Managing Change for Rapid Development

How to get people and technology to work together.

English mathematician and philosopher Alfred North Whitehead once said, “The art of progress is to preserve order amid change and to preserve change amid order.” Although he wasn’t talking about software development, I think his comment is apropos. The highly competitive software marketplace demands rapid response, yet we must preserve order so that change leads to desirable progress. But if our software process is too bureaucratic or stodgy, we can only maintain order by sacrificing our ability to change and compete. In this issue, Gene Forte offers a blueprint for managing change that balances these concerns.

—Roger Pressman

THE PACE OF DEVELOPMENT REQUIRED to produce software for the World Wide Web threatens to overwhelm our ability to manage change. Unfortunately, when competing in “Web space” we are tempted to regard processes and tools for managing change as overhead that leads to slower execution. But experience shows that appropriate change management techniques can speed software development by improving communications and preventing rework. More than ever, we must keep our focus on the desired outcomes—the right products and technologies delivered to the right customers at the right time—and regard change itself as a factor to be managed, just like technology, resources, and time.

When a product is successfully brought to market, it immediately becomes a target for imitation and can rapidly lose market share to follow-on competitors. Successful growth depends critically on the company’s ability to rapidly improve and extend its product in response to customer feedback. But when innovative companies try to shorten their engineering cycles while being sensitive to new ideas and evolving customer requirements, they discover that change is the enemy of rapid development. The combination of complexity, rapid engineering cycles, and rapid change creates a major software development risk.

RUNNING WITHOUT TRIPPING. We can overcome change’s accompanying risks only by actively managing change. We must understand the impact of every change. This requires that we
- know what changes are occurring, which are proposed, and the overall rate of change in the project;

We can overcome change’s accompanying risks only by actively managing change.

- assess the impact of each proposed change;
- consciously decide to allow or disallow each change, based on a cost/benefit function;
- immediately communicate each change to everyone who needs to know about it; and
- ensure that all the desired changes, and only those changes, actually appear in the shipping product.

Change management—doing only the most important things in the least amount of time—is in marked contrast to “change surrender.” Surrender occurs when changes enter a product from various sources such as marketing, sales, customer support, development, and so on, without assessment or prioritization. This results in all changes being given equal attention by default, no matter how severe their impact or how small their benefit.

CHANGE AS FEEDBACK. You can think of change control as a closed-loop feedback system that adapts in real time to changing inputs while maintaining focus on a desired goal. The feedback path involves knowing explicitly, and in detail, what changes are occurring and what the impact of each change is. Using the path also means having metrics on the total amount and rate of change—requirements changed, lines of code changed, and so on. The control function involves making conscious decisions to allow or disallow changes. Alternatively, in a multicycle product development scenario, you must decide whether to put the change into the next release cycle. In either case, you should base
your decision on a cost/benefit function that may sometimes be imprecise but is always deliberate. To make effective decisions requires more than a description of the proposed change. The change manager must also have its anticipated schedule impact (including dependent activities) and the perceived value for the product, say as a market share estimate. For example, a proposed feature change might add four weeks to the release date. Program management could then decide which path would result in greatest market share for the product: shipping on time without the feature or shipping three weeks late with it. Control can also be exercised by managing the rate of change to stay within control limits established for each phase of a project, based on prior experience.

Because of the sheer volume of information involved and its rate of change, the closed-loop feedback control system model represents major challenges for larger projects, especially when they consist of multiple coordinated development teams. To effectively put the right information into the hands of the right people so that they can manage change requires effective intra- and intergroup processes and information management systems.

The bottom line? Without a plan, a process, and an information system for managing change, all but the smallest, simplest projects are at risk of wandering out of control and adding time to the development schedule.

**CHANGE MANAGEMENT PRINCIPLES.** There is no “cookbook” for implementing effective change management within a particular organization. The optimum process and tooling are highly sensitive to the organization’s scale and business model. However, there are several key principles of change management that can act as guides to success.

- **Manage requirements and features.** Lack of appropriate requirements management or feature management is the most common “killer” of software schedules. You should prioritize all features and draw a feature “cut line” to establish a critical-mass feature set, defined as the minimum functionality that will create market momentum. All features and enhancements don’t have to go into the current release. A product that is introduced sooner than its competitors and that evolves rapidly over several releases gains the advantages of preemptive mind-share and earlier feedback from real users. To make this strategy effective, it must be accompanied by change management with a central point of control that extends from requirements through code deliverables. This ensures that you change only critical features and complete impact analysis for any variations before they are proposed.

- **Use project timeboxing.** Timeboxing is the practice of fixing resources and schedule, and adjusting the feature set to match. This constrains project size and duration of release cycles to manageable dimensions and forces the project team to focus on a relatively small feature set for each release. To make timeboxing work, you need a realistic commitment process with accurate project scheduling based on historical data. It’s also critical to have staff available when needed: the committed plan should have a defined staffing ramp with a control limit to act as an early warning if recruiting falls behind.

- **Design for change.** A well-thought-out, open architecture reduces the impact of critical feature changes later in the lifecycle. It supports reuse of subsystem components from previous projects or external sources. In addition, you can add deferred features in the next release without making major structural changes. A little extra time spent before the initial product introduction can result in faster product evolution once your product idea is exposed to competitors’ scrutiny.

- **Invent in an integrated change management system.** This includes change and defect tracking, full-featured configuration management, build automation, impact analysis, and release management. An integrated change management system helps ensure accurate and low-overhead movement of changes through the development and quality control process. The ability to track changes from a single task or source to multiple artifacts as a unit is key.

- **Provide a single point of control for changes.** As the project moves toward completion, the “change funnel” should get narrower, with fewer changes allowed. To ensure this happens, you need a central person to authorize changes, sometimes called the Change Czar. This role might be filled by one person or by a small group that represents several constituencies. The Change Czar should receive an impact analysis for each major change. Impact analysis tools help to identify dependencies to aid in cost assessment before change is authorized.

- **Allow major changes only at predefined project milestones.** Rather than allowing changes to ripple through a project whenever they arrive, you can redefine several change points in a project and have everyone make changes at these times. The exception to this is change resulting from defect removal, which should be continuous throughout the project.

- **Engage in proactive change communication.** If all parties involved in the software development life cycle are “kept on the same page” continuously, projects can avoid wasting large chunks of time on rework. An easily accessible repository lets all team members—including marketing—view requirements, design, and code artifacts online, and easily update documents to maintain synchronization. Internal web sites are an excellent mechanism to provide open access and establish dependency links (hyperlinks) between elements. A change notification mechanism, ranging from simple e-mail to auto-
mated document change notification tools, provides an early warning system that alerts developers to changes that may impact their work and result in project instabilities.

- **Employ early feedback through continuous integration.** Monitor the rate of change of requirements and code and maintain a continuous integration development process. These are excellent sources of early feedback on a product's stability and how well change is being managed. Continuous testing and integration helps teams maintain a high level of stability throughout the project by constantly evaluating the architecture, interfaces, and implementation, and by preventing the accumulation of latent defects. A developer/tester buddy system, along with daily builds and “smoke tests” (short regression tests) are key elements of continuous integration. Multiple internal releases also help the product achieve stability, which in turn helps in continuously refining schedule estimates.

- **Maintain project critical mass.** Soft factors also affect the ability to manage change. Knowledge of the product architecture, rationale for previous design decisions, and familiarity with the code itself all contribute to effectively managing change and responding to new product opportunities. Strong relationships between team members, a common vocabulary, common design notations, and sensitivity to each other’s thought processes and styles are also potent competitive factors. All of these considerations argue for keeping the core of project teams together for multiple product cycles if faster release cycles are important to your organization’s success.

To keep the focus on the desired outcomes—the right products and technologies delivered to the right customers at the right time—we must regard change itself as a factor to be managed, just like technology, resources, and time. While change management principles require effort and investment to implement, they can speed up software development by eliminating many of the common sources of risk, uncertainty, error, and rework. Especially when viewed over multiple release cycles, change management is one of the most potent weapons we have to continue accelerating the pace at which we can deliver more complex and useful software in greater quantities.

Gene Forte is the software tools program manager for Intel's Internet Communication Group, where he specializes in enabling methods and tools for rapid software development on PC platforms. Prior to joining Intel, Gene was an industry consultant and publisher of the Case Outlook report on computer-aided software engineering. Gene thanks Intel's Cynthia Ward, who coauthored an internal paper that was the source for many ideas expressed here.

quality time

Continued from page 119

My OO testing research explored an approach to assertion placement based on error masking. As expected, I found that information hiding and encapsulation can be detrimental to system-level and integration testing, but not always. For example, suppose that information deliberately hidden and accidentally corrupted in turn corrupts other information that is not hidden, such as information allowed to flow freely throughout the program. Here the hidden information propagates by piggybacking on the good fortunes of a second party. This example shows that you cannot simply assume information hiding enables error masking; you must employ the appropriate methods to determine if it can.

**ASSERTION CAVEATS.** For brevity, I have sidestepped several important, problematic assertion issues. Assertions are by nature intrusive. Also, deriving correct assertions can be problematic. For example, what if your assertions can’t recognize problems that have corrupted the state? Or equally annoying, what if your assertions trigger nonexistent problems? These shortcomings have no quick solutions. For some software systems, assertions will not be plausible. But for most, the advantages of intelligently placed assertions should not be overshadowed by the limitations.

I conclude with a hypothesis: Assertions injected into a program to boost the fault-revealing effectiveness of test scheme D cannot lower the fault-revealing effectiveness of a different scheme D’. I do not know how often this will hold, and I can quickly derive counter examples in which it doesn’t. But if my hypothesis does hold generally, it could move us away from decades of debate over which testing technique is better. Instead, research efforts could explore methods such as assertion instrumentation that add value to virtually any testing scheme—a change in focus that could provide a key paradigm shift for software testing practice and research.

Jeffrey Voas is a cofounder of Reliable Software Technologies and is the principal investigator on research initiatives for the Defense Advanced Research Projects Agency and the National Institute of Standards and Technology. Voas also co-authored Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995).