Achieving Leaner Software

Prospecting for programmer's gold.

SOFTWARE DEVELOPERS AND MANAGERS claim that they understand the benefits of keeping their software simple, but industry reports indicate otherwise. Developers, managers, marketers, and end users continue to stuff so many features into already bloated products that one software industry elder has publicly pleaded for leaner software (Niklaus Wirth, "A Plea for Lean Software," *Computer*, February 1995).

Whether its pudginess originates externally, from excess features, or internally, from overly complex designs and implementations, pudgy software takes more time to design, code, and debug than simpler software. In "Less Is More" (*Software Development*, October 1997), I noted that cost per line of code and defect rate per line of code both increase as software size increases. Larger software requires larger teams and all the inefficiencies they entail. These days, when "short time to market" is a rallying cry for almost all software projects, project teams should be looking for ways to make their software leaner, less complex, and easier to build and deliver.

Several practices can help keep porky software from bingeing on excess features.

SOFTWARE REQUIREMENTS DIETING. Early in the project, the most effective part of a lean software diet is a big helping of requirements scrubbing. Take your carving knife in hand and go over the software requirements specification with the following aims:

♦ trim all features that are not absolutely necessary, and

♦ simplify all features that are more complicated than absolutely necessary.

Entirely removing a feature is a powerful way to save time because you remove every ounce of effort associated with that feature: specification, design, testing, documentation—everything. You also eliminate all the possible interactions between that feature and the software's other features, which reduces effort even further. The earlier in the project you remove a feature, the more time you save.

The ultimate success of requirements scrubbing depends on follow-through. If you begin with 100 features and sweat the software down to 70 features, you might well be able to complete the project with 70 percent of the original effort. But crash fitness programs are risky. If you trim the list down to 70 features only to reinstate the omitted features later, the project will likely cost more than if you had retained the entire 100 features the whole time. Features added late in the project increase software complexity because they lack the early features' built-in compatibility with the rest of the software. Late features must be integrated with the rest of the software as special cases, which contributes enormously to complexity and software bulkiness. According to Barry Boehm and Philip Papaccio, such features typically cost from 50 to 200 times as much to implement as they would have cost if added early in the project ("Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, October 1988).

Entirely removing a feature saves time. The earlier in a project you remove a feature, the more time you save.

SOFTWARE FITNESS PROGRAM. Keeping software from getting chubby in the middle of the project is difficult because pressure to add more features originates from all project stakeholders. Customers want more features because they want their needs to be addressed specifically. Marketers want more features because they perceive the market to be feature-driven; they want their software to stack up well against the competition. Developers want more features because each developer has areas of special interest and will be sure to do whatever work is needed to satisfy those interests, even if doing that increases software complexity.

These stakeholders will try to work their favorite features into the software even after the formal

Continued on page 127

Editor:
Steve McConnell
Construx Software Builders
PO Box 6922
Bellevue, WA 98008
stevemcc@construx.com

Continued from page 128

requirements specification activity is complete. Users sometimes try to end-run the requirements process and coax individual developers into implementing their favorite features. Marketers build a marketing case and insist that their favorite features be added partway through the project. Developers implement unrequired features on their own time or when the boss is looking the other way.

Even the smallest features pack more calories than you might think. Each small addition can have ripple effects on the project's design, code, test cases, documentation, customer support, training, configuration management, personnel assignments, management communications, staff communications, planning, tracking, and ultimately on the project's schedule, budget, and quality.

Several studies have found that feature creep is the most common or one of the most common sources of cost and schedule overruns (J. Vosburgh et al., "Productivity Factors and Programming Environments," *Proceedings of the 7th International Conference on Software Engineering*, IEEE Computer Society, 1984; Albert L. Lederer and Jayesh Prasad, "Nine Management Guidelines for Better Cost Estimating," *Communications of the ACM*, February 1992; Capers Jones, *Assessment and Control of Software Risks*, Yourdon Press, 1994; The Standish Group, *Charting the Seas of Information Technology*, 1994). Feature creep is a major factor in project cancellations; changes resulting from feature creep can even destabilize a product—by adding complexity—to such a degree that it can't be finished at all.

Even the smallest features pack more calories than you might think.

Change board. Midproject feature additions are expensive, but putting the software on a starvation diet by stopping changes altogether is rarely in the project's best interest. Projects must remain flexible enough to add the functionality that customers or the marketplace declare to be absolutely essential. The question for most projects is how to limit feature additions to those that are absolutely essential.

The typical means of limiting changes is to use a software change board, which acts as a central clearing house for changes. It ensures that the costs and benefits of each proposed change are considered before the change is accepted, that all important viewpoints are considered, and that all concerned parties are notified of whether each proposed change is accepted or rejected.

A change board typically consists of representatives from each of the project's major concerned parties, including project management, marketing, development, quality assurance, documentation, and user support.

Multiversion planning. To effectively support lean software, the change board must say no to new features more often than it says yes. One great help in saying no to adding features now is being able to say yes to adding new features to a future version. Consider maintaining a list of features that will be implemented in the next version. Adding a requested feature to the list emphasizes that you're listening to people's concerns and plan to address them at the appropriate time. This approach works especially well if you create a multirelease plan that maps out a multiyear, multiversion software strategy. Seeing the plan helps people understand that the feature they want is more appropriate for a later release and that it won't be delayed indefinitely.

Short release cycles. Planning to deliver software often is one key to keeping each release from bulking up. For users and customers to accept the multiversion approach, they must have some assurance that there will in fact be a version after the current one. If you don't think you'll eat again until next week, you'll eat more than if you know your next meal is only a few hours away. If users fear that the current version will be the last version they ever see, they'll try hard to put all their pet features into it. Witnessing a series of short release cycles helps build users' confidence that their favorite new feature will make it into the software eventually, just not into the current release.

To support lean software, the change board must say no to new features more often than it says yes.

User interface prototyping. While change boards are probably the most *common* means of controlling midproject feature bloat, prototyping is one of the most *effective*.

Prototyping tends to lead to smaller systems. My wife doesn't necessarily know what I want for dinner, and the features that developers think users want are not always the same features users themselves would order from the software menu. Conversely, without prototyping, users sometimes insist on features that look good on paper but work poorly in the live software. User interface prototyping can cull those features early in the project, which makes it easier to keep both the feature set and the budget fit and trim. One survey of projects that used prototyping found that it reduced development effort (and, by implication, software size) by 45 to 80 percent (V. Scott Gordon and James M. Bieman, "Rapid Prototyping: Lessons Learned," *IEEE Software*, January 1995).

SOFTWARE MIRACLE DIETS. None of these practices is a software miracle diet, and none of them will keep a slothful software project from piling on the pounds. But when you combine requirements scrubbing, change board, multiversion planning, and user interface prototyping into a comprehensive software fitness program, the result can be fast delivery of svelte, powerful software.

Keep the fat where it belongs: At project end, use the money from your lean software savings to treat the development team to a well-deserved banquet. ♦