Seven Tips for Keeping Software Development Projects Healthy

Frank Hurley

In the beginning, most software development projects are a joy to work on. People are excitedly drawing on whiteboards, developing use cases, building their UML (Unified Modeling Language) models—it's the fun part of the project. Pretty soon, some initial code is up and running; clients adjust the requirements; developers change the UML, modify code, work out a few minor glitches; and soon the automated tests are giving the software a virtual thumbs up. Next, more functionality comes into play, tests expand to check the new code, developers implement fixes, and tests finally pass. Things are running fairly smoothly and everybody's relatively happy.

However, toward the end of the project's life cycle, the situation often changes drastically. Requirements have increased and changed, some budgets have been overrun, time has run out, and developers are 100 percent focused on patching bugs quickly, often leading to more bugs. Testing is haphazard at best. Many automated regression tests (if there are any) no longer work at all. Brainstorming has turned into blamestorming and nobody's happy.

SO WHAT'S THE PROBLEM?

For years, software management texts have discussed reasons for failures in software development projects: requirement errors, outdated software management methods, poor customer management resulting in feature creep, unrealistic scheduling—the list goes on and on. Recently, new approaches to software project management—such as UML, XP (Extreme Programming), and various requirements-tracking tools—have addressed some of these issues.

Every software development project goes through unplanned twists and turns. These tips help keep things on track.

Each of these has its own guidelines and caveats, such as a list of common modeling mistakes. But no matter what approach or design and development tools you use, problems can easily topple the delicate balance of running a software development project. To prevent these problems, you should employ good basic-practice techniques.

BASIC PRACTICES

The following tips form a guideline for basic practices that keep a project on track. The driving force behind these tips is efficiency; developers either ignore inefficient development processes, or progress slows to a crawl. Although these tips require some up-front work, they are general and open-ended enough to apply to any development project.

Tip 1:

Keep the human network up and running

Communication problems often crop up during a typical project. Developers complain that management is not clear about its goals. Management complains that developers are not forthcoming about problems. Unfortunately, the further into the project you are, the

Continued on page 60

1520-9202/02/$17.00 © 2002 IEEE

Continued from page 64

more people keep quiet and go on with their jobs. This behavior occurs because of the real concern that any suggestions developers have will directly affect (read "increase") their already overwhelming workload.

A project manager can only resolve these issues if well-known project prerequisites—such as firm project goals, source control, and so on—are in place. Even then, communication gaps occur in situations such as developer turnover or when no one owns a certain design space of the project.

Here are a few ideas for building a robust human network.

Establish point teams. Point teams resolve conflicts associated with the development project. Examples of conflict include requirements clarification, coding-style issues, errors in survival guides (see the next point), and changes to a class' interface. Developers contact one member of a point team with an issue, and that team member delivers an answer.

Normally, projects do have a point person (such as a chief architect) to resolve these issues, but often this one person is heavily overloaded, and the chances of getting a prompt answer are slim. So more often than not, conflict remains unresolved, resulting in inconsistencies throughout the project.

A point team is most efficiently set up as an e-mail group, so a disgruntled person need only send one e-mail to the group. Also, one person can belong to more than one point team.

Publish survival guides. Survival guides are short, project-specific documents describing how to work on a particular project. Examples include guides for source control, coding style, UML modeling, and point teams. No developer comes up to speed quickly when just given a stack of manuals to read. These guides should have specific examples, such as:

- to see the latest system-wide UML diagram, open J:\proj\models\latest\main.dlg;
- to get an example make file, from the command line type: cvs update -d /prog/example.mk; or
- if you don't like prepending your static final int variables with 'sFi_', write to coding_ptteam@mycompany.com.

Survival guides should evolve and be republished regularly, ideally as an intranet Web page. Most importantly, changes should be highlighted. Nothing is worse than reading weekly updates to a document and trying to pick through the whole thing looking for the lone change.

Use (but don't abuse) e-mail. E-mail can be the hero of efficiency when used properly. In particular, tools such as bug trackers that send out an automatic e-mail to the owner of the problem space prevent human error from causing unnecessary issues and wasting time.

However, too much e-mail can bring productivity to a screeching halt. Most of us have come into the office after a few days off, only to spend the first full day back slogging through unread mail. Here are three ideas for fighting off the e-mail flood:

- Publish information such as survival guides to an intranet Web site (as suggested earlier).
- Consolidate announcements into a single, scheduled e-mail. For example, write "new CVS guide is up, interface to the advertisementToXml class has changed, and team lunch is tomorrow" instead of sending out multiple messages. Or, instead of e-mail, set up internal newsgroups (or some similar communication mechanism) for announcements, topic discussions, and so on.
- Finish off runaway e-mail threads, which can severely reduce productivity. My suggestion is to alert the point team that owns the topic under discussion or establish a runaway-thread point team to summarize opposing arguments and present them for decision.

Tip 2:

Constantly look for and plug time/effort leaks

Worse than memory leaks, time/effort leaks insidiously sap the strength and morale out of developers. An outdated process that management still mandates is often the culprit here; technology can also waste time and effort. Three examples come to mind.

For one, many projects use UML-modeling tools that go the extra step of generating code skeletons for developers. This capability has the perceived double benefit of saving developer time and keeping the model and the code in sync. However, as the project goes into a mode that tracks down bugs, the cycle of changing UML, regenerating and testing code, finding more problems, and starting the process all over again can be severely debilitating. At this point in the project, it's often better to get things working and modify the UML model afterward.

Secondly, when deploying a project in a test environment, developers must often *bounce*—stop and restart—each component of the system. Depending on the project's size, the time required for bouncing can grow from several minutes to an hour or more, especially for multitiered applications running on more than one host machine.

To plug these time/effort leaks and other technology-oriented woes, the solution is to pull together a temporary *tiger team* (or person) to identify the problem and propose a solution. For example, instead of having a person sit there and bounce a multitiered, multihosted application, it might be possible to use a script. Such a script can be quite complicated and out of any single developer's realm, but well within the capabilities of a person or team with more cross-functional authority.

Finally, consider a situation in which management requires that developers not change the production system

without checking it into source control, building it on a development box, and then moving it to production. Along comes a tough bug, and this process turns hours of work into days.

Obviously, you cannot build a project of any magnitude without some process, but a rigid process can often override common sense. One solution is to establish a point team to collect opinions about the process. Even if long-term benefits outweigh public opinion, the point team should clearly publicize that decision. Keeping people in the dark often causes mistrust and, more importantly, substandard work output.

Tip 3:

Establish functional contracts (who checks what)

Invalid input data can cause big problems in software development. The major culprit—not checking an input string's size—has led to numerous security problems. Most of the time, user interfaces check parameters after manual testing uncovers problems. But often, the developers of internal functionality assume that data passed to them is valid, an overly optimistic assumption. On the other hand, having functions check all parameters all of the time is inefficient.

In 1985, Bertrand Meyer developed the Eiffel programming language that mandated *functional contracts*, which define and assign responsibility (to the function's caller or the function itself) for validating parameters. "Function" can literally refer to a programmatic function or a larger system component (for example, the B2B exchange server). Any development project can use this concept of functional contracts and reap three major benefits:

- *Reduced testing effort (and failures).* Test planners know which functions or components to create invalid-data tests for, and which ones assume validated data.
- *Increased code efficiency.* Functional contracts prevent data validation in more than one place. However, increasing robustness can require checking data validation in more than one place in the system.
- *Increased code reliability.* Invalid data never makes it into the production testing, preventing it from wreaking havoc on the system.

Tip 4:

Test early, but not too early

The golden rule of "test early, test often" is golden for good reason but often misunderstood. Although some projects start off with well-defined requirements (such as "play the

DVD" or "fly the airplane"), many do not. They may look like they do, but as the project progresses, new requirements creep in, the user interface changes, reports need new fields, and so on. Often, automated tests designed to test the system according to the original spec are constantly playing catch-up with the new system code.

Generally, you should develop high-level test plans early in the project, with test cases such as "enter the order, then cancel it, then reenter the order." Early on, avoid explicitly detailed test cases, such as "enter '04356' in the SKU field, hit the 'Submit' button, then hit the 'Cancel order' button." Although they might apply to an early system, such detailed test cases can fall by the wayside as the project progresses. Even worse, coding this detail into an automated test too early in the project will certainly become wasted effort. Save detailed automated regression tests for when the project has stabilized.

The golden rule, however, applies quite literally to unit testing. Follow the XP approach of writing tests before the code for functions, public class methods, and so on. Ensure that these tests cover all error conditions that can arise. Build or use stub code for dependencies (see tip 7, which follows) so that the test can control what is returned to it. Use a code coverage tool to gauge how well tests check the code. Remember the second golden rule of testing: "Assume code that has never been executed is broken."

Tip 5:

Support manual testing with automated tools

To maximize information about a bug, manual testers often go beyond the user interface, especially when testing a complicated, multitiered application. They will check log files, database information, and other sources to determine the state of the system. To duplicate an error condition, testers often repeat many steps to return to the state where the error condition occurred. Each of these steps can be time-consuming and are often easily automated. So supporting the test team with automated tools can greatly increase its efficiency.

Scripting tools, such as those written in Perl or Visual Basic, can interrogate databases, scan log files, and then calculate and combine as necessary to present useful information to the tester. They can also perform repetitive tasks to return to a specific state. For example, the script could login a test user, go to a specific product screen, select several products, go to the order screen, then stop and let the tester take over. Depending on the tester's technical knowledge, these scripts will require a developer's time and effort to implement. But testers should develop what the script does, and in many cases they can also support and modify these scripts.

Tip 6:

Use automated code checkers/generators

Modern software development involves the use of many other software tools to aid the development process. From yesteryear's Case tools to today's UML modeling tools, rapid-application-development tools, and the "wizards" integrated development environments, these tools increase productivity by eliminating manual work.

Building custom tools to parse code and automatically perform tasks that otherwise take up time during code reviews or, worse, debugging sessions, can extend this idea. These custom tools can be a tough sell, because building them often involves a lot of up-front work, but the payoff is well worth it.

Automated code checkers/generators provide three major benefits: They enforce a coding style, catch coding errors, and generate test skeletons. ***Enforcing coding style.*** Any software development project involving two or

more people requires a coding style (such as Hungarian notation), and many books have emphasized the reasons for adopting a style. However, these standards are typically enforced during code reviews; a fact that kills the review's efficiency. It's better to use a tool that parses the code and checks that developers have followed the correct naming conventions (among other things) than to spend person-hours on such routine checking.

Catching coding errors. Besides checking for coding policy, such a tool can also perform limited checking for code known to cause problems. For example, in some cases, Java code for a servlet cannot use any class instance variables. The solution is to build a tool that automatically detects and flags these conditions and to run this tool for all project code as part of a normal build process.

Generating test skeletons. As it is, having to write unit tests drags on developers' time; make their lives a little easier by automating at least part of the task. For example, a Java project that contains the class XmlOrderAdapter could have a script that creates XmlOrder Adapter_test, which subclasses JUnit and contains a call to each method of the class. The developer still needs to fill out the code but the script saves the effort of creating another file and putting in the method calls—it's not a very time-consuming task, but every little bit helps.

As mentioned earlier, these automated tools involve plenty of work, but the time saved in catching errors that would otherwise cause hours or days of debugging makes the effort worthwhile. Rather than building these tools from scratch, you can buy one of many third-party parsing products and configure and extend it to do the job.

For example, JavaCC is a freeware package for parsing Java (and other languages) and is extendable. C/C++ code checking tools such as *lint* help catch errors early. Also, you must strike a balance between the complexity of developing such tools and their potential benefits. The more complex the tool, the more time and energy you will spend to develop, debug, and support it. So choose the problems you solve with automation wisely.

Tip 7:

Write stub code where possible

Stub code acts like real code without doing very much. The advantage behind stub code is that developers who depend on another class that is still under development can continue working if stub code for that class is available.

For example, a class ProductOrder might need a class called OrderFulfillment. Depending on resource scheduling and the complexity of each class, ProductOrder may be ready to use OrderFulfillment before it's been developed. However, if developers initially write OrderFulfillment as stub code, methods like IsOrderShipped can just return True instead of checking the database.

For stub code to be effective, it must look just like the real code. If using stub code requires code changes on the dependent's part, it won't be worth it. For example, do not name the stub code for OrderFulfillment something else, like OrderFulfillment_stub, or name it in another namespace. Naming can be tricky, but clever use of location facilities (such as CLASSPATH for Java) can make it possible to name stub code without requiring changes to dependent classes.

As mentioned earlier, I don't intend for these tips to replace the need for well-known prerequisites in software development, such as clear requirements, reasonable scheduling, and so on. But these tips will help keep a basically sound project running smoothly. ■

Frank Hurley *is a senior consultant at Cigital in Dulles, Va. Contact him at fhurley@cigital.com.*

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

Circulation: IT Professional (ISSN 1520-9202) is published bimonthly by the IEEE Computer Society. IEEE Headquarters, Three Park Avenue, 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; voice +714 821 8380; fax +714 821 4010; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Annual subscription: $38 in addition to any IEEE Computer Society dues. Nonmember rates are available on request. Back issues: $10 for members, $20 for nonmembers. This magazine is also available on microfiche.

Postmaster: Send address changes and undelivered copies to IT Professional, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855. Periodicals Postage Paid at New York, N.Y., and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail (Canadian Distribution) Agreement Number 1445669. Printed in USA.

Editorial: Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in IT Professional does not necessarily constitute endorsement by the IEEE or the Computer Society. All submissions are subject to editing for style, clarity, and space.