Author Contact: james@satisfice.com

The Highs and Lows of Change Control

James Bach, SmartPatents

Change control. For me, this vital issue in software project management immediately conjures the image of the famous 1984 Macintosh commercial. Remember that one? Gray drudges standing row upon row, staring with numb compliance at a lecturing bureaucrat, set free by a hero who bursts in and hurls a hammer that breaks the spell of Big Bad Brother.

This image depicts my personal struggle with the problem of managing improvements to a product. Although my temperament is that of hammer-throwing hero, my role as quality assurance guy is more like lecturing bureaucrat. Part of me wants to open the flood gates to improvements and better ideas; the other part wants to protect the existing quality of the product by limiting change.

CHANGE CONTROL

Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas also originate in the minds of those rogues, and a burdensome change-control process could effectively discourage them from doing creative work.

Change control is vital. But the forces that make it necessary also make it annoying.

My ambivalence about this issue is only deepened by the fact that change-control processes are easily corrupted. Change control means risk analysis, and there's no easy or certain way to do that. Coupled with the amazing capacity we humans have for oversimplifying the complex, change control can become mindless resistance to change and an automatic rejection of all risk, regardless of potential reward.

Or, just as easily, change control may degenerate into a set of empty rituals that allow any change to be made as long as the rituals are honored. Such a process is less a practical device than a sort of gargoyle meant to scare evil spirits or impress clients. Midway between allowing nothing or everything, change control may also become a political filter where change is resisted unless you're in with the in-crowd, regardless of the situation's merits.

At SmartPatents, change control processes are (at least officially) my responsibility. How can I avoid the role of bureaucrat, gargoyle, or political pollster? How can I help the process work *well*? Let me describe the process we've arrived at, then decide for yourself if I deserve the hammer treatment.

PROBLEM AND PROCESS

The only legitimate purpose of defined process is to solve problems. So let's start there. What problem do we have?

SmartPatents is a market-driven software company, not a contract-driven one. We aren't regulated either. We don't need to justify ourselves to an outside client or agency. Our need for change control comes mainly from our desire to minimize the chance that a major problem will be introduced into the product while we're trying to improve it, especially late in the cycle. We want to minimize expensive and time-consuming regression testing. We also want to assure that the change process respects the concerns of each team member who may be impacted by a particular change. At the same time, our process must be flexible enough to let us add product functionality late in the development cycle, because that's how market-driven software companies compete.

When I joined the company last October, it had three change-management processes in place:

- We stored all source code in a source control system.
- we compiled and linked the product according to an official build process executed on a dedicated build machine; and
- we put the project under *code freeze* in the last few weeks of development.

Code freeze doesn't mean that code stops changing. It means that we impose a formal code-change protocol. Originally, following protocol meant that Gordon, our senior tester, had to grant a waiver for each change. Being an easygoing fellow, Gordon waived virtually everything. Since he knew about each change, he theoreti-

cally had some inkling of what needed to be retested for each new build.

This protocol was administered through a communal spreadsheet. Anyone requesting a change was supposed to enter the request as a single line in the sheet. Gordon reviewed the spreadsheet periodically and granted waivers. However, if someone, for whatever reason, made a change without going through the spreadsheet, Gordon would never know.

Many waivers were granted in the hallway, off the record. So the process got pretty sloppy. I don't mean people behaved badly or actively subverted the process. The process became sloppy because it was not designed to fit the way technical people work. People get distracted, forget details, and make decisions on the fly, often in hallways. Any robust method of change control must take this into account.

REDUCING THE RISK

To reduce the risk of a dangerous change, I wanted a more visible, reliable process. So we retired the old spreadsheet and installed a bug-tracking system to manage change requests. (A bug report at SmartPatents can be a change request, problem report, or other task assignment that may involve code changes.) For each code change after a code freeze, we required developers to place a check-in comment in the bug database with the change request's ID number. Prior to each build, a QA engineer examines the check-in comments and verifies that each code change corresponds to a waived bug in the database.

This process is a system for tracing code changes back to change requests and approvals. Barring intentional subversion, the system is theoretically airtight. Hallway waivers can slip through the cracks, and developers can still forget to follow the process. However, when that happens, the prebuild comment check detects the problem, and the build stops until each rogue change is justified and documented. This feedback loop allows the process to tolerate human error while gently discouraging it.

For this process to be more than an empty ritual, however, it needs a brain. How do we decide which changes to approve? We do that mainly via the *waiver meeting*, a systematic, daily review of the bug list by representatives of each major project function: marketing, support, documentation, development, and QA. In this meeting we decide whether to allow a change or postpone it to the next project release. Decisions are made by unanimous consent.

Since December 1997, this has been our official process for change control after code freeze.

> It is important that change control not become either mindless resistance to change or a set of empty rituals for permitting change.

Now, any wizened veteran of software process improvement who heard a story like this—perhaps over coffee during a morning break at a software quality conference—might well nod sagely and have another muffin. But behind the pastry, he'd probably wonder: "What part of James' story is real, and what part is fantasy? How does he know that his process is worth following? What are its problems and liabilities?"

HURLING THE HAMMER

Reality versus fantasy? Problems and liabilities? Sit down, this will take a while. There are a lot of problems with this approach. We've had to make a number of changes to our change-control protocol in order to make it more efficient and effective. I'm still not satisfied with it. If I call it a best practice, that's only because I haven't yet become aware of a better one.

Jerry Weinberg once quipped, "We don't manage projects, we manage stories about projects." My change-control story is a simplification of what we actually do, so here are some additional details to give you a better feel for our situation and the struggles we're having.

- It seems that every other waiver meeting is cancelled or postponed. Sometimes I feel too tired or harried to run them. Daily meetings wear everybody down, but I haven't yet thought of an effective alternative. If we don't grant waivers frequently, we hinder product improvement. Perhaps we are freezing code too soon and staying frozen too long.
- We go through periods where we're so busy that we neglect the prebuild comment check. The truth is, I'm not always very motivated to do it. We've had such good compliance to the protocol that I no longer expect to see unauthorized changes.
- What looks like good compliance to the protocol may actually be the unauthorized reuse of old waivers on new changes. I only have a developer's word that a particular change is genuinely related to the grant of a particular waiver. Even if a developer is absolutely honest, there is a lot of room for creative interpretation.
- Each waiver meeting is designed to handle many issues. We hunch around a monitor and discuss each problem until we reach a consensus. The group gets impatient if a discussion about an item goes more than a few minutes. Since we typically handle 15 to 20 problems per hour, and we never let any single meeting go more than 90 minutes, the pressure is on. I don't think that pressure creates a good environment for carefully considering each issue.
- Sometimes we encounter a problem that requires a detailed explanation and analysis by an expert who is not normally in the waiver meeting. The meeting is then disrupted while someone searches for that expert.
- We often have to hold the waiver meeting without an ideal set of participants. The minimum quorum is one person from QA and one development lead. This allows the meeting to go forward but increases the risk of a poor decision or an unpleasant surprise to support people or technical writers.
- Sometimes the developers are confused about what a waiver means. A waiver is always permission—but usually not a recommendation—to make a change. We rely on the judgment of each developer to determine which changes are too risky. (And what does "risky" mean? Its definition varies with the situation and the developer involved.)
- We rarely check that waived problems are actually being properly resolved. It's

possible for a waived problem to sit on a developer's plate for days (for example, if a developer forgets to check the tracking system) or to be resolved in an unacceptable way. We have occasionally been forced to postpone important changes just because an earlier decision to make the change was not noticed in time.

- Gordon and I still sometimes grant waivers outside of the waiver meeting. Since doing so bypasses the meeting, we try to grant hallway waivers only on noncontroversial matters. But sometimes we're wrong.
- What if a single waiver relates to the work of several developers? There is no easy way to handle that case within our system. It's possible for a problem to bounce around among several developers, none of whom takes full responsibility for it.
- What happens to changes that are mandated by a code review? To cooperate with our change protocol, developers must reduce each issue resulting from a code review to a change request. But the resulting paperwork has discouraged developers from holding code reviews after code freeze.
- Some problems in our system are chronically kicked forward from release to release, and are never resolved. We call these problems *bow wave bugs* because they keep getting driven ahead like dolphins riding the bow wave of a big ship. A bow wave situation develops when a problem is too hard or risky to fix during code freeze of the current release, but not quite important enough to make it into the next release's project plan.

SMACK!

We have refined the code freeze protocol in a number of ways to respond to these experiences and problems. In one case, the project lead proposed and implemented a special change-control board to consider changes that are too complex for waiver meetings. The CCB, as we call it, has the same membership as the waiver meeting, but instead of being held every day to consider a series of issues, it's held at the discretion of a project lead to consider a single issue. The CCB gives us a systematic way to have deeper conversations about changes.

I feel good about our continuing refinement of this process, but I wonder about something that may be a tragic flaw in this grand design. Perhaps you've already spotted it: Our change-control process may indeed work too well.

Good risk management late in the project does indeed compensate for problems with poor requirements analysis and design early in the project. However, if we believe we can clean anything up in the end, we don't have much incentive to learn better problem-prevention methods. Rather, we may feel an incentive to be even more reckless with our early design and coding, in the interests of pumping more features into the product.

In other words, my company may be addicted to change control. And each improvement in change control may deepen that addiction.

I experienced this sort of thing when I worked at Borland, where I first encountered the waiver meeting process, which we called *the bug council*. At first we administered the meeting by writing each approved change on a big whiteboard, which could hold only about 200 changes. As we reached that number we'd begin to murmur and moan about how there were too many changes coming through. Two hundred items on a whiteboard look intimidating. We'd start asking basic questions about the quality of our schedules and processes. When we "improved" the way we managed the meeting by putting the change requests and waivers online, a curious thing happened: We no longer worried about having 200 change requests, or even 2,000. Without the visibility of the whiteboard, we became numb to the psychological impact of all those change requests.

We have to control our changes. We try to do it well. Yet, unless we keep our eyes and minds wide open to the dynamics of the situation, our tactical success will ensure our strategic failure. Oh, my head hurts to think about it, but that's how it is in the nonlinear world of software processes. Be warned ye bureaucrats. Just when you get all your drudges in a row, reality bursts in and breaks the spell. ❖