Extreme Pragmatics

by Kevin Fox

Two of my favorite adages from my time in consultancies are, "You can have it good, fast, or cheap. Pick any two." and, "Nine women can't make a baby in a month." As trite as these sayings are, they came up in my head again and again as I read through the tenets of Extreme Programming.

Marketing aside (imagine building a revenue stream out of bundling together a huge batch of best practices), I feel that the concept of extreme programming has several fatal flaws when applied in real-world environments, corporate cultures where fast and cheap are just as vital to a company's survival as good, and where sometimes a baby is a baby, whether it took nine months or eighteen to be birthed.

Frankly, a strong case can be made for each of the 'rules of extreme programming' but applied together they can be dissonant not only with the organizational structures they are to be applied within, but even against each other. For example, 'simplicity' in design is often at odds with 'leaving optimization till last.'

Simplicity can be defined two ways: The first definition is objectified code consisting of elegant abstractions that allow code to be reused, easily understood, and applied in situations beyond those for which it was specifically written. Elegant code has documented interfaces, and contains contracts within it ensuring that unexpected inputs are handled appropriately.

The second definition is what could be called 'code that isn't overly engineered.' This means not creating discrete object models and frameworks when the engineer is only planning on using this code one place, and where the effort of defining an interface for the module would be greater than the final savings of doing so, considering that only one other thing will ever talk to this one.

Using the first definition, optimization is happening from the get-go. The establishment of interfaces requires the establishment of a well-thought-out architecture for the code. This, in turn, is the first and most important step of optimization. Production code should no more be written before the architecture is defined than planks should be going up on a house whose foundation is still in the architect's imagination. Under the second definition, the task of optimizing code that was written 'simply,' that is to say, to meet the need at hand, is so arduous that it would first require so much refactoring that it would make more sense to start over from scratch. Luckily refactoring is in the toolbox too, so it's apparently all part of the plan.

The reality is that all code isn't going into the space shuttle (oh that was a much better example when I thought of it last week. darn.) and that code can be over-engineered.

In a brand new adage (one of many oxymorons I can put my own name to), I say "better tools than rules." It seems that 'extreme programming' is in roughly the same place that usability engineering was ten years ago (hence Cooper's heavy stance on the issue). Usability researchers have many large and powerful tools at their disposal, from contextual inquiry to GOMS modeling, to task analysis to think-aloud testing, measuring and eliminating critical incidents. All of these are extremely valuable, but more valuable still is the discretion in choosing the right items from this Chinese menu to balance analysis with time and budgetary constraints, and knowing when it's better to go through six rounds of iterative testing with think-alouds and GOMS, and when two iterations with a heavier toolset is better called for.

I've used most of the rules, neh, 'tools' in the XP belt before, usually taught to me by people who know far more about solid coding techniques than I do. I've seen how designing with an expectation of refactoring can lead not only to complacency and lackluster code in the now, but how disastrous it can be when those outside the 'XP fold' decide that the product works well enough to ship, especially when the boat has a hole in it, and this program is the cork, however solid or not it may be.

Better to gain skill in each tool and foster confidence in those who make the decisions that you're using the appropriate toolset for the job. The best book I've ever read on the subject, surpassing even "The Mythical Man Month" is "The Pragmatic Programmer: From Journeyman to Master." It teaches many of the same concepts of XP, but from the standpoint that balance and the personal faith that you know how and when to use each tool makes that tool more powerful than it is when you throw everything you've got at the problem.

Simple as it may be, 'fast/cheap/good pick 2' doesn't perpetuate without reason. Often the ability to mitigate that quandary by leveraging several coding techniques to get the optimal combination of fast/cheap/good is better than trying to push a single 'unified best practice' dogma to try and achieve all three.