A Case for Establishing Evolutionary Policies and their Support Mechanisms, with Examples †

Nazim H. Madhavji
University of Western Ontario, Canada
madhavji@csd.uwo.ca

Josée Tassé
University of New Brunswick, Canada
jtasse@unbsj.ca

Abstract

An important trait of a mature discipline is that, amongst other things, practitioners have specific criteria to judge the appropriateness of the different courses of action to take under a given circumstance, or whether a given task has been well-accomplished. These criteria may be in the form of templates, checklists, rules-of-thumb, constraints, policies and laws, which have resulted from many years of experience with repeated application of these in different situations. There is data to support that software evolution practices are far from mature. Thus, in this position statement, we make a case for establishing a (i) comprehensive set of evolutionary policies and (ii) their support mechanisms, to guide development[1] in the context of the instituted policies. A benefit of utilising established policies and their support mechanisms is that the sustainability of the evolving systems would likely be increased.

1. Introduction

While the overall process maturity in software organisation continues to improve according to the SEI's Year 2002 Year End update [1], there are still a staggering 60% of the 1,345 organisations assessed worldwide (appraised *and* reported since 1998) that have been calibrated at Level 1 or 2 on the software Capability Maturity Model (CMM) [2] and another approx. 25% at Level 3. The first two levels denote *chaotic* and *repeatable* practices, respectively, while Level 3 denotes *defined* processes in an organisation. Both technically and numerically, majority of the organisations are far from the 15% organisations that are at the, desired, higher levels of maturity (Level 4 -- *managed* and Level 5 -- *optimising*). Overall, therefore, the worldwide picture of software development can be considered quite gloomy.

Moreover, because most significant software projects in industry are in evolutionary stages (i.e., beyond the first release of a software system), we can assume that the software projects assessed were typically *not* new development projects. Also, whilst in general there are many factors that contribute to the overall low process maturity rating in software projects, there is no reason to believe that *software-evolution-related* factors (e.g., ability to control size growth, or amount of regression testing conducted in proportion to the degree of code change, etc.) were not amongst them. Software *evolution* community, both research and practice, thus has every reason to be concerned about the state of the art and of practice in software evolution.

Also, the Standish Group's CHAOS study [3] of 23,000 applications in US companies over five years (1994-1998) shows that, while more and more projects are succeeding, by 1998 approx. 28% were failing outright and another 46% were significantly challenged on the quality and delivery fronts. This is corroborated by data from another source [4] that also indicates that approx. 30% of the large projects get cancelled, and that the probability of a system of size 1 million lines of code (MLOC) or some 10,000 Function Points (FP) getting cancelled is approximately 50%. Not only this, large systems are notorious for: drastically overshooting schedules and budgets; severe reduction in requirements, features or functions after project start; not delivering what was promised; major reliability and performance problems following delivery; and many other issues [5].

Add to this abysmal record the approximately 8% annual growth (new + changed), though in migration projects (hardware or software based), volatile environments, or in early evolutionary life, the growth can be significantly higher (25-100%) [4]. This, therefore, raises a challenge as to how to increase the life expectancy of, say, a 10,000 function point system from the current average of 10-15 years.

2. Position

There are many lines of attack in attempting to solve software evolution problems. In this position statement, however, we make a case for establishing a (i)

† This work is supported, in part, by research grants from NSERC (Natural Science and Engineering Research Council of Canada).

[1] In this paper, by "development" we mean *evolutionary* development unless indicated otherwise.

comprehensive set of *evolutionary policies*[2] and (ii) their *support mechanisms*, to guide development[3] in the context of the instituted policies.

A rationale for this strategy is that, in a mature discipline, amongst other things, practitioners have specific *criteria* to judge the appropriateness of the different courses of action to take under a given circumstance, or whether a given task has been well-accomplished. These criteria may be in the form of templates, checklists, rules-of-thumb, constraints, policies and laws, etc., which have resulted from many years of experience with repeated application of these in different situations.

In the building industry, for example, single-glazed or ¼" double-glazed windows would be considered inadequate for the deep wintry conditions of Quebec (typically in the range -20 to -30 °C); whereas, they would be considered acceptable-to-comfortable for the mild winters of New Zealand. Such knowledge is often a result of past mistakes. For example, when early British settlers emigrated to New Zealand, the orientation of many houses did not maximise solar access in the principal rooms which, in the years to come, precipitated house remodelling.

In the field of software *evolution*, however, while progress has no doubt been made over the last thirty-odd years, exemplified by Lehman's laws [6], the general principle of "design for change" [7], or by numerous other empirical studies (some of which are cited later in the section on discussion) it is our contention that, as a community, our rate of progress in adopting and developing evolutionary policies and their support mechanisms has been undeniably slow[4]. For example, the Trial Version 1.00 of The Guide to the Software Engineering Body of Knowledge (SWEBOK) [8] – specifically Chapter 6 (Software Maintenance) and Chapter 10 (Software Engineering Tools and Methods) -- neither mentions policies for evolving software nor their technological or methodological support as a critical issue.

Thus, in the absence of a *concerted*[5] effort by the software evolution community, developers have often resorted to use, manually of course, *ad hoc* policies and rules of thumb, such as:

> *If the number of files edited for a given change is ≤ six then self-reviews would suffice; otherwise, independent inspection would be conducted.* [9].

While an argument in favour of such practice is "better this than none", it does little to further the discipline as a whole. Consequently, even in a single *large* project, let alone *across* projects, divisions or organisations, one may find inequity in the application of specific policies. The net result is an imbalance in software quality in different parts of even the same system; integration delays due to hold ups, or feature or test reduction to cope with integration and release schedules; higher evolutionary costs; and ultimately, user dissatisfaction.

Ad hoc and esoteric practices in a given project almost certainly imply a lack of a comprehensive (or practically viable) set of policies concerning different aspects of software evolution. Much remains to be done, therefore, in defining *detailed* policies to guide, monitor and verify project-specific actions in all areas of software evolution (e.g., from release planning, detailed analysis, to release implementation, and involving numerous types of software artefacts).

From the preceding description, it should be evident that the total number of policies required to comprehensively satisfy the needs of a software evolution project would be *quite large*. There is thus a danger that such a large set of policies could become the heart of a bureaucratic machine reeked with policy management problems, which would defeat the purpose of institutionalising policies in the first place.

To avoid this danger, but also, in fact, to apply policies effectively, there is a need for technological support to design, codify, organise and evolve policies and verify development against these policies. In our work thus far, we have concentrated mainly on the last of these. Detection of development-violation against the policies would help fix product or process problems at their earliest; whereas, any "positive" feedback would help build stronger confidence in the development team. Collectively, thus, a significant benefit of utilising established policies and their support mechanisms is that the sustainability of the evolving systems' quality would likely be increased.

---

[2] An evolutionary policy is defined as a statement of rule, guiding principle, strategy, plan, course of action, procedure, or constraint, to follow during the process of software evolution.

[3] In fact, we also need mechanisms to ensure *continued* relevance, comprehensiveness and soundness of the enacted policies. But we choose not to delve into policy management and evolution issues in this position statement so as not to lose focus on development issues, which are clearly of first order importance.

[4] While one may argue that this slow pace is due to the lack of a general theory of software evolution, we contend that there are nuggets hidden in numerous empirical studies and in general practice awaiting discovery and their synthesis into formalized policies that can be supported by automated means. A prime purpose of this position statement is to demonstrate a humble beginning in this direction.

[5] While it is not our intention to give here a particular blueprint for such a concerted effort, examples exist in other disciplines where such effort has resulted in benchmarks, body of knowledge, standards, Open Source software, etc., which have proved invaluable for experimentation, learning, and business.

3. Examples

In this section, we give two brief examples of policies derived from third-party empirical work [10, 13]. These examples deal with pertinent issues in software evolution, such as: re-engineering change-prone modules, and consistency between code and documentation. Some more examples can be found in a companion paper [12].

3.1 Example 1: Re-engineering change-prone modules

Mattsson and Bosch [10] have proposed an approach to identify those modules of a system that require re-engineering. Proactively maintaining the software (an object-oriented framework in their case) by restructuring the change-prone modules could "simplify the incorporation of future requirements″. In their approach, the change-prone modules from past releases are identified based on their size, change rate and growth rate.

Once it has been decided which modules need to be re-engineered during the development of a particular release, an important issue then is to ensure that all the identified modules do in fact go through the re-engineering process. However straightforward this may appear by itself, such monitoring -- basically carried out manually today – is leaden with the risk of losing track of the tasks involved amidst project pressures.

In an automated system, however, a policy such as the following could be defined:

Policy:

$\forall$ c $\in$ {p $\in$ TypedEntSet("Component") | p.name $\in$ <*list of components*> } •
$\exists$ r(a,m,t) $\in$TypedRelSet("activity consumes component") • a.name = "re-engineering"

where, "<list of components>" denotes the list of modules to be re-engineered. The policy says that for each component in the given list, there should be an activity called "re-engineering" that consumes (or operates on) the component.

This policy would be used to verify the development *plan*, such as that shown in Figure 1. This plan shows two versions of the same system, called V-elicit[6], (see the two double circles) and the new-release development process (see the hierarchy of boxes representing the process activities). Version 5 (V5) of the system consists of the components (see ellipses linked to V5): `visualization_V2`, `policies_V1`, `generator_V2`, `base_code_V5` and `view_matching_V1`. This system is to be updated to version 6 (V6), whose planned components (also shown by ellipses) are likewise linked by its arrows. The new-release development process (*model*) consists of the activities: `make_changes`, `re-engineering` and `testing`. For simplicity, no further activity breakdown is shown here.

Let us now assume that the Mattsson-Bosch approach identified two components for re-engineering: `view_matching_V1` and `generator_V2` (from version 5). The *"<list of components>"* in the policy description above would thus be replaced by these two

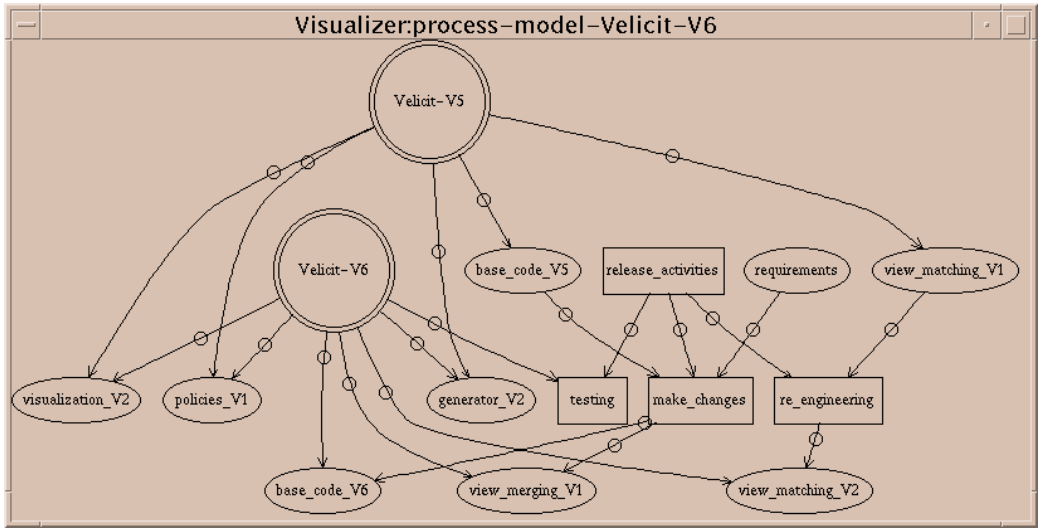

Figure 1 - Overall plan for the development of the sixth version of the V-elicit system.

[6] V-elicit is a system for eliciting models of processes or products [11]. Its operational details are not relevant to this paper.

component names[7].

The policy checking mechanism[8], described in [12], accepts two inputs (a policy and a model) and produces feedback as to whether or not the model complies with the policy and, if not, identifies the offending elements and relationships of the model.

Evidently, the plan in Figure 1 is not correct. Specifically, the component `generator_V2` (identified for re-engineering) is mistakenly left out from the re-engineering effort (i.e., this component is not an input to the "re-engineering" activity box in Figure 1). Such mistakes do occur when building prescriptive models in the planning phase, even in moderate sized projects. This is why it is quite important to verify the planned process - against the prescribed policies -- prior to its execution, in order to prevent evolution errors.

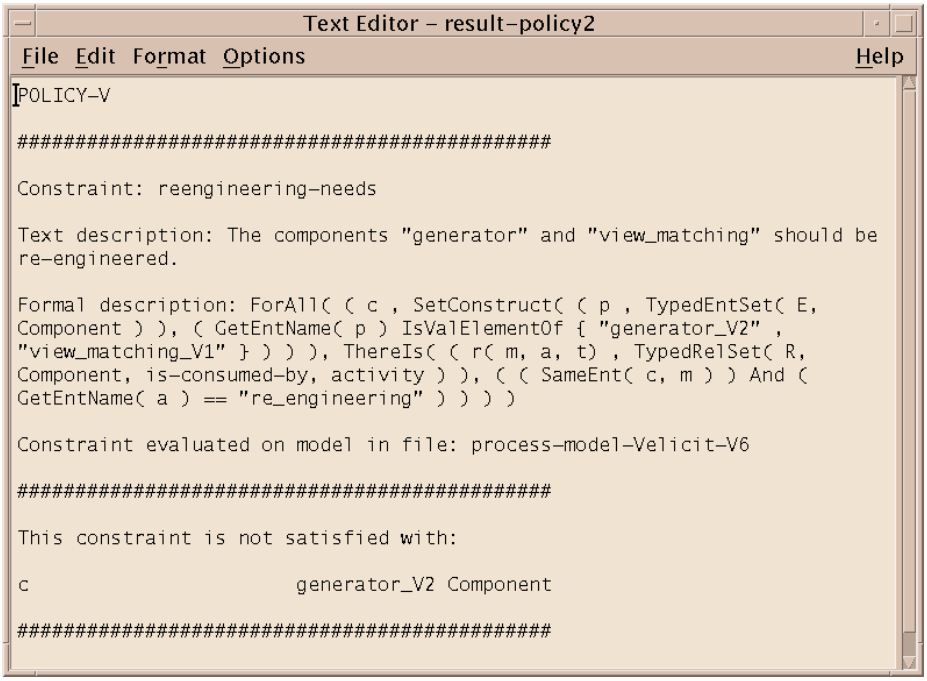

Figure 2 - Verifying the plan for re-engineering change-prone modules.

Figure 2 shows the result of verifying the plan against the described policy. The top part of the figure describes the policy informally and then formally. The bottom part lists the violations -- that is, those components that were supposed to be re-engineered but have not been included in the plan.

Such *automated* checking is much preferable to hand-checking the plans, and its value is particularly felt: in large or complex systems; when many individuals are involved in the project; when quality is at stake; and when time is at a premium. Also, the policy checking mechanism can be used in either *prescriptive* or *descriptive* contexts. For example, in the former context, as described above, it is used to ensure that the plan is complete prior to process enactment. In the latter context, it can be used to monitor a project's progress by verifying the process-trace against the policy at a desired time in the project.

3.2 Example 2: Code-documentation consistency

A case study by Tryggeseth [13] shows that the availability of valid documentation during software evolution increases system understandability and productivity. However, maintainers and evolvers often document their work by means of memos, which are not always integrated into the master documentation [7]. Over time, therefore, the documentation gets increasingly out of date to the point that the documentation is no longer trusted or used. Often, this triggers costly and intensive reverse-engineering of the system to recover the "lost" design, architectural, requirements or other software artefacts.

A preventative approach would ensure that with any new development or changes the documentation and implementation are congruent with each other. For example, in an object-oriented system one may want to verify that the code implements exactly the class diagram in the design document (i.e., no classes missing or added, and all attributes and method interfaces properly implemented). This can be achieved by comparing the class diagram from the design document against that generated by reverse-engineering the new or updated code, guided by a policy that specifies those entities and relationships that should be similar in the two diagrams. For example, the following policy verifies whether all the classes in the code are included in the design document.

[7] An advanced form of this policy could automatically detect the components that should be re-engineered, for example, components with a change rate higher than a certain threshold.

[8] This mechanism is relevant here in concept, not so much in its details.

Policy:

∀ c1 ∈ {c ∈ TypedEntSet("Class") | c.source = "code" } •
∃ c2 ∈{d ∈ TypedEntSet("Class") | d.source = "documentation" }•
c1.name = c2.name

Figure 3 (bottom part) then shows that the *code* class `line-on-invoice` does not match the class diagram from the *documentation*. Likewise, policies can be written to verify in more detail whether related classes have the same attributes and functions (including parameters and return values).

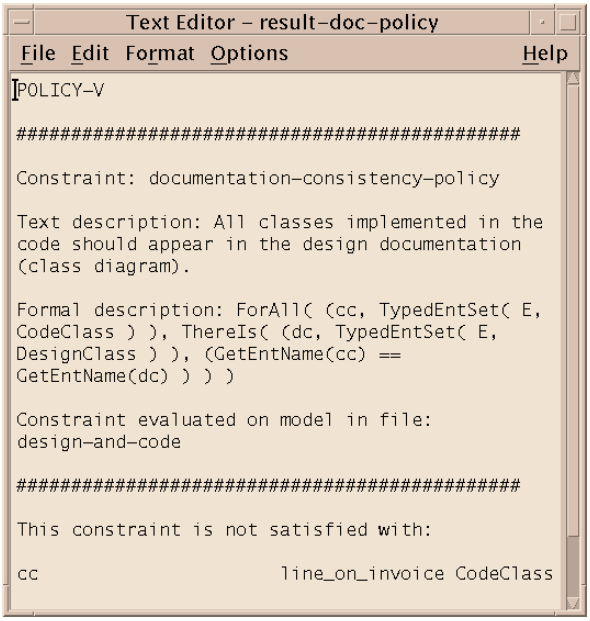

Figure 3 - Verifying consistency between documentation and code.

3.3 Discussion and Closing Remarks

The described two examples are illustrative in the basic idea of evolutionary policies and their supporting mechanism, though, needless to state, much further work is necessary to make this an industrial-scale reality. As a step in this direction, we have derived, codifed in logic and, in some cases, pseudo-codified for preliminary assessment, a number of other policies interpreted from empirical studies or experiential works of others, e.g.: 35 policies from Davis' 201 principles of software development [14]; 42 policies from Lehman's laws of evolution [6]; Munson's proportional regression testing policy [15]; Humphrey's optimal value of "*Appraisal-to-failure ratio*" [16]; and Ramanujan et al.'s "*standard for variable naming*" [17]. What this does suggest is that evolutionary policies are numerous, if implicitly buried in their rudimentary forms in the literature or in specific practices.

Time is thus ripe to dig into such literature, all the while conducting more empirical studies to establish evolutionary facts; use the findings to design suitable evolutionary policies; experiment with such policies to assess their validity in case studies and in practice; and investigate into policy-design and support mechanisms (see [12], for example, where we describe a mechanism to verify evolutionary software artefacts and processes against instituted policies). The overall objective of this work is to improve software evolution practice and to improve the quality-sustaining power of software systems. This is no mean task, however, and therefore to make significant progress, it would require a concerted, community, action as opposed to isolated efforts of a few individual researchers and practitioners.

ACKNOWLEDGEMENTS

We are thankful to the three anonymous referees, whose comments have helped us to improve this position paper.

4. References

[1] The SEI, "Software CMM CBA IPI and SPA Appraisal Results 2002 Year End Update", April 2003, available from: www.sei.cmu.edu/sema/profile.html (accessed: May 2003).

[2] M.C. Paulk, C.V. Weber and B. Curtis, "The Capability Maturity Model: Guidelines for Improving the Software Process", Addison Wesley Professional, 1995.

[3] "CHAOS: A recipe for success", The Standish Group International, Inc., 1999.

[4] C. Jones, "Applied Software Measurement – Assuring Productivity and Quality", 2nd edition, McGraw Hill, 1996.

[5] W. Wayt Gibbs, "Software's Chronic Crisis", Scientific American, September 1994, pp72-81.

[6] M. M. Lehman and J. F. Ramil, "Rules and Tools for Software Evolution Planning and Management", Annals of Soft. Eng., Vol. 11, 2001, pp. 15-44.

[7] D.L. Parnas, "Software Aging", Proc. Of the 16th International Conference on Software Engineering, Sorento Italy, May 1994, IEEE Press, pp. 279-287.

[8] SWEBOK -- Guide to the Software Engineering Body of Knowledge, Trial Version 1.00, May 2001, IEEE Computer Society.

[9] H. Dayani-Fard, Personal communication, 2002.

[10] M. Mattsson and J. Bosch, "Observations on the Evolution of an Industrial OO Framework", Proc. of the International Conference on Software Maintenance 1999, pp. 139-145.

[11] J. Tassé and N. H. Madhavji, "View-Based Process Elicitation: a User's Perspective", Software Process Improvement and Practice, vol.6 no.3, Sept. 2001, pp. 125-139.

[12] N.H. Madhavji and J. Tassé, "Policy-guided Software Evolution", Proc. of the International Conference on Software Maintenance, September 2003, Amsterdam (To appear).

[13] E. Triggeseth, "Report from an Experiment: Impact of Documentation on Maintenance", Empirical Software Engineering, vol.2 no.2, Kluwer Academic Press, 1997, pp.201-207.

[14] A. Davis, "201 Principles of Software Development", McGraw Hill, 1995.

[15] J. C. Munson, "Measuring Software Evolution", chapter submitted for consideration in "Software Evolution" (eds.) Madhavji, N.H., Lehman, M.M., Ramil, J.F. and Perry, D., Wiley (pending).

[16] W. S. Humphrey, "Using a Defined and Measured Personal Software Process", IEEE Software, vol.13 no.3, May 1996, pp. 77-88.

[17] S. Ramanujan, R. W. Scamell, J. R. Shah, "An Experimental Investigation of the Impact of Individual, Program, and Organizational Characteristics on Software Maintenance Effort", Journal of Systems and Software, vol.54 no.2, October 2000, pp. 137-157.