Normal Processes

by Brian Marick

In this issue, Mark Johnson has an article about ISO 9000 registration…

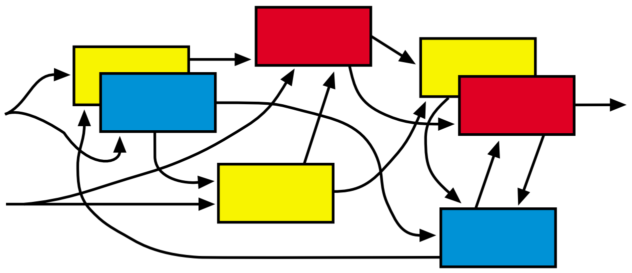
Wait! Come back! It's a refreshing take on a tired subject. It grapples with the messiness and uncertainty of the *real* workplace. Too often, process designers yearn for the abstract Right Process. I know I do. That leads to processes that don't work with real people in real situations. Mark's article is a corrective.

Process design and the management of software development is a theme of this issue. While editing, I found myself thinking about catastrophes. In his 1984 book *Normal Accidents*, sociologist Charles Perrow claims that some systems encourage local problems to blossom into systemwide accidents, despite efforts to make them safe. Sometimes the attempts to increase safety themselves cause problems. (The experiment that led to the Chernobyl disaster was intended to improve a safety system.)

These systems have both *complex interactions* and *tight coupling*. (I'll define those below.) Software development processes also have these characteristics, perhaps inevitably. What worries me is that I see process improvement efforts that add *more* complexity and *tighter* coupling. So I'd like to sketch Perrow's ideas and suggest that you keep them in mind next time you about think about how software should be created.

Complex interactions. A complex interaction is best defined in contrast to a *linear* one. That term is intended to evoke the image of an old-fashioned automobile assembly line, in which production takes place in a series of steps that are isolated and visible. When something breaks, it's easy to tell which step is to blame. It's also easy to understand the consequences: the following steps must stop. You can fix a step without causing something unexpected to happen to earlier or later steps.

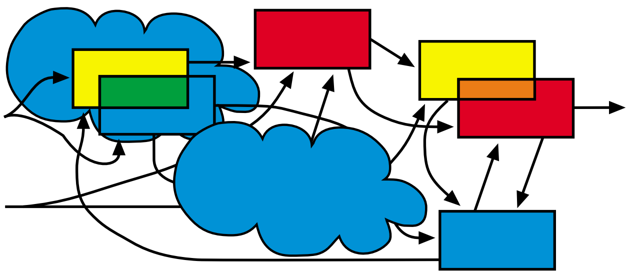
What if the interactions aren't so linear? What if they look more like this?



Failures now have less predictable consequences. If a step takes input from *two* preceding steps, what happens when just one of them fails? If a step's output is simultaneously used by two later steps, can you adjust the first without simultaneously mal-adjusting the second? What if the steps are involved in feedback loops, which are notoriously difficult to understand?

Notice also the overlapping boxes with no arrows between them. Those boxes are not in *production sequence*—they have no intentional interactions. However, they nevertheless interact. For example, they may be so physically close that a failure in one causes a failure in the other. (Water in a drinking water tank may spray onto an outflow valve, freeze, and crack it.) These interactions are often unknown until after an accident.

Moreover, complex systems provide limited visibility even to known interactions. In the Three Mile Island nuclear accident, the operators relied on instruments and indicators that, in some cases, lied—and at best provided an incomplete picture. They were forced to guess about what was going on. When they guessed wrong, they compounded the problem. Therefore, the previous picture is misleading. To capture the experience, connections should be obscured, like the following:



You find more specialists in complex systems. But when unplanned interactions span specialization boundaries, it's much harder for anyone to understand what's going on and know how to fix it without breaking something else.

Tight coupling. Again, I'll define by contrast.

If you're building a house, fall behind schedule, and the shingles arrive before you're ready, you stack them somewhere for a while. You can do that because the arrival of shingles is *loosely coupled* to their use. It's much more difficult to make a chemical reaction stop while you fix a downstream step.

If the shingles arrive late, you can still make progress by doing something else that you originally planned to do later. In contrast to houses, there may be only one order in which you can make a chemical product.

More generally, a loosely-coupled system allows alternate methods to reach the goal, as well as the time in which to find them. Methods are opportunistic—successful jury-rigging is common. In a tightly-coupled system, however, attempts to jury-rig often make things worse.

(Note that Perrow's definition of "tight coupling" is at odds with how the term is used in the software engineering literature, in which its meaning is closer to what Perrow calls interactive complexity.)

Seeing Through Perrow's Lens

Suppose I'm a development manager. One of my developers has read about "pair programming" as used in the rapid development style called Extreme Programming (**www.armaties.com**), and thinks we should try it.

Teams using pair programming never write code except in pairs sitting together at one computer. One types, while both actively observe and talk through what is being created (essentially doing a real-time review). At the end of a development task, pairs split up and move on to other tasks in other parts of the system.

Having read Perrow, what thoughts might I have? Pair programming looks good. It works to reduce complexity by preventing dangerous specialization. Using different partners means that eventually everyone gains familiarity with many parts of the system. That makes it easier to make changes without inadvertently breaking something you didn't know about—and when you do break something, you can figure the problem out faster. Since Extreme Programming is designed for complex programs whose requirements change frequently, generalists are essential.

Pair programming increases visibility. In most organizations, there's a formal process and an actual one. The formal process might say "no coding until after the design review." The actual one says, "You don't have *time* to wait for the design review. Start coding now, and incorporate the results of the review after it happens." (This is an attempt to reduce coupling, akin to starting another task while waiting for the shingles to arrive.) When the project runs into problems, the Powers That Be tend to intervene in what's supposed to be happening, not what's actually happening. The interventions can make the actual process worse, as happened at Three Mile Island. In pair programming, the formal and actual processes are closer because monitoring is more immediate. Coding while waiting for a review is easy—who's looking?—but slipping unwatched changes past your "other half" is hard.

Why do reviews take forever? The reviewers all have other tasks that are more important to them. My code doesn't depend in any way on the reviewer's—we're not in production sequence—but my progress depends on hers. If she's having problems, my review will be hurt. (In the lingo, she causes a *common mode failure*.) In pair programming, the person I depend on thinks our task is as important as I do.

Coding before the review is done causes yet more complexity. The output of the design now feeds into two parallel steps: reviewing and coding. When coding reveals problems in the design—which it *always* does—more "components" (people) are affected. In pair programming, everyone immediately concerned is sitting right there at the moment problems are found, so the feedback loop is simpler.

I'm making no blanket condemnation of design reviews. There's ample evidence that we should do more of them, not less. My point is only that Perrow's analysis is one way to help us better anticipate consequences of our process decisions. STQE

STQE magazine is produced by STQE Publishing, a division of Software Quality Engineering.