The Challenge of "Good Enough" Software

 james@satisfice.com

This article is a slightly updated version of the the original that was published in American Programmer magazine in 1995. I've changed some of my vocabulary, since 1995, but not the essence of my beliefs on this subject. These ideas are rooted in my experiences at Apple Computer in the late 80's, and Borland in the early nineties. This article was treated as a counter-culture rant in 1995. But today, the agile development movement, which exemplifies these ideas, is well established. See also Good Enough Quality: Beyond the Buzzword, and A Framework for Good Enough Testing. Find them at my website: www.satisfice.com.

Quality is an increasingly critical factor as customers become more sophisticated, technology becomes more complex, and the software business becomes more intensely competitive. But there is a powerful alternative to the orthodox, expensive, and boring methodologies that aim at the best possible quality. An alternative that is one secret to the success of Microsoft and many other companies: the discipline of *good enough* software development.

I call this discipline the utilitarian approach. It isn't simply a minimized or compromised version of the orthodox approach, any more than a city bus is an airliner without wings, or a station wagon is a race car with faux wood siding and a hatchback. To produce good enough software requires certain processes and methods that are almost universally ignored or devalued by software process formalists and by popular process models like the SEI's CMM [1]. On the other hand, even for projects that do require the utmost in methodology, the utilitarian approach can help in getting the biggest bang for the buck, and making better decisions in an uncertain world.

Views of quality

Software quality is a simple concept, at least in the textbooks. Just determine your requirements, and systematically assure that your requirements are achieved. Assure that the project is fully staffed and has adequate time to do its work. Assure that the quality assurance process is present in every phase of the development process, from requirements definition to final testing. Oh, and remember that it's absolutely critical that management be committed to quality *on the unquestioned faith that it is always worth whatever it will cost.* Otherwise, ha ha, forget the whole thing.

Software quality is not so simple in the field: where requirements shift and waver like a desert mirage, projects are perpetually understaffed and behind schedule, software quality assurance is often a fancy word for ad hoc testing, and management seems more interested in business than in the niceties and necessities of software engineering. Oh, and remember that there's lots of money to be made if you can sell the right product at the right time, or even something close enough to being right.

We often talk about quality as if it were a substance; something that could, in principle, be read from a gauge. But quality is more like an optical illusion-- a complex abstraction that emerges partly from the observed, partly from the observer, and partly from the process of observation itself. Behind the veneer of metrics and Ishikawa diagrams, quality is just a convenient

rendezvous for a set of ideas regarding goodness. As Jerry Weinberg says, "quality is value to some person [2]." While I agree with that definition, there are also some higher order patterns that we can discern in the use of the word:

- **Aesthetic view**

Quality is elegance; an ineffable experience of goodness. This view is often held by software developers, who may describe it in terms of various attributes, but cannot unambiguously describe the idea itself. We need to consider this view because it enhances morale and leads to pride in workmanship. A danger of this view is that it can become a cloak for perfectionists and underachievers.

- **Manufacturing view**

Quality is correctness; conformance to specifications. This view leaves open the whole problem of creating a quality specification and so isn't of much help in the problem of design, nor in the problem of comparing the relative importance of various nonconformances. A danger of this view is that we may create perfect products that satisfy no one.

- **Customer view**

Quality is fitness for use; whatever satisfies the customer. The problem here is that quality becomes not only subjective, as in the aesthetic view, but we also lose control of it altogether. Who are the customers? How do we incorporate their values into a product they haven't yet seen? There may be more than one customer, or the customers may not know what they want, or what they want may be some shade of unachievable. A danger of this view is that we will find ourselves chasing a will-o'-the-wisp instead of creating a product that will be happily accepted once customers do experience it.

Measurable quality factors? Mostly smoke and mirrors.

With respect to any of the above three viewpoints, we can identify certain factors that are characteristic of quality software: The *ISO 9126* standard [3] contains 21 such attributes, arranged in six areas: functionality, reliability, usability, efficiency, maintainability, and portability. The *Encyclopedia of Software Engineering* includes an excellent article on software quality factors [4], including ideas on how to measure those factors.

On the face of it, quality assurance would seem to be a matter of assessing the product for the presence of each quality factor with respect to each of the three views. But the problem is not that simple. In a particular context, some factors are more important than others. Some factors detract from others, as in the classic functionality-versus-portability conflict. For most of the factors we can identify, there are neither straightforward nor inexpensive ways to measure them, nor to compose those individual factors, once measured, into an overall quality metric. Furthermore, no matter how we tend to these factors in general, a single bug in the product may possibly negate everything else that works right.

Finally, let's face facts: our clients, whether internal or external, will never know the quality of our products. They will form a *perception* of quality and act on that basis, whether or not that matches our perception. Customer perception depends among other things on their values, skill level, past experience, and profile of use. Some of these factors we can study; none of them do we control.

So, we have some idea what software quality is, but no certain idea. We have some methods to produce it and measure it, but no certain methods. Quality is inhibited by the complexity of the system, invisibility of the system, sensitivity to tiny mistakes, entropy due to development, interaction with external components, and interaction with the individual user. In summary, creating products of the best possible quality is a very, very expensive proposition, while on the other hand, our clients may not even notice the difference between the best possible quality and pretty good quality. This leads us to three critical questions:

- How much of which quality factors would be *adequate*?
- How do we measure it *adequately*?
- How do we control it *adequately*?

One answer to these questions is to punt on the whole notion of measurable quality. Instead of using our heads to create a universal metric of quality, and then working to optimize everything against that metric, we can use our heads to directly consider the problems that quality is supposed to solve. When those problems are solved, we automatically have good enough quality.

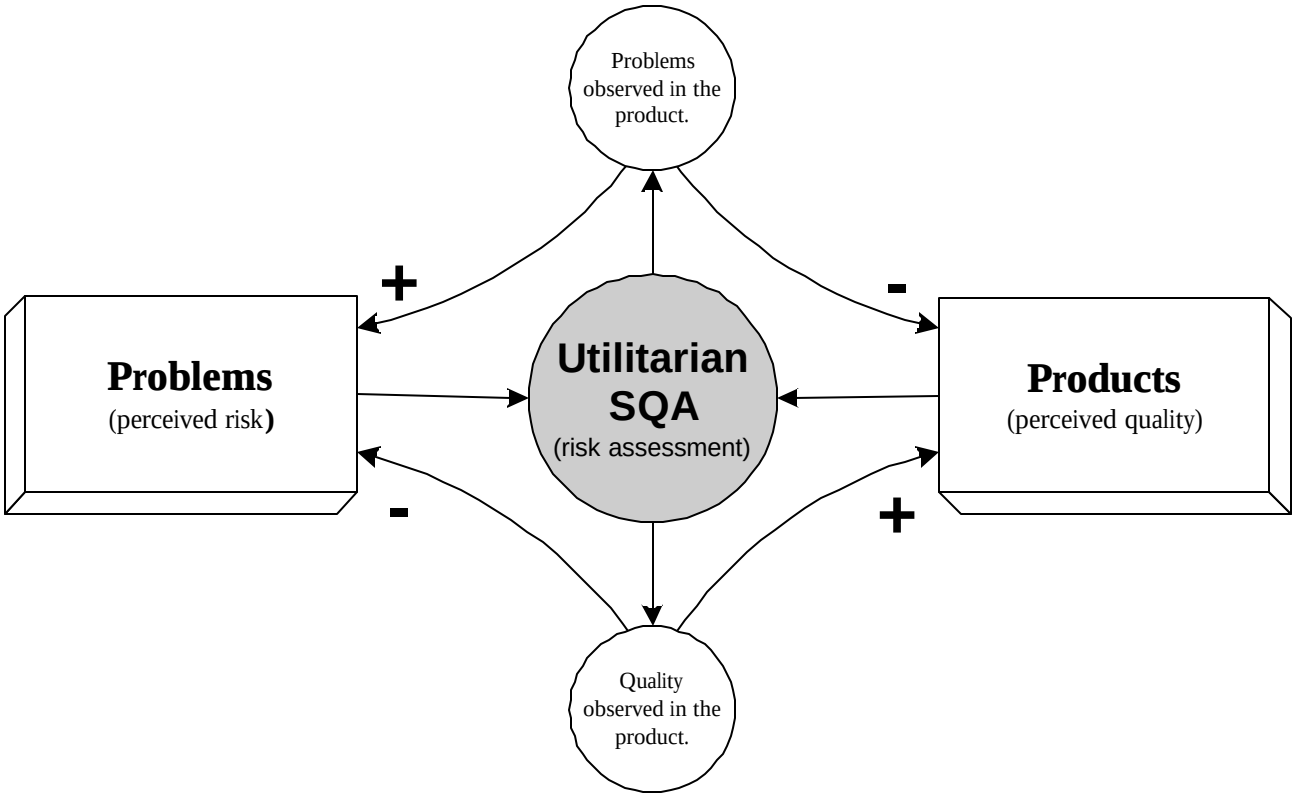

figure 1:

The goal is to reach an acceptable level of risk.
At that point, quality is automatically good enough.

The utilitarian view of quality

Utilitarianism is a nineteenth century ethical philosophy. A branch of Consequentialism, it asserts that the rightness or wrongness of an action is a function of how it affects other people.

The utilitarian view of quality is framed in terms of positive and negative consequences. The quality of something should be considered good enough when the potential positive consequences of creating or employing it acceptably outweigh the potential negatives in the judgment of key stakeholders. This view incorporates all the other views above, but replaces blind perfectionism with vigilant moderation. It applies both to projects and products. It focuses us on identifying problems and improving our problem-solving capabilities. For an interesting look at how a problem-oriented attitude is fundamentally different from a goal-oriented attitude (quality being a goal), see Bobb Biehl's book with the straight-ahead title *Stop Setting Goals if you Would Rather Solve Problems* [5].

One way of expressing the utilitarian view is to say that quality is the optimum set of solutions to a given set of problems. In these terms, the answers to the three critical questions above follow from the process of understanding the problems we face, studying the tradeoffs, and matching them with appropriate processes. We boldly cut corners. A danger of the utilitarian approach is that we may cut too many corners.

Figure 1 lays out the relationship between utilitarian SQA and quality. The SQA process examines the product and all known problems that relate to the project. When problems are observed with the product, the perception of product quality drops, while perceived risk of shipping increases. The opposite happens when quality is observed in the product. But, quality per se is not the determinant of when we ship. We ship when we believe the risks to be acceptably low-- however low or high that may be in absolute terms. At that point, quality is automatically good enough.

Our challenge lies in predicting, controlling, and measuring the consequences of creating and employing the product. In terms of *employing* the product, those consequences are mirrored in product quality. In terms of *creating* the product, those consequences are mirrored in the quality of the process, staff, and resources. We can certainly improve quality by operating directly on some chosen metric, doing whatever is necessary to drive that metric in the desired direction, but I assert that in doing so we lose sight of the full spectrum of the product, project, and customer. Instead, by working the consequences side of the mirror, we can see the whole problem.

For example, we might decide that a reasonable quality metric is the number of known defects in the product. If we follow the orthodox approach, we would improve quality either by preventing defects or by fixing them before we shipped. Either way, we would minimize the number of defects in order to maximize quality. If we follow the utilitarian approach, however, we would examine the consequences of each problem, and decide on a case-by-case basis which were important to fix. The quality metric would then either take care of itself or else become irrelevant. Prevention is a concern, but not blanket prevention, only prevention of important problems, and only prevention to the extent necessary.

To be good enough is to be good at knowing the difference between important and unimportant; necessary and unnecessary. The orthodox approach glosses over such

considerations, or tends to translate any discussion of them into a battle between Good Engineering and Bad Management.

Apple shipped the first version of Hypercard with about 500 known bugs in it, yet the product was a smashing success. The Hypercard QA team chose the *right* bugs to ship with. They also chose the right qualities. I'm not sure how many thousands of bugs were shipped with Windows 3.1, but you can bet that it was at least several. I was working at Apple when Macintosh System 7.0 shipped, and it went out with thousands of bugs. Successful software quality managers will tell you, it isn't the number of bugs that matters, it's the *effect* of each bug.

You know, no matter what other approach to quality we talk about, we all use the utilitarian approach. We all make judgments of risk, whether we hide that in terms of some quality metric like failure density or whether we avoid all explicit metrics and processes. The issue is how effectively we assess and control risk. Call me crazy, but I believe that we will be better judges of risk if we admit that's what we're doing and learn how to do it directly, rather than indirectly through psychological games like "six sigma" and slogans like "quality without compromises."

From problems to products: the double-cycle project model

In order to explore the means by which we can create good enough software, let's start by examining the simple project model in figure 2. The model consists of the following elements:

- **Problems**

Problems are the motivators of the project. In the absence of problems, there would be no project. Problems can be defined broadly as the difference between an actual and a desired state, or more narrowly as some work to be done. Either way, we can also characterize them in terms of an ecology of causes and consequences. In other words, problems represent risk. A major part of utilitarian software development is choosing which problems to avoid, which to accept, and which to solve. That process is one of risk management.

Problems, in this model, include project problems as well as product problems. To do a project well enough is to be left with an acceptable set of problems at the end.

- **Products**

Products include the total output of the project, including reusable by-products, ongoing support, experience, the ship date, and even the project team itself. At Microsoft, there's a saying that the project has two deliverables: the product, and the team. If we are to evaluate projects to assess whether they are indeed good enough, our notions of failure and success must include all these factors, not merely those qualities that are intrinsic to the software.

A product is the manifestation of the solutions to the problems that motivate the project. Product quality emerges from those elements, intrinsic to the product, that when combined with extrinsic factors (such as an actual customer) solve a problem. Thus, as in figure 1, there is a loosely inverse relationship between problems and product quality. Higher quality leaves us with fewer problems than lower quality, all other things being equal. But, is the difference between “high” and “low” quality really significant? Are all other things really equal? These questions easily get lost when we focus on quality for its own sake.

- **Project**

The project is the entire means by which risks are managed and products are created. The arrows in figure 2, connecting problems, project, and product, denote a broader dynamic than those in figure 1. The arrow from project to problems refers to the creation or identification of new problems by the project. The arrow from project to product refers to product development. Note that a single project action may simultaneously enhance the product and create problems. The arrows back to project, from either side, indicate that both problems and products are not only outputs of the project, but also inputs to it.

This double-cycle project model is the simplest way I can think of to visualize basic project dynamics. We start with nothing but problems, and we finish with a new more livable set of problems and a new set of products and by-products. The project itself evolves along with everything else, though, leaving us at ship-time with a capability to produce that is possibly better than when we started or possibly worse. The "good" of a project should be judged by the total output of all that transpires: the problems, solutions, and resulting capability.

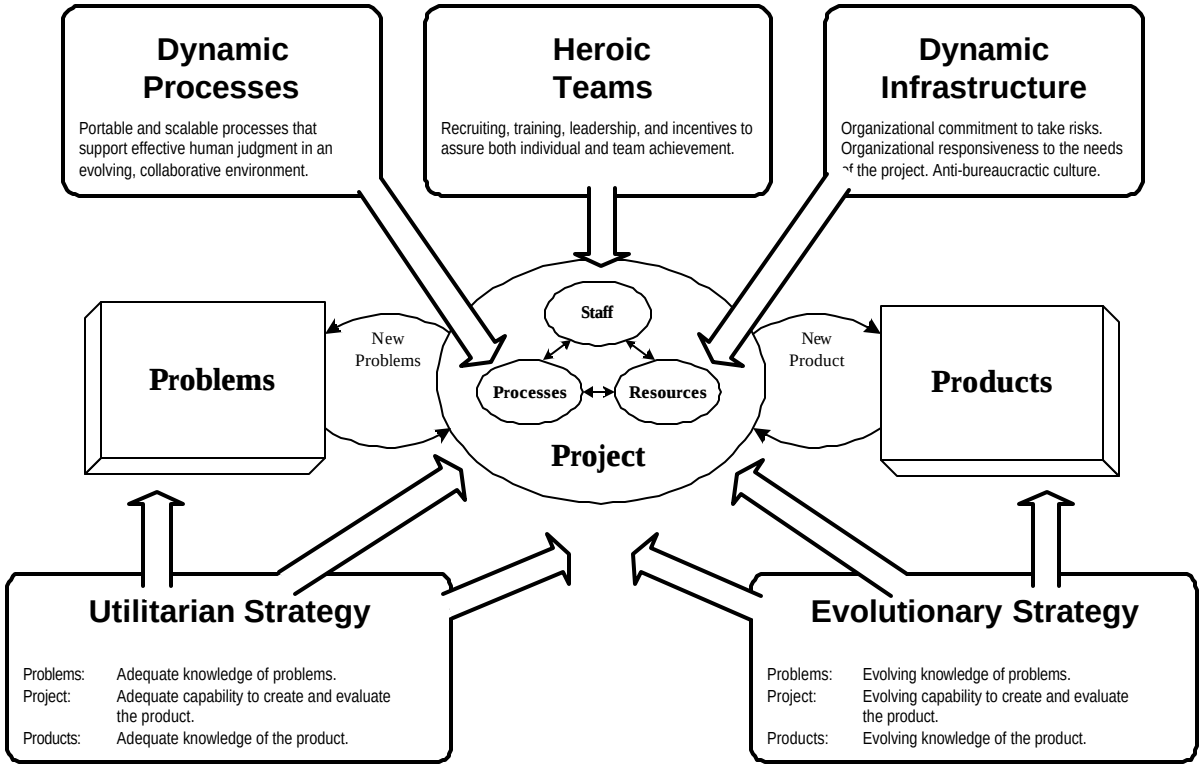

figure 2: The double-cycle project model and keys to good enough development

For our purposes, the principal elements of projects can be summarized as staff, processes, and resources:

- **Staff**

The project staff is the agent that solves all the hard problems. They employ processes to turn those problems into products. They also solve problems even without any discernible processes

or direction. I call this "process heroism." People are thus the most versatile and critical part of the project.

- **Resources**

Resources include anything that money can buy, such as brute labor, software, hardware, office space, and information. Resources are important to note chiefly because they support the project's staff and processes. They also can create problems, as in the need to maintain equipment; or they can contribute directly to products, as when an application framework is used as a foundation for development.

- **Processes**

There are many definitions of process. I'm using the term to mean a pattern for solving a problem. It may solve a problem in conjunction with a tool or in conjunction with a person. The result of the all the problem-solving is that a product is created. Note that in many industries, this would be a very roundabout way to conceive of product development. After all, we wouldn't say that the guy making burgers at McDonald's is solving problems, but rather that he is following a defined process of sandwich construction. It's only because software is so purely a construction of the mind, and that each software product is such a specific and unique solution to a very particular problem that we are driven to consider problem-solving as the central activity of software development.

Processes include project lifecycles, written and unwritten plans, agreements, procedures, meeting schedules, and basically any strategy or methodology that a team or individual may employ. The utilitarian approach takes a broad and flexible view of process. In terms of the double-cycle model, processes are distinct from staff and resources. Processes are just concepts. They only manifest solutions in concert with staff or resources. Hence, process *usability* is a prime concern for us.

The five key process ideas (KPIs) of good enough software

In my experience, the key elements of creating good enough software are a utilitarian strategy, an evolutionary strategy, heroic teams, dynamic infrastructure, and dynamic processes. These elements relate to the project model as shown in figure 2. Let's look at each of these elements in turn.

- **Utilitarian Strategy**

The utilitarian strategy applies to problem, projects, and products, as in figure 2. The term is one that I've coined out of necessity (or possibly ignorance, as I just haven't found a suitable alternative). It refers to the art of qualitatively analyzing and maximizing net positive consequences in an ambiguous situation. It encompasses ideas from systems thinking, risk management, economics, decision theory, game theory, control theory, and fuzzy logic. For the purposes of this paper, let's explore systems thinking and risk management in a bit more detail.

Systems thinking is about understanding and reasoning with system models. It emphasizes qualitative rather than quantitative factors, and dynamic rather than static interpretations. Tarek Abdel-Hamid and Stuart Madnick have applied systems thinking to software engineering in their fascinating book, *Software Project Dynamics* [6]. In the book, they hypothesize a set of dynamics and use them to construct a computer simulation of a typical software project. In the realm of project estimation, for instance, some of those dynamics are that

productivity is affected by perceived project status, which is in turn affected by the estimation of the magnitude of the project and the way that estimation is applied. One of their discoveries from this experience is that *a different estimation creates a different project*. In other words, goodbye static thinking. There is no right or wrong project estimate, just an integrated, dynamic estimation process.

Systems thinking feeds nicely into risk analysis. Why risk analysis? Here's why. We are engulfed by uncertainty in every second of software development. When was the last time you were on a project that could not fail? If it really couldn't fail, weren't some of the resources or staff for the project more sorely needed elsewhere? In most situations, if we are on a project that can't fail today, tomorrow our project will be expanded, or our resources will be stolen. Efficient software development necessitates risk taking: The real question is whether we take *calculated* risks or accidental risks.

One way to avoid accidents is through a habit of integrated, structured risk management. By integrated, I mean that every strategy we use to create a positive result (e.g. adding a feature) has an associated risk management strategy (e.g. testing the feature). I've observed this habit in many companies but it was especially apparent in the Borland Languages division, where I worked for several years. We identified risks all through the project planning process, we reviewed them, we put risk reduction strategies in place, and we used postmortems and post-ship metrics to calibrate our judgment. We never quantified our risks, however, because we felt that such quantification would be impractical and misleading.

[See Heuristic Risk-Based Testing, by James Bach, for more on this. – ed.]

A systematic approach to risk uncovers areas where our expertise, processes, or resources are weak. Then, by applying the other four keys, below, we can take corrective action.

- **Evolutionary strategy**

An evolutionary strategy, applied either to problems, projects, or products, alternates observation with action to effect ongoing improvement. On the project level, this means ongoing process education, experimentation and adjustment, rather than clinging to a notion of the One Right Way to develop software. On the product level, this means planning and building the product in layers, which allows concurrent design, coding, and testing. It also provides opportunities to respond to changing requirements or unforeseen problems. On the problem level, it means keeping track of history, and learning about failure and success over time. Here are some of the elements of using the evolutionary approach:

- Don't even try to plan everything up front.
- Converge on good enough in successive, self-contained stages.
- Integrate early and often.
- Encourage disciplined evolution of feature set and schedule over the course of the project.
- Salvage, reuse, or purchase components where feasible.
- Record and review your experience.

The evolutionary approach is popping up everywhere in the recent literature, especially in relation to Rapid Application Development [7], but evolution is discussed mainly in regard to the project lifecycle. What I'd also point out is that we can take an evolutionary view of our people, processes, and resources. I don't mean the artificial evolution of the CMM maturity levels, nor the ideological frenzy of TQM, but the natural evolution that results from calculated

trial and error; from observing sources of pain within our projects, and working to resolve them. This concept dovetails completely with utilitarian strategy.

- **Heroic Teams**

For some reason, the most fundamental key to good enough development also seems to be the most controversial. There is a strong disdain, among many methodologists, for the very word "hero". I'm not sure why that is, since evidence supporting the role of heroes in computing is just a shade less compelling than evidence supporting the role of electricity. I think it's because there are several definitions of hero. The definition I use is based on the work of Joseph Campbell [8].

I say a hero is someone who takes initiative to solve ambiguous problems [9]. Everyone who's ever written a single line of useful code is therefore a hero! Why? Because every line of code represents a fundamentally ambiguous problem; and no matter how intrusive development managers can be, they can't stand behind each and every programmer to personally authorize code as it's written. Programmers must exercise initiative.

So, when I speak of heroic teams, I'm not speaking of the Mighty Morphin Power Programmers. Heroic teams consist of ordinary skillful people working in effective collaboration. The art of building heroic teams is often called Peopleware [10]. The idea includes savvy recruiting and mentoring, providing ample office space, minimizing interruptions, co-locating teams that work together, and maintaining a compelling shared vision of the product.

Orthodox methodology doesn't fly with good enough software companies for one reason more than any other: it's boring. Bored people don't work hard. They don't take initiative, and they avoid ambiguous problems instead of tackling them with gusto. Take care to create an exciting environment that fosters responsible heroism, and great software will follow.

- **Dynamic Infrastructure**

Dynamic infrastructure means that the company rapidly responds to the needs of the project. It backs up responsibility with authority and resources. Dynamic infrastructure provides life support for the other four key process ideas. Some of its elements are:

- Upper management pays attention to projects.
- Upper management pays attention to the market.
- The organization identifies and resolves conflicts between projects.
- In conflicts between projects and organizational bureaucracy, projects win.
- Project experience is incorporated into the organizational memory.

The opposite of this is wasteful bureaucracy and power politics. Achieving a good dynamic infrastructure requires a heroic team of managers. At Microsoft, Bill Gates is the anchoring hero. In the absence of a dynamic infrastructure good enough software can still be achieved, but mainly in the short run, and on the backs of an insanely committed project team. In the long run, the other four key process ideas will falter if saddled with the weight of unresponsive bureaucracy.

- **Dynamic Processes**

Dynamic processes are those that change with the situation; ones that support work in an evolving, collaborative environment. Dynamic processes are ones you can always question

because every dynamic process is part of an identifiable meta-process. If we lose sight of its meta-process, the dynamic process become static and brittle. Take testing for instance. If we lose sight of the meta-process to which testing belongs (which is usually risk management) then the testing process becomes a blind reflex activity. Thus, the ability to discuss process and meta-process is a basic skill of utilitarian software teams.

Three other important dynamic process attributes are portability, scalability, and durability. Portability is how the process lends itself to being carried into meetings, shared with others, and applied to new problems. Scalability is how readily the process may be expanded or contracted in scope. A highly scalable process is one that can be operated by one person, manually, or by a hundred people, with tool support, without dramatic redesign. Durability is how well the process tolerates neglect and misuse.

A final note about dynamic processes: the most dynamic process is *no process at all.* One popular and effective technique is to clarify project roles so that everyone knows who is supposed to solve what kinds of problems. Then, if the team works well together, peer-to-peer delegation can replace most orthodox processes. For example, if we need to ship the product, we can assign someone to oversee the process of signoff. That person then *embodies* the signoff process. I've seen this approach work pretty well even on 100-person projects. This "clarify and delegate" strategy is a good basis for all other dynamic processes.

Good enough to be the best

Microsoft does things right enough, and enough of the right things right, to be a lasting success. It's true that their processes of software development are more mature than they were some years ago, but they haven't become mature by adopting maturity as a goal. Rather, in the Microsoft culture, the top priority, the guiding principle, is *solve the problem*. Because they face different problems as a multi-billion dollar company than they did as a multi-million dollar company, they have also changed their processes.

The Good Enough KPIs, above, are particularly vital when engineering software in a market-driven environment, where there is no pre-arranged contract with the customer, and competitors vie to lure your customers away with their own offerings. But they are nevertheless very applicable in the contract-driven realm.

I could list a number of process ideas, from prototyping to beta testing, but the specific processes are less important than understanding and applying the five KPIs. In summary, if we develop and empower teams who use dynamic processes within a dynamic infrastructure to employ utilitarian and evolutionary strategies, then we will produce good enough software. Good enough, even, to be the best.

Bibliography

[1] Paulk, Mark C., Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, Capability Maturity Model for Software, Version 1.1, CMU / SEI-93-TR-24, Software Engineering Institute, 1993

[2] Weinberg, Gerald M., Quality Software Management, v. 1 Systems Thinking, Dorset House, 1991

[3] International Organization for Standardization, ISO / IEC 9126:1991(E) Quality Characteristics and Guidelines for Their Use, ISO / IEC, 1991

[4] Marciniak, John J., editor, Encyclopedia of Software Engineering, v. 1, pp 958-969, John Wiley & Sons, 1994

[5] Biehl, Bobb, Stop Setting Goals if you Would Rather Solve Problems, Random House, 1995

[6] Abdel-Hamid, Tarek K., Software Project Dynamics: an Integrated Approach, Prentice-Hall, 1991

[7] Martin, James, Rapid Application Development, Macmillan, New York, 1991

[8] Campbell, Joseph, The Hero with a Thousand Faces, Princeton University Press, 1949

[9] Bach, James, Enough About Process: What We Need are Heroes, IEEE Software, March 1995.

[10] DeMarco, Tom, Timothy Lister, Peopleware: Productive Projects and Teams, Dorset House, 1987