. The automated testing tools do more exhaustive testing.

Our automated testing approach runs CTest on the client but, instead of reporting results on the command line, it submits testing results to Dart, an open source, Web-based test reporting tool developed at the GE Global Research Center. CTest is typically run at night by cron or the task scheduler and does a complete checkout, build, and test cycle. Typically, a single client tests multiple projects or revision branches during the night and submits multiple entries to Dart. Dart, in turn, runs on a server and accepts results from many clients. A single server might handle many software projects and provides a quick way to see if any build or testing issues exist on the test platforms. Figure 1 shows a Dart dashboard for CMake. Each row summarizes testing results for a single test client. Web links let developers drill down to see the specific text of any build, including warnings or errors, as well as the output of any tests. Typically, clients submit coordinated dashboard entries (generally running at night) based on the same source code revision, so you can compare one client's results against another's. Dart accepts entries continuously, and most larger projects use this feature to continuously report build and test status for a subset of testing clients as the source code changes throughout the day. This lets you find and correct cross-platform issues, often on the same day they're introduced.

In addition to reporting build and test results, Dart accepts and reports code-coverage and dynamic-analysis results. Code coverage provides a quick metric of how much of the code the testing process is exercising. You can set this up easily on a Linux test client by changing the compile flags for one of that client's nightly runs (for example, using gcov). CTest will automatically discover the coverage files, analyze them, and submit them to Dart, where developers can view them and drill down into them, line by line if they want to. For dynamic analyses, we typically use the free Linux tool valgrind. Again, this is easy to set up, typically requiring some minor changes to the compile flags and pointing CTest to valgrind's installation site. Dynamic analysis of the tests reports array-bound write errors, uninitialized memory reads, and so on. Purify and Bounds Checker are other options for performing dynamic analysis.

Using Dart, CTest, CMake, valgrind, gcc, and gcov, configuring a mature test process is fairly quick—you can typically do it in a day once you're familiar with the process. The challenge then becomes access to testing platforms (that is, a combination of hardware, operating system, and compiler). For open source projects that will be compiled by many people, a wide



CMake Dashboard

June 27, 2006 3:53:28 PM EDT

[Nightly] [Continuous] [Experimental] [Weekly] [Coverage] [Style] [DynamicAnalysis]

« Nightly - 6/26/06 9:00 PM to 6/27/06 9:00 PM

Site	Build Name ↑	Update	Build			Test				TimeStamp
			Error	Warning	Time	NotRun	Failed	Passed	Time	
v60n177.pbm.ihost.com	AIX51-xlC	6	0	0	5.6	0	0	65	14.9	6/26/06 9:00 PM
G5.Nfsnet.Org	Darwin8.3-gcc4	7	0	0	5.3	0	0	63	12.5	6/26/06 9:00 PM
salmon.nlm.nih.gov	Darwin8.5-gcc4	6	0	0	6.9	0	0	68	10.3	6/26/06 9:00 PM
RogueResearchPPC	Darwin8.6-G4-gcc4.0.1-univ	0	0	0	25.7	0	0	67	31.5	6/26/06 9:00 PM
RogueResearchPPC	Darwin8.6-G4-xcode2.3-univ	0	0	0	74.5	0	0	64	42.2	6/26/06 9:00 PM
G4.Nfsnet.Org	Darwin-c++	7	0	0	3.1	0	0	63	12.3	6/26/06 9:00 PM
krondor.kitware	Darwin-c++	6	0	0	15.7	0	0	67	19.8	6/26/06 9:00 PM
midworld.kitware	DarwinG5-g++	6	0	0	4.1	0	0	67	70.7	6/26/06 9:00 PM
midworld.kitware	DarwinG5-XCode15	0	0	0	8.2	0	0	64	13.9	6/26/06 9:00 PM
dashmacmini2.kitware	DarwinIntel-g++	0	0	0	1.5	0	0	67	8.1	6/26/06 9:00 PM
dashmacmini2.kitware	DarwinIntel-Universal	6	0	0	3.1	0	0	67	8.9	6/26/06 9:00 PM
dashmacmini1.kitware	Darwin-Tiger-Xcode21	6	0	0	15.0	0	0	61	10.0	6/26/06 9:00 PM
dashmacmini2.kitware	Darwin-Tiger-Xcode21-univ	0	0	0	7.1	0	0	61	6.2	6/26/06 9:00 PM
pre.vision.cs.rpi.edu	FreeBSD-CC-gmake	6	0	0	17.4	0	0	64	14.5	6/26/06 9:00 PM
pre.vision.cs.rpi.edu	FreeBSD-CC-make	6	0	0	17.5	0	0	63	14.0	6/26/06 9:00 PM
mr-orange.obtech.net	gentoo-linux-x86_64-gcc-4.0.2	6	0	0	0.8	0	0	61	3.0	6/26/06 9:00 PM
heart	HP-UX9000-8-aCC	6	0	0	12.7	0	0	65	20.9	6/26/06 9:00 PM

Figure 1. Part of a Dart dashboard for CMake.

variety of test platforms is mandatory. Even with closed source projects, it's good to have access to many platforms because, for example, any one compiler might not identify problems with the source code. For a small group, obtaining these resources can be difficult and cost prohibitive. At Kitware, we have about 30 dedicated test machines and 30 developers. Although this is expensive, you can leverage some cost-cutting measures. On Microsoft Windows clients, you can use an MSDN subscription to provide the operating system and compilers for your dedicated test machines at a greatly reduced price. Likewise, for Linux, the key software tools that we use are all free. Some vendors, such as Intel, provide discounted lease arrangements for testing and development hardware. We also use vendor-provided test machine access from IBM and HP for testing coverage on some AIX and HPUX platforms that we don't have in house.

Release process

The process for managing a release tree involves checking out both a main tree and a branch version of the software on the same machine. The release manager then periodically merges patches from the branch to the trunk (you can use tools such as WinMerge to perform merging). Developers send email to the manager when they want changes moved to the branch. At some point, the branch is tagged again and a new (incremental) release is made.

Having a stable branch of the source code to work from is only part of the problem; we must then package the software for release. Two types of releases exist: source and binary. We typically release to three major platforms: Windows, Unix, and Mac. Each has a different way of packaging sources and binaries. On Windows, we use a tool called NSIS (Nullsoft Scriptable Install System) to create Windows installers; on Unix, we use compressed tar files; and on Mac, we use a DMG disk installer.

This process is similar on the various platforms in that they all create a binary build of the software and then package it for distribution. To leverage this commonality, Kitware is developing a new tool called CPack, which abstracts out the creation of the various types of binary releases.

Costs and lessons learned

Having worked with this software development process for eight years, we've learned numerous lessons and reckoned with many costs. It might seem obvious, but one of our clearest findings was that a project needs a quality champion or many of these processes will break down. Although we automate wherever we can, in the end, someone must care about the state of the software for this to work. Someone must look at the testing dashboard and investigate or delegate any problems that show up.

We've found that, even for small projects,

Although we automate wherever we can, in the end, someone must care about the state of the software for this process to work.

testing quickly pays for itself. For us, automated testing identifies roughly four issues for every user-reported issue. Our automated testing has shown that after an issue is fixed, it's often reintroduced in later code changes, and automated testing catches this. In the area of testing, our mantra is, "If it isn't tested, it doesn't work." Even for experienced developers, this is often true. Daily testing also reduces the cost of fixing issues because identifying a problem's source is much easier when you're considering only a single day's changes. On the few occasions when we deferred testing until release, developers had to inspect many months' worth of changes to find an issue's cause.

We've found that automation is key to making processes work within a small organization. Everyone is busy, and software engineering practices are often one of the first places developers cut corners. Automating common coding standards through editor modes and enforcing them by automated commit checks takes much of that effort off the developers. The same applies to testing. If creating, running, and examining tests isn't easy for developers to do, they simply won't do it. Using Dart and CTest lets developers see the state of the software quickly by visiting a Web page; they can then perform a quick drill-down to identify any problems.

Acquiring and maintaining a large test farm is also challenging for small organizations. You can use vendors' test farms to some extent, although sometimes the contractual agreements to obtain access aren't worth it. Open source projects can leverage external developers' resources to expand the suite of testing platforms. Surprisingly, we've found that the main costs of testing farms are the HVAC, electricity, and labor, not the price of the hardware and software.

Although we've developed in-house tools for our software process, this generally isn't an option for small groups. CMake, CTest, and CPack represent more than six years of development, and without supporting funding from NLM, NIH, NSF (National Science Foundation), NAMIC (National Alliance for Medical Image Computing), ASCI (Accelerated Strategic Computing Initiative), and DoE (Department of Energy) as well as contributions and patches from the open source community, it wouldn't have been possible. For small software organizations, typically the only viable option is to leverage larger open source or publicly funded efforts. However, because most of our work in the software process area has been open source, other small groups could certainly exploit our process as described with modest initial costs.

The Wiki has proven to be a powerful training tool for new employees. Prior to the advent of Wikis, new employees often asked certain expert employees how to perform tasks. (For example, one person might be an expert in creating and maintaining CVS branches.) To avoid interruptions, those experts can put their knowledge into the company Wiki for anyone to see. Employees maintain the Wiki for selfish reasons, almost guaranteeing that it will be useful.

We're always looking to improve and automate our software process. Our next plan is to create tools that can quickly configure the entire development process for a specific project. Currently, if the project duration is short, it's often not cost-effective to set up a testing dashboard, Wiki, and mailing list because configuring a project requires setup time from the system administrator. We plan to make the setup a Web-based, automatic process that lets even the smallest project exploit the software process from inception.

Although a small organization might not have the resources a large company has, it can reap the benefits of many software engineering practices. We've seen a much greater degree of cross-platform compatibility by using these techniques. Many of the tools we've mentioned can be leveraged when setting up documented procedures for FDA compliance. For example, issue tracking and revision control for both code and communications are two key aspects of policies typically put in place for FDA QSR 820 and ISO 13485 compliance. With the variety of open source and closed-source tools available, a small organization can build high-quality robust software and save time in the process.

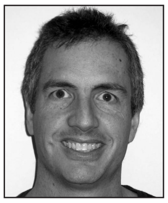
References

1. S. Garcia, C. Graettinger, and K. Kost, eds., *Proc. 1st Int'l Research Workshop for Process Improvement in Small Settings,* tech. report CMU/SEI-2006-SR-001, 2006.
2. W.J. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object Oriented Approach to Computer Graphics*, 4th ed., Kitware, 2006.

3. L. Ibanez et al., *The ITK Software Guide*, Kitware, 2003.
4. K. Beck, *Extreme Programming Explained: Embrace Change,* Addison-Wesley, 1999.
5. D. Astels, *Test Driven Development: A Practical Guide,* Prentice Hall, 2003.
6. K. Aguanno, ed., *Managing Agile Projects,* Multi-Media Publications, 2005.
7. B. Leaf and W. Cunningham, *The Wiki Way: Quick Collaboration on the Web*, Addison-Wesley, 2001.
8. K. Fogel and M. Bar, *Open Source Development with CVS*, 3rd ed., Paraglyph Press, 2003.
9. K. Martin and B. Hoffman, *Mastering CMake: A Cross-Platform Build System,* 3rd ed., Kitware, 2006.
10. W. Sewell, *Weaving a Program: Literate Programming in WEB*, Van Nostrand Reinhold, 1989.

About the Authors

Ken Martin is a vice president of Kitware and a lead architect for both the Visualization Toolkit and CMake. His main research interests are visualization techniques, development management, and cross-platform tools. He received his PhD in computer vision from Rensselaer Polytechnic Institute. Contact him at Kitware, 28 Corporate Dr., Clifton Park, NY 12065; ken.martin@kitware.com.



Bill Hoffman is a vice president of Kitware and creator of the CMake project. He works on computer vision and software process projects at Kitware. He received his master's degree in computer science from Rensselaer Polytechnic Institute. Contact him at Kitware, 28 Corporate Dr., Clifton Park, NY 12065; bill.hoffman@kitware.com.