Impact of Overtime on Productivity

http://xprogramming.com/articles/jatsustainablepace/
Ron Jeffries, Posted on: April 14, 2006

More features will always bring more revenue, more customer satisfaction, other good things. Therefore there is always pressure for people to work harder, longer hours. This is demonstrably a Bad Idea. Here’s some evidence, and some ideas about how to know if pressure is too high.

**Industrial Accidents and the Professional Programmer**
The XP Practice “Sustainable Pace” suggests that the team should work at a pace which produces a steady flow of results, sustained indefinitely. They work overtime when it’s appropriate, and work always to maximize productivity week in and week out.

There’s always a business need and desire for more results. This is entirely natural. As a result, it’s common to expect programmers to work long hours “to get more done”. Clearly there must be some upper limit to how many hours we should put in, but how can we figure out what that limit is?

I was watching a TV program about industrial accidents, and it got me to thinking that we might be able to draw some conclusions from industrial data. I did some surfing on the subject, and the evidence is pretty clear: *Industrial accidents increase disproportionately as hours increase above forty per week, or above 8 hours per day. More than half of all industrial accidents occur in jobs with extended working hours. The generally-accepted hypothesis is that the accidents result from tiredness.*

There has also been recent news about the impact of long hours on medical interns, reporting that after long sessions they are twice as likely to have an auto accident while driving, and five times more likely to have a near miss. After a month of overtime, they drive, literally, as if they had had three or four stiff drinks.

This last study makes me resolve to ask the doctor at the emergency room how long he has been working before I let him make any big decisions about me. For our purposes here, what do these studies suggest about programming?

**Is Mental Acuity Involved in Programming?**
Does programming require us to be mentally alert, to be sharp and on top of things? Well, let me think: of course it does. Programming is a work of the mind. We need to keep many details in our heads at the same time, from the details of the programming language to the only partially understood details of the program we’re working on, which often seems to have been written by some bizarre consortium of fools and fiends.

If the risk of an accident is essentially doubled at twelve hours compared to eight, what about the risk of inserting bugs? By the time a measurable accident occurs, the worker has probably been fumbling around, working erratically, for quite some time, and finally got so far off track that an accident was the result. My educated guess would be that the insertion rate for defects increases far more rapidly than the risk of industrial accidents. What the accident statistics probably tell us is that a tired programmer has double the chances of putting in a very serious bug, but has probably also put in a much higher percentage of smaller ones.

**Disproportionate Impact of Defects**
Defects are very disruptive to software development. I commonly visit teams who are spending one third to one half their time fixing bugs. This is time that could be spent producing features. Were it not for these bugs, these teams could produce between one and a half to twice as many features!

In addition, defect fixing is very unpredictable. It’s common for a team who fixes bugs in an average of a couple of hours to encounter a few bugs that take days or weeks to find. The fact that they have this experience tells us that those bugs are very important to find, since otherwise they wouldn’t spend the time.

It’s well-established that it takes less time and costs much less to prevent defects than to fix them. A recent article in Crosstalk, the Journal of Defense Software Engineering, reports that “finding and fixing bugs is the most expensive cost element for large systems and takes more time than any other activity.”
Our progress will be faster and more predictable if we avoid putting defects into the software. In Agile methods, the primary approach to defect prevention is comprehensive programmer and customer testing. How do those things play in an environment of long hours?

**Practices Slide Under Pressure**
A common effect of putting teams under pressure is that they will reduce their concentration on quality and focus instead on “just banging out code”. They’ll hunker down, stop helping each other so much, reduce testing, reduce refactoring, and generally revert to just coding. The impact of this is completely predictable: defect injection goes way up, code quality goes way down, progress measured in terms of net working features drops substantially.

A recent Circadian study addressed productivity in white collar workers. They found that even as little as a ten percent increase in work hours could result in a 2.4 percent drop in productivity, while sixty hour weeks could result in a 25 percent drop in productivity.

None of these studies have combined just the right ingredients to tell us with accuracy what will happen if we drive our people too hard, or where the line is. We can be certain, however, that code quality will go down, and defects will go up, and that the effects will be felt at pressures and hours well below 12 hours days or sixty hour weeks.

**What Can We Do?**
Combine high pressure with long working hours and the result is inevitable. Real productivity declines, defects go up. Unfortunately, we have no real way to measure defect injection until after the fact. The less capable our testing is, the less we know about defects. Pressure and long hours break our instruments just when we need them most.

Now I would be inclined to the notion that management and customer should put on only a tiny bit of pressure, enough to keep everyone energized and aware that there’s a need to perform. I believe that a single threat, even if “intended” in a humorous way, will slow things down for at least a day. So hold back on the “heads will roll if we don’t get things done” comments, even if you say them with a grin. I really don’t think they help.

But you’re not me, and you might feel differently. Is there some way to “measure” whether pressure is too high, hours too long? Frankly, I’m not sure, but here are some ideas.

Let’s imagine that for a while, we have done the things we know we should: we have comprehensive unit tests, we have customer tests for every feature, we have a continuous build process that tells us when the build is broken, and so on. Then here are some indicators that things may be going wrong:

- Is the ratio of test lines to code lines going down?
- Is the ratio of customer tests to features going down?
- Is time spent fixing bugs going up?
- Are people pairing less often?
- Are integrations taking longer?
- Is the build broken light coming on more often and staying on longer?
- Are cosmetic tasks in the code being skipped?
- Is the refactoring board filling up with red cards?
- Is the team room messier than it used to be?
- Are there more food containers in the waste cans?
- Are people asking more questions about what is going on in the standup meeting?
- Have you stopped even having a standup meeting?
- Are there more arguments, more people who are angry with each other?

Almost all of these things could be charted on simple, informal “Big Visible Charts”. Most of them are pretty harmless to track, but could be good indicators of things going wrong.

What additional ideas can you come up with to tell you when the pressure and hours are too high? Pass them along, and I’ll update this article. Meanwhile, don’t work too hard. It won’t pay off.