DOI:10.1145/1743546.1743549

Workflow Tools for Distributed Teams?

The "Profession of IT" Viewpoint "Orchestrating Coordination in Pluralistic Networks" by Peter J. Denning et al. (Mar. 2010) offered guidance for distributed development teams. As a leader of one such team, I can vouch for the issues it raised. However, my coordination problems are compounded because email (and related attachments) is today's de facto medium for business and technical communication. The most up-to-date version of a document is an email attachment that instantly goes out of date when changes are made by any of the team members; project documents include specifications, plans, status reports, assignments, and schedules.

Software developers use distributed source-code control systems to manage changes to code. But these tools don't translate well to all the documents handled by nondevelopers, including managers, marketers, manufacturers, and service and support people. I'd like to know what workflow-management tools Denning et al. would recommend for such an environment.

Ronnie Ward, Houston, TX

Author's Response:

Workflow tools are not the issue. Many people simply lack a clear model of coordination. They think coordination is about exchanging messages and that related coordination breakdowns indicate poorly composed, garbled, or lost messages (as in email). Coordination is about making commitments, usually expressed as "speech acts," or utterances that take action and make the commitments that produce the outcome the parties want. People learning the basics of coordination are well on their way toward successful coordination, even without workflow tools.

We don't yet know enough about effective practices for pluralistic coordination to be able to design good workflow tools for this environment.

Peter J. Denning, Monterey, CA

Time to Debug

George V. Neville-Neil's "Kode Vicious" Viewpoint "Taking Your Network's Temperature" (Feb. 2010) was thought-provoking, but two of its conclusions—"putting `printf()`... throughout your code is a really annoying way to find bugs" and "limiting the files to one megabyte is a good start"—were somewhat misleading.

Timing was one reason Neville-Neil offered for his view that `printf()` can lead to "erroneous results." Debugger and `printf()` both have timing loads. Debug timing depends on hardware support. A watch statement functions like a `printf()`, and a breakpoint consumes "infinite" time. In both single-threaded and multithreaded environments, a breakpoint stops thread activity. In all cases, debugger statements perturb timing in a way that's like `printf()`.

We would expect such stimulus added to multithreaded applications would produce different output. Neville-Neil expressed a similar sentiment, saying "Networks are perhaps the most nondeterministic components of any complex computing system." Both `printf()` and debuggers exaggerate timing differences, so the qualitative issue resolves to individual preferences, not to timing.

Choosing between a debugger and a `printf()` statement depends on the development stage in which each is to be used. At an early stage, a debugger might be better when timing and messaging order are less important than error detection. Along with functional integration in the program, a debugger can sometimes reach a point of diminishing returns. Programmers shift their attention to finding the first appearance of an error and the point in their programs where the error was generated. Using a debugger tends to be a trial-and-error process involving large amounts of programmer and test-bench time to find that very point. A `printf()` statement inserted at program creation requires no setup time and little bench time, so is, in this sense, resource-efficient.

The downside of using a `printf()` statement is that at program creation (when it is inserted) programmers anticipate errors but are unaware of where and when they might occur; `printf()` output can be overwhelming, and the aggregate time to produce diagnostic output can impede time-critical operations. The overhead load of output and time is only partially correctable.

Limiting file size to some arbitrary maximum leads programmers to assume (incorrectly) that the search is for a single error and that localizing it is the goal. Limiting file size allows programmers to focus on a manageable subset of data for analysis but misses other unrelated errors. If the point of error-generation is not within some limited number of files, little insight would be gained for finding the point an error was in fact generated.

Neville-Neil saying "No matter how good a tool you have, it's going to do a much better job at finding a bug if you narrow down the search." might apply to "Dumped" (the "questioner" in his Viewpoint) but not necessarily to everyone else. An analysis tool is meant to discover errors, and programmers and users both win if errors are found. Trying to optimize tool execution time over error-detection is a mistake.

Art Schwarz, Irvine, CA

George V. Neville-Neil's Viewpoint (Feb. 2010) said students are rarely taught to use tools to analyze networking problems. For example, he mentioned Wireshark and tcpdump, but only in a cursory way, even though these tools are part of many contemporary university courses on networking.

Sniffers (such as Wireshark and Ethereal) for analyzing network protocols have been covered at Fairleigh Dickinson University for at least the past 10 years. Widely used tools for network analysis and vulnerability assessment (such as nmap, nessus, Snort, and ettercap) are available through Fedora and nUbuntu Linux distributions. Open source tools for wireless systems include NetStumbler and AirSnort.

Fairleigh Dickenson's network labs run on virtual machines to limit inadvertent damage and the need for protection measures. We teach the basic network utilities available on Windows- and/or Posix-compliant systems, including ping, netstat, arp, tracert (traceroute), ipconfig (ifconfig in Linux/Unix and iwconfig in Linux wireless cards), and nslookup (dig in Linux). With the proper options, netstat displays IP addresses, protocols, and ports used by all open and listening connections, as well as by protocol statistics and routing tables.

The Wireshark packet sniffer identifies control information at different protocol layers. A TCP capture specification thus provides a tree of protocols, with fields for frame header and trailer (including MAC address), IP header (including IP address), and TCP header (including port address). Students compare the MAC and IP addresses found through Wireshark with those found through netstat and ipconfig. They then change addresses and check results by sniffing new packets, analyzing the arp packets that try to resolve the altered addresses. Capture filters in Wireshark support search through protocol and name resolution; Neville-Neil stressed the importance of narrowing one's search but failed to mention the related mechanisms. Students are also able to make connections through (unencrypted) telnet and PuTTy, comparing password fields.

My favorite Wireshark assignment involves viewing TCP handshakes via statistics/flow/TCP flow, perhaps following an nmap SYN attack. The free security scanner nmap runs with Wireshark and watch probes initiated by the scan options provided. I always assign a Christmas-tree scan (nmap –sX) that sends packets with different combinations of flag bits. Capturing probe packets and a receiving station's reactions enables identification of flag settings and the receiver's response to them. Operating systems react differently to illegal flag combinations, as students observe via their screen captures.

Network courses and network maintenance are thus strongly enhanced by sniffers and other types of tools that yield information concerning network traffic and potential system vulnerabilities.

Gertrude Levine, Madison, NJ

What Jack Doesn't Know About Software Maintenance

I agree that the community doesn't understand software maintenance, as covered in the article "You Don't Know Jack about Software Maintenance" by Paul Stachour and David Collier-Brown (Nov. 2009), but much more can be done to improve the general understanding of the important challenges.

The software-maintenance projects I've worked on have been difficult, due to the fact that maintenance work is so different from the kind of work described in the article. The community does not fully understand that maintenance involves much more than just adding capabilities and fixing bugs. For instance, maintenance teams on large projects spend almost as much time providing facility, operations, product, and sustaining-engineering support as they do changing code.[1] Moreover, the work tends to be distributed differently. My colleagues and I recently found maintenance teams spending as much as 60% of their effort testing code once the related changes are implemented.

Other misconceptions include:

The primary job in maintenance is facilitating changes. We found that support consumes almost as much effort as changes and repairs;

Maintenance is aimed at addressing new requirements. Because most jobs are small, maintenance teams focus on closing high-priority trouble reports rather than making changes;

Funding maintenance is based on requirements. Most maintenance projects are funded level-of-effort; as such, maintenance managers must determine what they can do with the resources they have rather than what needs to be done;

Maintenance schedules are based on user-need dates. Maintenance schedules are written in sand, so maintenance leaders must determine what can be done within a limited time period;

Maintenance staff is junior. Average experience for maintenance personnel is 25 years during which they tend to work on outdated equipment to fix software written in aging languages; and

Maintenance is well tooled. We found the opposite. Maintenance tools are inferior, and development tools and regression test suites do not unfortunately support the work.

Maintenance involves much more than Stachour and Collier-Brown indicated. In light of the changing nature of the work being done every day by software maintenance teams, my colleagues and I urge *Communications* to continue to cover the topic.

Reference

1. Reifer, D. Allen, J.-A., Fersch, B., Hitchings, B., Judy, J., and Rosa, W. Software maintenance: Debunking the myths. In *Proceedings of the International Society of Parametric Analysis / Society of Cost Estimating and Analysis Annual Conference and Training Workshop* (San Diego, CA, June 8-11). ISPA/SCEA, Vienna, VA, 2010.

Donald J. Reifer, Prescott, AZ

Authors' Response:

In our slightly tongue-in-cheek description of software maintenance, we were concentrating on the "add a highway" side of the overall problem, rather than "repair the railroad bridge." We try to avoid considering software maintenance as a separate process done by a different team. That's a genuinely difficult problem, as Reifer points out. We've seen it tried a number of times, with generally disappointing results.

A better question might be the one asked by Drew Sullivan, president of the Greater Toronto Area Linux User Group, at a presentation we gave on the subject: "Why aren't you considering maintenance as continuing development?" In fact we were, describing the earlier Multics norm of continuous maintenance without stopping any running programs. We're pleased to see the continuous process being independently reinvented by practitioners of the various agile methods. In addition, we're impressed by their refactoring and test-directed development. These are genuinely worthwhile improvements to the continuous approach, and we hope the techniques we re-described are valuable to that community.

Paul Stachour, Bloomington, MN
David Collier-Brown, Toronto

Communications welcomes your opinion. To submit a Letter to the Editor, please limit your comments to 500 words or less and send to letters@cacm.acm.org.