Methodology and Culture: Drivers of Mediocrity in Software Engineering?

Marian Petre
Centre for Research in Computing
The Open University, UK
m.petre@open.ac.uk

Daniela Damian
Department of Computer Science
University of Victoria, Canada
Danielad@uvic.ca

ABSTRACT
Methodology implementation failure is attributed to developer mediocrity (by management) – not to organizational mediocrity (rigidity or control-driven, process-driven management), or to a lack of adaptation capability in the methodology. In supporting software construction as a creative process, however, we must promote excellence rather than conformity. We argue that we – through principled research -- must pay attention to the interplay between methodology and culture – the local adaptations needed to make things work, understand how the two co-evolve and how they may contribute together to software quality.

Categories and Subject Descriptors
D.2.10 [**Design**]: Methodologies, Representation

General Terms
Management, Performance, Design, Human Factors, Standardization, Theory.

Keywords
Methodology, software engineering, culture.

1. METHODOLOGY: PROMOTING SUCCESS – OR PROMOTING MEDIOCRITY?
A great deal is claimed for methodologies. Methodologies are supposed to facilitate, empower, and promote project success:

"One thing that is important if you want to have project success is having a consistent methodology across the whole organisation. … The alternative of allowing different methodologies or no methodology is often inefficiencies, higher costs, longer schedules and of course higher risk." (Birley [1])

In the right circumstances, methodology can be an effective tool. But methodology is not *necessarily* an effective tool – and when it is not, the first recourse is usually to blame the practitioner [9]. The *assumption of mediocrity* is an issue. What is attributed to individual mediocrity may be another phenomenon entirely.

This paper considers the relationship between the discourse on software engineering methodology (i.e., what's in the blogs and conference discussions, as well as in the professional and academic literature) and the 'assumption of mediocrity', in an attempt to bring attention to other phenomena that affect software quality and project success, such as organizational setting, market pressures, and pragmatic adaptation. The scope and definition of 'methodology' are kept intentionally broad; we are referring to a body of processes, procedures, methods, principles, rules, and conventions intended to introduce systematic practice into software development. Software development methodology typically embodies or implies a model of the software development process. We acknowledge that cultures often arise around introduced 'methodology' which interpret, extend and refine it (and not necessarily as the originators intended).

We argue that methodology is a tool, but it cannot be the whole recipe for producing quality software. In order to understand in which circumstances methodology is an *effective* tool, we need to understand much more fully the interaction between methodology and culture – and hence how methodology is interpreted effectively into practice.

If a methodology truly embodies 'good practice', then why wouldn't practitioners adopt it? It might be that the methodology is too expensive (i.e., the overheads of adoption are too high, or the investment in using it 'correctly' is not warranted). It might be that the methodology is too constraining. It might be that there just isn't time (or budget) to do the best thing, rather than the expedient thing. Decisions that look 'mediocre' may be appropriate in context. Therefore, before blaming the practitioner, one probably ought to consider the context.

1.1. Is Mediocrity Bad?
Let's consider how 'mediocre' is used. It means "average:", "from Latin *mediocris* 'of middle height or degree', 'somewhat mountainous', from *medius* 'middle' + *ocris* 'rugged mountain'" [12]. With respect to software engineering, it might be used to refer to average practice, or 'routine' software production.

Methodology is about providing structure both to the development process and to its outputs. The introduction of software engineering and methodology was historically a response to the increasing need for increasingly complex software, without a population of good developers at the ready to produce it [10]. This is the common articulation of the 'Software Crisis': that quality software must be developed by mediocre people (because there just aren't enough exceptional developers to meet the need), and that methodology should be the answer.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666607

Software development methodology is about systematising (process) and standardizing (process and outputs) to achieve consistency and thereby provide leverage for communication, coordination, and, notionally, quality. It suppresses variation, because there is some value to be realised from systematic constraint, consistency, convention, and standardization. But consider: making things consistent is about making things conform to a norm: "an average level of achievement or performance" [4], i.e., about making them 'average' or mediocre.

However, 'mediocre' is more often used in the context of software engineering to mean: below-standard, sub-optimal. The discourse of software production is that anything less than exceptional is not good enough. But is the 'Average Joe Developer' really so bad? Much software production is routine, a matter of applying familiar solutions to familiar problems (cf. Vincenti's distinction between radical and normal development [16]). Perhaps 'consistency', 'standardisation', and 'normal production' are useful in this context, if they help get the work done reliably. Maybe 'mediocrity' is the appropriate response to routine development?

By definition, not everyone is going to excel or be exceptional. The language of 'exceptional' and 'mediocre' developers and/or development flies in the face of reality, obscures key issues, and impedes real progress by distracting people with over-simplifications. But can software quality improve, and can methodology (or something else) promote individual, team, and organizational development toward excellence (i.e., toward reliable production of quality software)?

1.2. Methodology as Mediocrity Management

One goal of software engineering methodology is to 'bring up the bottom', to raise the standard of software development by providing a structured process (e.g., [1]). In other words, it is a form of mediocrity management. But the rigid imposition of that structured process suppresses local adaptation and drives local expertise to ground (cf. Scott, 1999); i.e., methodology potentially manages (if not promotes) mediocrity at the cost of constraining creative and innovative performance (i.e., constraining 'the top'). A rigid imposition of methodology can encourage a mechanical or 'production line' approach to development, inhibiting creative problem solving (e.g., [18]).

Of course, creativity is not the only contributor to quality, and there are times (as in safety-critical systems) when quality is judged strictly in terms of freedom from error or unexpected behavior. In such cases, a highly prescriptive, formal process is essential and may be embraced – but at a cost. Fishman's account [7] of NASA's software development process portrays just such an example. Similarly, Gawanda [8] makes the case for checklists, which provide a rigid methodology that reduces failure when a quick response is needed in a complex situation.

2. CONTROL VS. ADAPTATION

A key distinction lies in whether the purpose of introducing methodology is managing (expressed as control, correction, standardization) – or enabling (as coordination, adaptation, leverage). The former is a management perspective, the latter a development perspective. We might characterize the difference between these perspectives as a series of contrasts, as in Table 1.

Wastell *et al*. [18] make a similar distinction, expressed in terms of two opposing paradigms underlying the software process approach: the 'paradigm of control' and the 'paradigm of learning'. They use the distinction to consider process support systems; we use it to consider the implementation of methodology, highlighting the *intention* driving implementation: a desire to control the process (viewing methodology as a specification of what should happen and controlling activity accordingly) or to learn (using the methodology to provide insight and understanding, allowing performance to be improved).

Table 1. Contrasting perspectives on the purpose of methodology introduction

Management	Development
control	adaptation
standardization	coordination
correction	leverage

The interpretation of methodology as 'something one adheres to', rather than 'something one learns from and leverages', is constraining. One might learn from resonances in how other domains address methodology: one first learns rules and vocabulary, and one then learns *from* them and adapts them for use. For example, in judo one learns techniques and demonstrates them formally in kata; however, applying those techniques in order to fight well requires adapting them to one's own physique and strengths. The utility of this distinction is in highlighting that the issues lie, not in the methodology *per se*, but in how the methodology is implemented, how it is applied in a specific organizational and development context.

Methodology provides structure, but culture, within the organizational milieu, *interprets* that structure. Software designers' decisions are influenced not just by methodology (and the constraints it imposes), but also by context (and the constraints it imposes), including the work environment, organizational culture and politics, management processes, tools, and so on. The interplay between methodological and contextual constraints is crucial. If they clash, then methodology implementation as intended fails (either because the methodology is rejected, or because its application does not result in quality software). Arguably, the expert understands and manages the interplay.

2.1. 'Adapting Is Adopting'

What if what looks like 'mediocrity' (i.e., what looks to a methodologist like sub-standard practice, or incomplete adoption) in methodology implementation is something else? For example:

- Applying methodology selectively or loosely may be pragmatic adaptation: 'good enough' implementation can be cost-effective, providing some leverage without incurring the full costs of rigidly enforced implementation.
- The decision to compromise over methodology at a given time might be underpinned by a deep understanding of the local development culture. If that culture includes other systematic practices (whether formally identified as methodology or not), then what looks like compromise might actually be an intelligent, effective integration of approaches.
- Methodological work-arounds are not necessarily devious; they're often creative adaptations that bridge between effective development culture (that includes other approaches) and management culture.
- 'Ad hoc' deviations from or adaptations of methodology are not necessarily haphazard or ill-considered. Many high-performing teams evaluate methodologies and tools systematically – including evaluating their fit to the team's ethos and evolved practices – before deciding what to adopt and to what extent. [13]

'Adoption through adaptation' of methodology may not be a sign of inadequacy. Ironically, pragmatic adaptation may be evidence of excellence. Suchmann's work [15] on 'plans and situated actions' provides some insight into the balance of process and adaptation. Plans (in this case methodology) become artefacts that in turn support planning as an action situated within a particular context. Adaptation is a response to situation. Hence, adaptation situates methodology, implementing process in context. Good adaptation means making the most of process to further the goals of the situation.

It's not that methodology and development tools (whether the waterfall model, flow charting, formal methods, UML) are never applied – rather, they're used when they're useful, but not for everything. We mustn't mistake the lack of *wholesale* adoption for a lack of systematic behavior. When developers say that they 'don't use a standard methodology', that doesn't mean that they don't have systematic practices. The evidence is that high-performing teams deliberately evolve systematic practices, which draw from and are informed by a variety of sources, which gives them the benefits (or some of them), but bypasses the costs [13].

"If you trust that your developers are highly competent and self-disciplined, you'll organize your software differently than if you assume developers have mediocre skill and discipline. One way this shows up is the extent that you're willing to rely on convention to maintain order." (Cook [5])

Organisations need a culture of 'quality', based, not in blind rigid application of methodology, but in adaptive use of methodology to achieve goals – one of which is developing quality software, and one of which is developing quality developers.

2.2. Organizational Mediocrity

Perhaps the mediocrity we should be most concerned about is the mediocrity in organisations, as expressed in, for example, rigid impositions by management; highly constraining structures and procedures; assumptions of mediocrity in personnel; emphasis on uniformity over productivity. Scott [14] writes about how societies introduce standardisation in order to facilitate exchange and communication, but that centrally-managed social plans to impose administrative order can be profoundly destructive. Success of designs for social organization depends on the recognition that local, practical knowledge is as important as formal, epistemic knowledge. Similarly, Crenshaw [6] wrote: "The more stringently we enforce the methodology, the more likely we are to get a mediocre product."

There's a balance to strike between standardization (and the benefits it may bring) and local adaptation. So methodology must admit flexibility. We still need to understand the *application* of methodology in a specific socio-technical context.

Consider, for example, the adoption of agile development: it is selective, adaptive. Arguably, that the 'agile movement' admits variation has contributed to its adoption. The agile manifesto specifies principles, but admits a wide range of behaviours within that framework.

Consider physical tool use: the right tool can make a job easier, but only if one knows to choose it and how to use it. That is, expertise is embodied not just in the toolkit, but in the knowledge about which tools to use, in which circumstances, and in which ways. Many tools are used effectively in ways never intended or envisioned, and tools can be more powerful in the hands of an expert. Moreover, specialist tools, even ones that provide advantages, may just not be worth the investment.

Consider lessons from Software Carpentry [19], designed to introduce engineering practices into scientific software. The philosophy is that, if scientist developers adopt one practice at a time, skill will accumulate, with direct impact on the quality of scientific software. In this context, enforcement of methodology is viewed as an impediment to skill acquisition, whereas gradual adoption is a mechanism for skill development.

Perhaps mediocrity lies less with the developer than with the organization that, in assuming and mitigating against mediocrity, constrains away learning, selection, adaptive use. Maybe organisational mediocrity is what stops Average Joe from developing expertise or creativity? Michael Church makes this case [3], arguing that context (constraint, the wrong project, work environment) makes software engineers mediocre (meaning less than effective).

3. PROMOTING EXCELLENCE VS. MEDIOCRITY MANAGEMENT

If we assume that practitioners are competent, then what drives their decisions? What do they take from methodology – when do they adopt it, and when do they decline it?

Methodology affords potentially valuable leverage:

- structure
- coordination (standardisation, consistency)
- re-use
- communication (common language)
- sharing artefacts (especially in a potentially diverse context)

The importance of a specified methodology may be greater for less-experienced developers or for organisations that haven't already evolved their own mechanisms for these things.

There are times when developers interpret methodologies strictly:

- When they are first learning them.
- When they fit their context well.
- When the perceived/experienced benefits outweigh the costs.

There are also times when developers deviate from strict interpretation:

- To adapt to local needs.
- When the cost of adherence exceeds the perceived benefit.
- When the methodology (or its underpinning philosophy) is at odds with an effective existing culture.
- When adherence is too constraining.

We already have evidence that there are effective practices 'in the wild' that differ from the 'received wisdom', and that principled empirical study of effective practices can benefit the discipline as a whole [13]. There is rich information in both the similarities and differences between practice and methodology. We argue that it is appropriate to shift the discourse from mediocrity management to promoting excellence – not just through the development of tools (such as methodology), but through deeper understanding of how culture shapes adoption and performance. Methodology is a support, *not a replacement*, for critical thinking.

4. VARIATION COMPLEMENTS SYSTEM; CO-EVOLUTION WITH CULTURE

The important role of methodology is promotion (of excellence), not suppression (of mediocrity). Methodology will never be the whole answer. Excellence requires a balance between system and variation, and hence attention to the practice and culture that achieves an effective balance. "Just because an employee does things differently doesn't mean he or she won't do the job right or

as well. If you establish expectations of the goal and the standards to follow, then methodology shouldn't be an issue." (Mackay [11]) In general, effective adoption of methodology needs to go beyond strict interpretation. It needs to leverage the contribution to coordination, while tolerating productive adaptation.

We're not arguing that 'anything goes'. Achieving balance also requires reflection, evaluation, self-correction. Reflection is part of promoting excellence, in terms of both individual practice and organisational development. Expert practice is highly reflective, giving due attention both to seeking insight along a suggested design trajectory and to reconsidering the trajectory as understanding develops – in contrast to most software engineering methodologies. Expert behavior includes significant elements of reflection, correction, and reassessment of the design problem, as well as the application of effective engineering practices [13].

In order to make real progress, we need to understand the interplay of factors, rather than focusing on one factor (such as methodology or notation). We need to understand how to make methodology work in context (and *for* the context), how to value and develop individuals – and then deploy them in a way that uses their strengths and compensates for their weaknesses, how to create an organizational context that balances process and support with adaptation and creativity (cf. [17]).

Quality software comes from using well what people do best, having people work intelligently, reflectively, collaboratively, providing room for creativity but also a safety net to find and address things that go amiss, and to drive organizational learning. So, that means using a tool as appropriate, e.g., understanding when freedom from error is a priority and using an agreed process (methodology) to leverage coordination and provide the mechanisms the improve quality; or understanding when creativity is a priority and providing tools that empower rather than constrain.

We should emphasise design for co-evolution of software practice (i.e., creative adaptation) and context (i.e., environmental constraints). Just as the methodology may be adapted under changing constraints, so the project milieu may also change as a result of how the methodologies are adapted/implemented.

The implications are that we need to understand:
1. the nature of creative adaptations – including what 'hooks' need to be built in to accommodate adaptation;
2. culture as part of software engineering: how do 'ways of working' suggest adaptations of methodology?;
3. how working environments/practices are affected by methodology adoption.

"Great designs come from great designers. Software construction is a creative process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge." (Brooks [2]) Whereas the difference between poor conceptual designs and great ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Software construction is a creative process. We must promote excellence rather than conformity; we must respect individuals; we must pay attention to the interplay between culture and methodology – to the local adaptations needed to make things work. The blogosphere is attuned to this debate; we need principled research to underpin it. Too much of the empirical work on methodology has been undertaken to demonstrate that a proposal works, rather than to understand the bigger picture of how methodology and culture co-evolve, and how they may contribute together to software quality.

5. REFERENCES

[1] Birley, P. 2011. The importance of methodology if you want to achieve project success. http://www.enterprisecioforum.com/en/blogs/peterbirley/importance-methodology-if-you-want-achie [acc. 13 May 2014]

[2] Brooks, F.P. 1986. No silver bullet - essence and accidents of software engineering. In: IFIP Congress, 1069–1076.

[3] Church, M.O. 2012. The world sucks at finding the right work for engineers. http://michaelochurch.wordpress.com/2012/11/16/the-world-sucks-at-finding-the-right-work-for-engineers/ [accessed 13 May 2014]

[4] *Collins Dictionary of the English Language* 1986. 2nd edition. William Collins and Sons Ltd.

[5] Cook, J.D. 2011. Software architecture as a function of trust. http://www.johndcook.com/blog/2011/05/26/software-architecture-and-trust/ [accessed 13 May 2014]

[6] Crenshaw, J. 2010. On mediocrity. http://www.embedded.com/electronics-blogs/programmer-s-toolbox/4206199/On-Mediocrity [accessed 13 May 2014]

[7] Fishman, C. 1996. They write the right stuff. http://www.fastcompany.com/28121/they-write-right-stuff [accessed 13 May 2014].

[8] Gawanda, A. 2010. *The Checklist Manifesto*. Profile Books.

[9] Glass, R. 2006. *Software Creativity 2.0*. developer.* Books

[10] Haigh, T. 2010. Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72 http://www.tomandmaria.com/tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf [accessed 13 May 2014]

[11] Mackay, H. (n.d.). http://www.brainyquote.com/quotes/quotes/h/harveymack528707.html [accessed 13 May 2014]

[12] *Oxford Dictionaries*: English (online) http://www.oxforddictionaries.com/definition/english/mediocre [accessed 13 May 2014]

[13] Petre, M. 2009 Insights from expert software design practice. ESEC/FSE'09 Joint 12th European S/w Eng. Conf. (ESEC) and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-17). ACM. 233-242.

[14] Scott, J.C. 1999. *Seeing Like a State*. Yale University Press.

[15] Suchman, L. 1987. *Plans and situated actions: The Problem of Human-Machine Communication*. Cambridge Univ. Press.

[16] Vincenti, Walter G. 1990. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. Johns Hopkins University Press.

[17] Walker, D. 2010. The Soup, the Bowl, and the Place at the Table. *Design Management Journal*, 4, 4, 10–22.

[18] Wastell *et al*. 1999. The human dimension of the software process. In: J.C. Derniame, *et al* (Eds.): *Software Process*, LNCS 1500, Springer-Verlag, 165-199.

[19] Wilson, G. 2014. Software Carpentry: lessons learned [v1; ref status: indexed, http://f1000r.es/2x7]