Why Isn't There an "*I*" in *Team*?

I'd like to start off this column with a story about a friend of mine. We'll call my friend "Jeff." The story involves his daughter, who we will call "Jill." It seems that Jill, who is an accomplished swimmer, once had a contest with a friend of hers to see who could hold her breath the longest. Jill easily won the contest – a fact that had to be verified by her friend – as Jill had the endurance and willpower to hold her breath so long that she passed out. I also hear that ambulances were involved afterwards.

You know, as a father, I would be proud to have a daughter like Jill[1]. I know Jeff is! I imagine anybody with that kind of willpower is rather immune to peer pressure, and will also ultimately succeed in whatever she sets her mind to. She definitely is tenacious – which is a synonym for stubborn. Obviously, she has the makings of a fine engineer.

I have long thought that those children who have the potential to be truly gifted engineers could be identified easily in playschool. All it requires is a sandbox and a few toys. Insert children. There will be a few children who will happily play with others, enjoying the sand and the toys. These are not future engineers. Over in the corner of the sandbox, however, there will be a child happily playing alone, by himself or herself. Whenever any of the other children come too close, this child will throw sand at the intruders, and may go as far as knocking intruders on the heads with the sand-building instruments, all the while clutching at his/her toys shouting, "Mine!" This is a potential engineer!

Let's face it – for developers, a good chunk of our day is spent sitting alone in cubicles, ignoring the hustle and bustle surrounding us. We occasionally put on earphones and listen to music, or just learn to tune out surrounding noise. And there we sit – oblivious to the outside world, creating "stuff." The problem, as any good software tester will tell you, is that putting your stuff together with my stuff will uncover interface errors. And again, as any good tester will tell you, interface errors are numerous and often difficult to fix. From my point of view, the problem is your code. You, of course, might have a slightly differing opinion. Many testing authorities say that up to 75 percent of all errors stem from interface problems.

It's not all our fault; we're trained to operate in isolation, starting in college. We are encouraged to come up with individual, innovative solutions to problems. The problem is that your individual, innovative solution might not make a lot of sense to me, and I have to write the software that interacts with yours. To make my software interact with your software, we need to communicate before and during the software creation. This can be done via the tried-and-untrue method of meeting after meeting after meeting – but this method hardly ever works. What I need is a way to encourage good teamwork and cooperation, without spending 50 percent to 70 percent of my day in meetings. Enter Personal Software Process℠ (PSP℠) and Team Software Process℠ (TSP℠).

Granted, I am a PSP/TSP zealot. PSP encourages good individual programming practices, and TSP creates an environment that permits motivated individuals to work as an efficient team. I have been teaching PSP/TSP since 1998 – and it's one of the few tools that I teach to developers that (almost) always prove beneficial. Why almost? Because some developers just don't want to interact as a team. They don't have the interpersonal skills to play well in the sandbox. Part of this is because of insecurity ("Everybody else is so much better than me – and I don't want anybody knowing this."). Part is because they never learned good team-building skills. (Note that spending a full day at an off-site team-building exercise where you learn to fall backwards into your teammates arms is not really that useful). However, I have found that the majority of developers I have worked with do have the skills and motivation to become part of an integrated team.

PSP/TSP works because teams are inherently more effective than individuals. The so-called synergistic effect really works – where the combined action of two individuals is more than the sum of their individual efforts. Developing software through teamwork is nothing new, but what makes PSP/TSP effective is that it includes quantifiable metrics that allow developers, as a group, to accurately plan and track their progress. These team metrics make the team effective, because, as Lord Kelvin said back on May 3, 1883, in a lecture to the Institution of Civil Engineers:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science. [1]

My personal translation of this – which I have repeated to every PSP class I have taught – is, "If you can't count it, you can't account for it." But Lord Kelvin only mentions the need for metrics, not teamwork. If only I could find a reputable source to support teamwork:

> Two are better than one, because they have a good return for their work: If one falls down, his friend can help him up. But pity the man who falls and has no one to help him up! ... Though one may be overpowered, two can defend themselves. [2]

— David A. Cook, Ph.D.
The AEgis Technologies Group, Inc.
<dcook@aegistg.com>

References

1. Kelvin, Lord <http://www.cromwell-intl.com/3d/index.hml>.
2. The Bible. Ecclesiastes 4: 9-10, 12 (NIV)[2].

Notes

1. In case any of my daughters read this column – yes, you are "tenacious," also. And I am proud of you.
2. The biblical scholars among you might note that I skipped verse 11: "Also, if two lie down together, they will keep warm. But how can one keep warm alone?" *This* is not the kind of teamwork that I either encourage or condone among my coworkers!