Essentials of Software Process

Hakan Erdogmus

In “The Challenge of Good Enough Software” (American Programmer, Oct. 1995), James Bach introduced the utilitarian view of software process. Well over a decade has passed since this article was published, but I find myself increasingly subscribing to this idea. It's likely to dominate the post-big-process, post-agile, post-model-driven-development era to come. Whatever is emerging from grassroots and craft-oriented process trends colliding and converging with traditional research- and engineering-based approaches has a lot to do with the utilitarian view. Here, I attempt to deconstruct and extend this view in terms of seven essential characteristics that mutually reinforce and balance each other.

I define “process” like Bach does (see www.satisfice.com for an updated version of the 1995 article) but with a slight adaptation. A process is a collection of patterns and activities for solving a class of problems. We can place process trends inside a triangular map according to their emphasis relative to three aspects, represented by the vertices: people, technology, and rigor. Plan-oriented, engineering, and research-based approaches tend to view software as a rigid artifact that’s difficult to change, so they stress technology and rigor aspects over the people aspect. Evolutionary approaches tend to view software development as an organic, skills-driven technical activity, so they stress people and technology over rigor. But this scheme of positioning process approaches is rather rough. A more complete scheme requires a dissection in more than three dimensions. The new dimensions comprise the seven essential characteristics mentioned earlier: human-centricity, technical orientation, discipline, pragmatism, empiricism, experimentation, and value orientation.

Human-centricity

Human-centricity in a nutshell is peopleware, after Tom DeMarco and Tim Lister’s famed book of compelling narratives. People’s role is so evident that I first wondered whether to bother listing it. On the other hand, this role is still notorious for receiving lip service. Clearly, software development is an intellectual endeavor, and serious software development is a team activity. A human-centric process emphasizes collaboration, recognizes the importance of effective leadership, and caters to the needs of creative professionals who take pride in their work.

Human-centricity requires understanding what motivates knowledge workers and how they perform their work and interact with others. It also demands an appreciation of how non-software professionals perceive software development, what customers want, and, when human users are involved, how people use the software. Thus, human centricity is about valuing soft skills as much as hard skills.
But can peopleware singly guarantee success?

Technical orientation

It’s rather hard to get useful software by throwing together a bunch of people with wonderful soft skills but not enough individual technical competencies to do the job. Here, competence in both fundamentals and software technology matters. Fundamentals include base knowledge of such things as algorithms, data structures, programming paradigms, design patterns, testing, requirements, user interfaces, and reference architectures. Technology includes languages, tools, environments, frameworks, platforms, and other relevant infrastructure. A technically oriented process fosters possession and acquisition of teamwide competence in fundamentals and technology.

The trademark of any sensible process is continuous attention to technical excellence, in which a solid infrastructure fit for the purpose is primary. A technically oriented process leverages tools and automation so that people are liberated from performing mechanical, error-prone tasks and can focus their energies on more intellectually demanding ones.

A technical orientation perceives as disingenuous failure to automate mundane, repetitive activities, such as the build or deployment process (see “Software Builders,” IEEE Software, May/June 2008). But it also advocates not aspiring to mechanize activities that require creative, skillful human intervention. The definitive artifacts of a technically oriented process are the ones subject to human manipulation at the lowest abstraction level.

Technical orientation adds meat to human-centricity, but doesn’t any serious, sustainable, scalable business need managerial and procedural rigor?

Discipline

Many view a process’s methodical, managerial, and operational dimensions as the highway to predictability, sustainability, and repeatability. These dimensions constitute the rigor aspect of a process. But discipline isn’t equivalent to formal procedures, top-down management, maturity levels, unrelenting compliance, and a forever-ordered, sterile development. Discipline is the systematic use of synergistic practices appropriate for the context. It’s about robust structuring of workflow and division of labor as well as accepting, expecting, and accordingly managing chaos.

A disciplined process applies its chosen technical practices consistently, if not blindly. These practices may include coding standards, frequent builds, requirements modeling, architecting, informal design, code inspections, regressive unit testing, and exploratory testing. A disciplined process also consistently applies management practices to coordinate activities, share knowledge, and control product artifacts. These might entail business analysis, requirements management, documentation as needed,
time-boxed iterations, phasing, estimation, progress tracking, and retrospectives.

Discipline is essential, but it often needs to be moderated for specific needs.

Pragmatism
Pragmatism is the moderating counterforce to discipline. It’s as important as discipline for making a process tick. To be pragmatic, a process activity should be continuously feasible, adapted to your context, scalable, and compatible with your organization’s changing culture and risk-taking preferences. A process activity should serve a clear purpose, whether it’s to meet customers’, project teams’, or users’ needs or to satisfy regulatory requirements. So pragmatism is just common sense, which Bach’s utilitarian view advocates.

It’s possible to be pragmatic with process choices, but how does one know that the decisions, no matter how sensible they seem, are paying off as expected? Extra elements need to be in place in order to take calculated risks with a pragmatic but disciplined approach.

Empiricism
This brings us to empiricism—a concept not too popular with grassroots circles because of its traditional association with “big” process, quantitative management, and academic research. Empiricism at its heart is supporting decisions through evidence based on data, both observations and measurements. By observations I mean occurrences that we can simply record. By measurements I mean things we can count, calculate, or quantify. Measurements have values, whereas observations have descriptions, possibly including contextual information. Observations provide deeper insight in areas in which measurements serve only as proxies for other constructs. Each kind of data has a place, and empiricism entails collection and use of both kinds.

The right data give a team the ability to preserve knowledge, connect actions to consequences, and relate inputs and outcomes. This awareness in turn helps the team figure out when it’s failing or succeeding or whether it’s improving or deteriorating, sense why things are going the way they are, and take corrective, preventative steps.

Empiricism increases the quality of decision making and the value of planning. Management activities—estimation, quality assurance, release planning, resource allocation, progress tracking—become more meaningful and better grounded through empiricism. As Robert L. Glass states in Facts and Fallacies of Software Engineering (Addison-Wesley, 2003), “managing in the presence of data is far better and easier than managing in its absence.”

Empiricism also helps us test our theories about what works and what doesn’t more objectively than with just gut feeling alone, hence fostering learning.
Experimentation
To learn, we need experimentation. Experimentation is possible in an environment in which alternative solutions to problems, viability and effectiveness of ideas, and hypotheses about prospective actions can be tested safely and cheaply to resolve uncertainty.

The concept of experimentation should be familiar to many. The Extreme Programming idea of a spike to assess a technical solution’s feasibility is a form of experimentation. Prototyping, simulations, role playing, split runs, and early usability evaluations constitute other examples.

In the absence of uncertainty, experimentation is unnecessary. But in its presence, an unfavorable answer is obviously a possibility. Without a realistic expectation of a negative outcome, learning can’t occur. Thus critical in effective experimentation is the acceptance of occasional, transient failure. Learning happens with deliberate trial and error. A process that’s intolerant and punishing of unfavorable transient results will discourage experimentation.

We have almost come full circle. Systematic experimentation is actually a value-generating activity.

Value orientation
Value orientation goes hand in hand with pragmatism. As such, it’s the next characteristic that closely relates to Bach’s utilitarian approach. John Favaro provides a stereotypical manifestation of value orientation in “When the Pursuit of Quality Destroys Value” (IEEE Software, May 1996).

Value orientation is parsimonious purposefulness. Software projects operate under scarce resource and schedule constraints. Decisions and activities are subject to not only availability of time and money but also the unrealized benefits of future opportunities. In such environments, it’s critical to reason about what ultimately matters, to whom, and why. Such reasoning entails resourceful consideration in cost-benefit terms of the trade-offs between productivity and quality, repeatability and flexibility, customer/user and employee satisfaction, predictability and creativity, authority and autonomy, and short- and long-term goals.

The bottom line often constitutes the most important component of an activity’s value to stakeholders. Early realization of benefits and deference of costs increase economic value. Obviously, so does minimization of overhead. According to financial theory, under conditions of uncertainty, economic value also increases through waiting, learning, flexibility, and risk management. This is a critical but less obvious component of value orientation. Value orientation thus implies perceiving the contribution of process activities in a new light of value maximization under uncertainty.

The seven essential characteristics I just listed aren’t orthogonal. Rather, they resemble suspended blobs connected with invisible forces: you tug at one, and the others move in every direction. They help highlight the strengths and inevitable weaknesses of process approaches.

If your perspective differs, write me at hakan.erdogmus@computer.org.

CALL FOR ARTICLES
Software Development for Embedded Systems

This special issue will share proven embedded-systems ideas and experiences from all phases of the development life cycle and from a range of industry domains. Of specific interest is how practices and methods from enterprise and desktop computing can be transferred into the embedded domain. Topics of particular interest include:

- Design methods (so-called DFX)
- Domain-specific languages, including hardware related issues
- State of the practice in modeling, design, verification, and validation
- Communication protocols and middleware
- Modeling of quality attributes (for instance, where safety meets security)
- Concepts for maintenance, robustness, and remote diagnosable architectures
- Schemes for remote software maintenance and testing

Publication: May/June 2009
Submission deadline: 1 Nov. 2008
Guest Editors:
- Christof Ebert, Vector, christof.ebert@vector-consulting.de
- Jürgen Salecker, Siemens, juergen.salecker@siemens.com

Complete call: www.computer.org/software/cfp3.htm
Author Guidelines: www.computer.org/software/author.htm
Submission details: software@computer.org

Software www.computer.org/software