A Methods "Discussion/Debate":

Method Agility
Or
What's a Methodology For?

Scott P. Duncan
Quality/Process Improvement Consultant
SoftQual Consulting
901 Douglas Drive
Ellerslie, GA 31807
(706) 888-5021 (cell)
sduncan@computer.org, sduncan@acm.org

What's the "right" method to use for a software development project according to all the "best practices" advice? How would you answer this question or is this really a sensible question to ask? Many folks advocating "lite" or agile methods would suggest there is no "best" practice you can apply across the board. Articles and columns in IEEE and ACM publications have addressed this very point. On the other hand, advocates of "disciplined" methods (to use the term Boehm and Turner have in their book on agile and more formal methods) would say there is vast industry experience pointing to "best practices."

This paper is about beginning the process of answering some methodology related questions, beginning with the question:

Why have methods?

Aren't skilled, dedicated people more important than all the tools and process documentation you can muster? Indeed, if we cannot rely on responsible people to do their best to achieve product quality and other project goals, what can we rely upon? Do we really expect methodologies to cause quality to occur in software development regardless of the talent involved? If not, what's a method for then?

Various formal quality and software approaches seem to be based on manufacturing models. Classic quality principles applied to physical products exist because of the tremendous cost impact to manufacturing if changes occur or design mistakes are discovered after tooling and assembly-line processes are prepared. But software isn't manufacturing; it's design engineering. The only "manufacturing" in software is when a finished asset of software components that have been "built" into a full system get copied to disks, CDs, etc. for distribution. Since software changes more and addresses more "unknowns" than any other product domain, how can any single method be "right," especially those that attempt to get it right up front and freeze changes for months?

This raises the question then of:

What should methods do for us?

One position taken implicitly is that software methodologies exist to control development staff who feel their work is an art driven by creative inspiration, not a step-by-step, cookie-cutter process driven by formal principles. For the former, methodologies are there to prevent incorrect results from being produced, just as quality systems in manufacturing exist to guarantee requirements are met, rework is minimized and schedules are made predictable.

However, shouldn't methodologies be viewed as being there to help do the job well not (just) prevent people from doing it badly? The latter should be achieved by defining, applying and monitoring the methodology effectively. And, certainly, a methodology should be easy to use correctly – at least, easier than doing it badly or not at all – and be viewed as an integral part of the "real work." (One of the problems in software development, of course, is getting agreement on what the "real work" is.)

A methodology should also support feedback on how it can be improved. Grumbles, complaints and ideas from those following it should not be ignored, forgotten or treated as just so much developer generated "background noise." ("You know those developers. They always complain about the schedule being too tight. And they're always late and over budget anyway.")

Perhaps it would be better to think of methodologies as "the line down the middle of the road." You can cross that line and drive on the other side of the road almost without consequence if you can see far enough down the road. And, even if you can't, crossing the line doesn't mean instantaneous disaster. However, where the line is a dash, suggesting you can cross it to pass another, that doesn't guarantee that you can or should in a given situation. So, the "line down the middle of the road" is simply guidance or a warning that risk exists if the line is crossed and consideration needs to be given to what that risk means. Why not view a methodology as the same thing?

(Of course, staying on your own side of the line doesn't guarantee safety, either, does it? There's always the person coming the other way who is willing to take that risk of crossing the line and then there are other road warnings and dangers. So…)

What do we need to do?

It seems software projects fare better if they accomplish two things through the methods they employ:

- Expand the bandwidth of communication and
- Be open to (prepared for) change.

Nothing improves communication (especially about change) more than regular, direct contact with the customer, within the development team and from the software itself. So how do we get all of these?

The Customer

One way is to have the customer as close to (or on) the development team as possible. They can then participate directly in specifying needs, defining acceptance criteria, regularly reviewing functioning software, and providing rapid feedback on what is and is not needed while learning more about how to achieve those needs.

As the distance between customer and developer grows - physically and through less frequent feedback - the communication bandwidth shrinks. It takes an increasingly greater commitment of effort to overcome that distance. At some point, the distance can increase to where people feel there is little that can be done to overcome it. In the face of that, passing very formal documentation back and forth replaces more effective communication. Indeed, when this happens, true collaborative communication is replaced with various forms of negotiation formality.

(Interestingly, perhaps the lowest bandwidth form of communication – formal written documentation – is considered one of the most highly regarded and expected forms of communication in "disciplined" methodologies and quality systems.)

The Team

All members of the development team should also be located as close together as possible so regular face-to-face communication is possible. In this way, brief, daily meetings of the entire team can be held, so everyone knows what everyone else has done and plans to accomplish that day.

Employing an iterative development lifecycle and committing to meet each near-term iteration plan contributes toward the least delay in or confusion over what is to be delivered to the customer. The team can then deliver a functioning release every 3-4 weeks and get customer feedback on what does and does not meet needs.

The Product

Finally, and related to the last point above, seek frequent product feedback by building and testing the system (at least) daily to know immediately what problems exist and what progress is being made. Then, by delivering a functioning release every 3-4 weeks, you can get useful customer feedback to know if you are meeting customer's (evolving) expectations. All this requires an environment, technology and team commitment to support iterative development and rapid turnaround, which contribute to being open to change.

These ideas get addressed through the:

Agile Methods Principles and Their Importance

The agile methods community drafted a set of four principles several years ago, which they felt would guide them in achieving goals like those noted above (i.e., expanded communication bandwidth and responsiveness to change):

- Individuals & interactions (are valued) over processes & tools
  (which produces an emphasis on small teams of motivated, supported individuals.)
- Working software (is valued) over comprehensive documentation
  (which drives "… early and continuous delivery of valuable software" and a desire to convey information face-to-face as much as possible.)
- Customer collaboration (is valued) over contract negotiation
  (which leads to the goal of having the customer maintain constant involvement with the project. The contract is then about how developers and client will work together.)
- Responding to change (is valued) over following a plan
  (which recognizes that it is really only possible to be sure about a detailed plan for the next several weeks (not months or years), within which, team momentum and commitment can be maintained at a steady, enthusiastic pace.)

Note, the "conclusions" drawn after each principle are mine, not the Agile Alliance's (www.agilealliance.org), so consider them accordingly. However, those employing agile methods do seem to have some:

Critical Ideas and Practices

First, it has been appropriately said, that software development is like driving a car, not aiming a bullet. That is, constant corrections are made in near-term time frames to keep the vehicle moving where you want it to go. This is in contrast to a lot of preparation before the trigger is pulled since you lose control over the bullet after that and hope you have done everything right up front to make sure it gets where you want it to go. Many "disciplined" approaches seem to suggest the latter is the model that applies and, for automated, manufacturing assembly lines, it likely is.

Other ideas and practices include:

- A real user on the team and available full-time to direct "requirements specification" and develop acceptance tests.
- A steady, predictable work pace (40-hour week) with team "ownership" of all work.
- Writing tests first based on (and as an addition to) the requirements, then designing and developing code to pass the tests, employing simple design and removing complexity when discovered.
- Frequent (daily) builds (and integration testing) of system components, leading to frequent (3-4 week) delivery of functionality, to get feedback on what works and does not as fast and as often as possible.

On the other hand:

Agile methods may not work if…

In place of regular dialog, customers, developers and/or stakeholders require large specifications, infrequent adjustments to a project plan, or fixed contracts with infrequent delivery of software. This tends toward the "big bang" approach to software development where it can be months (even years) before people can see if anything "works" or is really heading in the right direction. It presumes the ability to get the requirements "right," then get the design and architecture "right," then … in a sequential manner, avoiding (or heavily controlling) change along the way because of the large commitment in effort it takes to get to each milestone.

Agile methods may also be a bad choice if, at the technical team/management level, long hours are expected to "prove" one's "commitment" to the work and coding prowess is valued over communication. The "win one for the gipper" approach to projects imposes moral approbation on people when schedules, budgets and quality targets (probably not reasonably arrived at) at not met since the people obviously were not "doing their best." The "if you can't do it, I'll find someone who can" approach to cost and scheduled negotiation usually comes with such an approach to software quality.

Since agile methods expect close team communication, they may also not work as well if the development staff will exceed 15-20 people and the available technology prevents (at least) daily build-test cycles. Other things in the environment that discourage cooperation could make agile methods a poor choice, e.g., cubicles with no easily accessible common area, the team being separated on multiple floors or locations, distractions to and interferences with communication.

IEEE Recommended Practice

There is also an IEEE Software and Systems Engineering Standards Committee project (IEEE 1648) to develop a Recommended Practice to explain the critical concepts/expectations of agile methods to a (potential) customer so the customer would understand:

- what to expect from a development organization using such methods,
- what obligations they would have as a customer, and
- What issues occur when agile methods meet more formal process expectations.

A Recommended Practice is not quite as strong a document as a full standard, using the word “should” rather than “shall” (a full standard) or “may” (a guideline). However, an RP’s wording could be used in a contractual situation.

The intent of the Working Group is to end up with a document that:

the agile community would look at and say, "Yeah, that's what we’re about, what we expect from our customers, and what we commit to do in working on a project."

and

the acquisition/customer community would look at and say, "Okay, I understand what an agile methods developer would expect from me and what I could expect from them and I am comfortable (or not) with that."