Crossroads
The ACM Student Magazine

The Many Dimensions of the Software Process

By ***Sebastián Tyrrell***

The software process is a becoming a big issue for companies that produce software. As a consequence, the software process is becoming more and more important for permanent employees, long-term practitioners, and short-term consultant in the software industry.

A **process** may be defined as a method of doing or producing something. Extending this to the specific case of software, we can say that a **software process** is a method of developing or producing software.

This simple definition shows us nothing new. After all, all software has been developed using some method. *"Try it and see"* is a perfectly valid process. However, as highly trained consultants we would be more likely to refer to it as a *heuristic approach*.

We can also see that a *process* is nothing without the *something* that gets developed - in our case the software itself - that the process *produces*. Again, this is nothing new. Every process produces something.

Two extremes, and a whole spectrum of views between them, characterize most professionals' view of the process. One extreme represents the `hero tendency'. This view says concentration on the process achieves nothing more than adding bureaucratic overhead. It gets in the way of the real work, which is programming. The

second says that process is all. So long as you have and follow a good process the software will almost write itself.

This guide will attempt to steer a balanced course through these troubled waters, identifying where, when, and how a greater emphasis on process issues may be of greater help, and maybe more important, where they would not.

The importance of process

In the past, such processes, no matter how professionally executed, have been highly dependent on the individual developer. This can lead to three key problems.

First, such software is very difficult to maintain. Imagine our software developer has fallen under a bus, and somebody else must take over the partially completed work. Quite possibly there is extensive documentation explaining the state of the work in progress. Maybe there is even a plan, with individual tasks mapped out and those that have been completed neatly marked - or maybe the plan only exists in the developer's head. In any case, a replacement employee will probably end up starting from scratch, because however good the previous work, the replacement has no clue of where to start. The process may be superb, but it is an *ad-hoc* process, not a *defined* process.

Second, it is very difficult to accurately gauge the quality of the finished product according to any independent assessment. If we have two developers each working according to their own processes, defining their own tests along the way, we have no objective method of comparing their work either with each other, or, more important, with a customers' quality criteria.

Third, there is a huge overhead involved as each individual works out their own way of doing things in isolation. To avoid this we must find some way of learning from the experiences of others who have already trodden the same road.

So it is important for each organization to define the process for a project. At its most basic, this means simply to write it down. Writing it down specifies the various items that must be produced and the order in which they should be produced: from plans to requirements to documentation to the finished source code. It says where they should be kept, and how they should be checked, and what to do with them when the project is over. It may not be much of a process. One that says that all products are to be kept in a shoe box under the stairs, be checked by weighing the shoe box to see if it is over

a certain minimum weight and to burn the shoe box the minute the check is received is unlikely to win any awards. However, once you have written it down, it is a defined process.

The purpose of process

What do we want our process to achieve? We can identify certain key goals in this respect.

Effectiveness. Not to be confused with efficiency. An effective process must help us produce the right product. It doesn't matter how elegant and well-written the software, nor how quickly we have produced it. If it isn't what the customer wanted, or required, it's no good. The process should therefore help us determine what the customer needs, produce what the customer needs, and, crucially, verify that what we have produced *is* what the customer needs.

Maintainability. However good the programmer, things will still go wrong with the software. Requirements often change between versions. In any case, we may want to reuse elements of the software in other products. None of this is made any easier if, when a problem is discovered, everybody stands around scratching their heads saying "Oh dear, the person that wrote this left the company last week" or worse, "Does anybody know who wrote this code?" One of the goals of a good process is to expose the designers' and programmers' thought processes in such a way that their intention is clear. Then we can quickly and easily find and remedy faults or work out where to make changes.

Predictability. Any new product development needs to be planned, and those plans are used as the basis for allocating resources: both time and people. It is important to predict accurately how long it will take to develop the product. That means estimating accurately how long it will take to produce each part of it - including the software. A good process will help us do this. The process helps lay out the steps of development. Furthermore, consistency of process allows us to learn from the designs of other projects.

Repeatability. If a process is discovered to work, it should be replicated in future projects. Ad-hoc processes are rarely replicable unless the same team is working on the new project. Even with the same team, it is difficult to keep things exactly the same. A closely related issue, is that of **process re-use**. It is a huge waste and

overhead for each project to produce a process from scratch. It is much faster and easier to adapt an existing process.

Quality. Quality in this case may be defined as the product's fitness for its purpose. One goal of a defined process is to enable software engineers to ensure a high quality product. The process should provide a clear link between a customer's desires and a developer's product.

Improvement. No one would expect their process to reach perfection and need no further improvement itself. Even if we were as good as we could be now, both development environments and requested products are changing so quickly that our processes will always be running to catch up. A goal of our defined process must then be to identify and prototype possibilities for improvement in the process itself.

Tracking. A defined process should allow the management, developers, and customer to follow the status of a project. Tracking is the flip side of predictability. It keeps track of how good our predictions are, and hence how to improve them.

These seven process goals are very close relatives of the McCall quality factors ([**8**]) which categorize and describe the attributes that determine how the quality of the software produced. Does it make sense that the goals we set for our process are similar to the goals we have for our software? Of course! A process is software too, albeit software that is intended to be `run' on human beings rather than machines!

It has already been hinted that all this is too much for a single project to do. It is essential that the organization provide a set of guidelines to the projects to allow them to develop their process quickly and easily and with the minimum of overhead. These guidelines, a form of meta-process, consist of a set of detailed instructions of what activities must be performed and what documents should be produced by each project. These instructions are generally known as the Quality system.

Quality

Now we all know, more or less, about Quality systems. Somehow the initial capital `Q' changes the meaning utterly. It is every engineer's nightmare. On our first day in a new job we find our supervisor is away or doesn't know what to do with us. So we are handed a mountain of waste-paper and told to familiarize ourselves with `the Quality system'.

Such Quality systems are often far removed from the goals I have set out for a process. All too often they appear to be nothing more than an endless list of documents to be produced in the knowledge that they will never be read; written long after they might have had any use; in order to satisfy the auditor, who in turn is not interested in the content of the document but only its existence. This gives rise to the quality dilemma, stated in the following theorem: *it is possible for a Quality system to adhere completely to any given quality standard and yet for that Quality system to make it impossible to achieve a quality process.* (I will describe this from here on, in a fit of hubris, as Tyrrell's Quality theorem).

So is the entire notion of a quality system flawed? Not at all. It is possible, and some organizations do achieve, a quality process that really helps them to produce quality software. Much excellent work is going in to the development of new quality models that can act as road maps to developing a better quality system. The Software Engineering Institute's Capability Maturity Model (CMM) is principal among them.

The meaning of quality

The key goal of these models is to establish and maintain a link between the quality of the process and the quality of the product - our software - that comes out of that process. But in order to establish such a link we must know what we mean by quality or even Quality. The word has many meanings, and this has helped sow confusion in the minds of both developers and managers.

Earlier on, I defined quality as simply 'fitness for purpose'. Such a definition might surprise those of you who have not looked in detail at processes or quality systems. It is based on the British Standard Institute's definition (below).

Quality:

1. `The totality of features and characteristics of a product or service that bear on its ability to satisfy a given need.' (British Standards Institute [**2**]).
2. "We must define quality as `conformance to requirements.' Requirements must be clearly stated so that they cannot be misunderstood. Measurements are then taken continually to determine conformance to those requirements. The non-conformance detected is the absence of quality" [**4**].
3. `1. Degree of excellence, relative nature or kind of character 2. Faculty, skill,

accomplishment, characteristic trait, mental or moral attribute' [**12**].

Now for many of us, the BSI and Crosby definitions are counter-intuitive. Life might be a great deal easier if we had *conformance* systems - for intuitively the BSI definition could perhaps better be applied to the word *conformance*! Crosby in particular explicitly rejects the notion of quality as `degree of excellence' because of the difficulty of measuring such a nebulous concept.

Yet Crosby's own definition has serious gaps. Wesselius and Ververs [**13**] provide an excellent example. The example comes from the US's ballistic missile warning systems. These regularly gave false indications of incoming attacks, and would be triggered by various, mainly natural, events. One was a flock of geese, and another a moonrise. By Crosby's definition there was no quality problem with the system. How come? Because the model didn't include a moonrise. Therefore, the system remained completely conformant to its specifications, and hence its quality was unaffected!

Intuitively this is simply wrong. Few of us, especially given the nature of the application, would agree that this system was flawless! There has to be a subjective element to quality, even if it is reasonable to maximize the objective element. More concretely in this case we must identify that there is a quality problem with the requirements statement itself. This requires that our quality model be able to reflect the existence of such problems, for example, by taking measures of perceived quality, such as the use of questionnaires to measure customer satisfaction. Such methods can begin to measure the `yes, its what I asked for but it still doesn't feel right' type response that indicates a possible requirements specification problem.

A number of sources have looked at different ways of making sense of what we should mean by quality. Most of these take a multi-dimensional view, with *conformance* at one end and *transcendental* or *aesthetic* quality at the other.

For example, Garvin [**5**] lists eight dimensions of quality:

1. **Performance quality**. Expresses whether the product's primary features conform to specification. In software terms we would often regard this as the product fulfilling its functional specification.
2. **Feature quality**. Does it provide additional features over and above its functional specification?

3. **Reliability**. A measure of how often (in terms of number of uses, or in terms of time) the product will fail. This will be measured in terms of the mean time between failures (MTBF).
4. **Conformance**. A measure of the extent to which the originally delivered product lives up to its specification. This could be measured for example as a defect rate (possibly number of faulty units per 1000 shipped units, or more likely in the case of software the number of faults per 1000 lines of code in the delivered product) or a service call-out rate.
5. **Durability**. How long will an average product last before failing irreparably? Again in software terms this has a slightly different meaning, in that the mechanisms by which software wears out is rather different from, for example, a car or a light-bulb. Software wears out, in large part, because it becomes too expensive and risky to change further. This happens when nobody fully understands the impact of a change on the overall code. (A good description of `the aging software plant' may be found in [**9**]). Cynics might say that, on that definition, most software is worn out before it's delivered. Other cynics would say that many consultants keep their high-paying jobs purely on the basis that, as the only person in an organization to understand their own software, that software automatically `wears out' as soon as they leave!
6. **Serviceability**. Serviceability is a measure of the quality and ease of repair. It is astonishing how often it is that the component in which everybody has the most confidence is the first to fail - a principle summed up by the author Douglas Adams: "The difference between something that can go wrong and something that can't possibly go wrong is that when something that can't possibly go wrong goes wrong it usually turns out to be impossible to get at or repair" [**1**].
7. **Aesthetics.** A highly subjective measure. How does it look? How does it feel to use? What are your subconscious opinions of it? This is also a measure with an interesting variation over time. Consider your reactions when you see a ten-year old car. It looks square, box-like, and unattractive. Yet, ten years ago, had you looked at the same car, it would have looked smart, aerodynamic and an example of great design. We may like to think that we don't change, but clearly we do! Of course, give that car another twenty years and you will look at it and say `oh that's a classic design!'. I wonder if we will say the same about our software.
8. **Perception.** This is another subjective measure, and one that it could be argued really shouldn't affect the *product's* quality at all. It refers of course to the perceived quality of the provider, but in terms of gaining an acceptance of the

product it is key. How many of us would regard a Skoda as desirable a car as a Volkswagen? Very few, yet they are made by the same company. This is key to a wider issue: the reputation of the provider.

More specifically to software, McCall's software quality factors ([**8**]) define eleven dimensions of quality under three categories: product *operations* (encompassing correctness, reliability, efficiency, usability, and integrity - this last referring to the control of access by unauthorized persons), product *revision* (maintainability, flexibility, and testability) and product *transition* (portability, reusability, and interoperability). The primary area that McCall's factors do not address are the subjective ones of perception and aesthetics - possibly he felt they were impossible to measure or possibly the idea in 1977 that software could have an aesthetic quality would have been considered outlandish! But nowadays most professionals would agree that such judgments are possible, and indeed are made every day.

All of us will recognize that products do not score equally on all of these dimensions. It is arguable that there is no reason why they should, as they are appealing to different sectors of the market with different needs. To take an example with which we will all be familiar; many consumer software companies concentrate on features (performance quality and feature quality in the above descriptions) to the detriment of reliability, conformance, and serviceability. The danger for such companies is that this does damage their reputation over the longer term. The subjective measure of aesthetic quality suffers and their customers are very likely to desert them as soon as an acceptable alternative comes on the market.

Process and product quality

So which is the `right' definition of quality? Well, as it happens, it is all of them. But traditional quality systems, based on ISO9000, clearly focus on conformance to a defined process.

Why is this? You may argue that this is a flawed measure of quality, bearing little relationship to the quality of the end product.

Well, if you argue that, you would be wrong. Understandably so, because there is (see Tyrrell's quality theorem above) no guarantee that process quality (or process conformance) will produce a product of the required quality.

Such process conformance was never intended to give such a guarantee anyway. Your ISO9000 auditors don't know how good your product is, that isn't their area of expertise. They know about process, and can measure your conformance to your defined process and measure your process itself against the standards, but it is the people in your own industry that must judge your product.

The guarantee it does provide is the inverse of this. Process conformance is a necessary (but not sufficient) pre-requisite to the *consistent* production of a high-quality product. The challenge for the developers of the software meta-processes - those guides that say what the process should contain - is to strengthen the link between the process conformance and the product quality.

A key factor in this is psychological. The aim of the process should be "*to facilitate the engineer doing the job well rather than to prevent them from doing it badly*" [**11**]. This implies that the process must be easy to use correctly, and certainly easier to use correctly than badly or not at all. It implies that the engineers will want to use the process: in the jargon of the trade that they will *buy-in* to the process. It implies that there must be some feedback from the user of the process as to how to improve the process, leading to the virtuous circle that is the Holy Grail of process engineers: *Continuous Process Improvement* or *CPI*. This in turn implies that the organization provide the structures that encourage the user to provide such feedback, for without such structures the grumbles, complaints and great ideas discussed round the coffee machine will be quickly forgotten - at least until the next time someone's work is affected. Perhaps most of all it implies that the process should not be seen as a bureaucratic overhead of documents and figures that can be left until after the real work is finished, but as an integral part of the real work itself.

In the past, software quality has embraced only a limited number of the dimensions that truly constitute software quality. Focusing only on the process is limiting; it is only by including all the facets of software quality that a better evaluation of the quality of software can be obtained. The keys to better software are not simply to be found in process quality but rather in a closer link between process quality and product quality and in the active commitment to that goal of the people involved. In order to establish, maintain, and strengthen that link we must measure our product - our software - against all the relevant factors: those that relate to the specification of the product, those related to the development and maintenance of the product, and those related to our and our colleagues' subjective views of the product as well as those that relate to

process conformance.

Acknowledgements:

I would like to thank Kim Moorman of ACM Crossroads for help during the preparation of the final draft of this paper and David Covey and Jurgen Opschroef of Nokia Networks and Bill Culleton of Silicon and Software Systems Limited for comments on earlier drafts.

References

1 Adams, Douglas. *Mostly Harmless*, Macmillan, 1993.

2 British Standards Institute, **http://www.bsi.org.uk**

3 Capability Maturity Model, **http://www.sei.cmu.edu/cmm/**

4 Crosby P.B. *Quality is free: the art of making quality certain*, McGraw-Hill, 1979.

5 Garvin D. `*Competing on the eight dimensions of quality'*, Harvard Business Review November 1987, pp 101-109.

6 Humphrey, Watts. *A discipline for software engineering*, Addison-Wesley, 1995.

7 Humphrey, Watts. *An introduction to the Personal Software Process,* Addison-Wesley, 1997.

8 McCall J., Richards P., and Walters G. *Factors in Software Quality,* NTIS AD-A049-014, 015, 055, November 1977.

9 Pressman and Herron. *Software Shock*, McGraw-Hill, 1991.

10 Pressman, Roger (adapted by Ince, Darrel). *Software Engineering, A Practitioner's Approach: European Adaptation*, McGraw-Hill, 1994.

11 Silicon and Software Systems Limited's Process Group: motto. (**http://www.s3group.com**)

12 *The Concise Oxford Dictionary of Current English,* Clarendon Press, 7th Edition,

1982.

13

Wesselius & Ververs. *Some Elementary Questions on Software Quality Control*, Software Engineering Journal, November 1990.