Linking the Effect of Typographical Style to the Evolvabililty of Software

Position Paper

Andrew Mohan **Nicolas Gold**

Information Systems Group
Department of Computation
UMIST
UK

a.mohan@postgrad.umist.ac.uk
n.e.gold@co.umist.ac.uk

1. Introduction

The fact that comprehending software is a costly business is not in question. The question is whether this cost can be controlled, even reduced, as the software evolves. To answer that question one must analyse the source code, the key artefact in this evolutionary software lifecycle, to discover and maintain its evolvable quality.

This paper presents the problem of program comprehension and its relationship with programming style. It defines programming style as those characteristics of source code associated with formatting and commenting (i.e. typographical style [1]). The paper also outlines a position whose ultimate aim is to support the software evolution process through maintaining comprehensibility. This aim could be achieved by managing the cost of increasing code comprehensibility (in terms of deviation from a base programming style), through the application of groomative maintenance [2].

2. The Problem

In the software life cycle, maintenance is the final stage, taking place once the developed software has been incorporated into the business. However it is the most costly activity taking up between 40 and 70 percent of the cost of any software system [3, 4].

Software evolution is the result of the application of maintenance to software over time. To apply maintenance to existing code, the maintainer firstly requires a sufficient level of comprehension of that code [5, 6, 7]. This process of program comprehension is the most costly activity of software maintenance [8]. The key artefact in this process is the source code itself [9]. The ease of comprehending this source code is strongly influenced by the programming style (e.g. use of comments, variable naming, indentation) employed by the original developer and subsequent maintainers [1]. Therefore determining software's stylistic quality, by learning (or discovering) the base programming style, is a desirable activity.

Groomative maintenance is the activity in which software is changed, without changing its functionality, to improve its maintainability [2]. This maintenance activity is applied because as a program evolves it becomes more complex, and thus more difficult to comprehend and maintain [10]. If groomative maintenance is applied to increase the stylistic quality of the program, this should improve its comprehensibility and consequently maintainability (and evolvability). There are several associated problems:

1. What is program style?
2. Why and how does program style affect program comprehension?
3. When does this effect become problematic?
4. How can program style be learnt?
5. Who will benefit from improving the stylistic quality of a program?

To define program style (and metrics to record and subsequently compare program style), one must consider that these must reflect the quality of the software in terms of comprehensibility (as opposed to identifying its author for instance [11]). The difficulty is to identify what programming style attributes need to be measured that affect the quality attribute of comprehensibility. A major part of this difficulty is that each programmer is individual. The styles they prefer and that are easier for them to comprehend are individual to them [12]. This individual learning (or constructivist learning) implies that a coding standard is wholly effective only for that particular individual (although the imposition of organisational coding standards may facilitate a “middle ground” position that is sufficiently effective for everyone). Our work is focused upon the learning (or discovery) of a coding standard to use as a quality benchmark. This can then be used to measure the degradation of code quality in respect to comprehensibility for that individual, whether they be an individual person, group or company.

To clarify that any changes in comprehensibility result from changes in the coding style, a method of rating the comprehensibility of the program needs to be identified. This could be achieved either manually or automatically with some kind of tool. The problem with a manual method is that this is very subjective and costly (although accurate in terms of judging comprehensibility for that particular maintainer). An automatic method is objective but may be limited in terms of the capabilities of the evaluation tool.

3. Proposed Solution

The solution to the above problem can be expressed as proving, or otherwise, the following hypotheses:

1. A quality attribute of programming style can be learnt (or discovered) and related to program comprehension.
2. The degradation in the stylistic quality of a program is associated with an increased cost in comprehending and therefore maintaining it.
3. A degradation “boundary” for stylistic quality can be determined in the evolution of a program which can indicate the need for the application of groomative maintenance to improve this quality aspect.

The style used when writing or maintaining a program has a direct impact upon the quality of the software and consequently upon a program’s comprehensibility and maintainability [13]. Furthermore, as an evolving program changes its complexity increases unless maintenance is undertaken to reverse this [10]. This leads to the possibility of using programming style as a stylistic coding quality standard. An example standard (based upon [14, 15], but not exhaustive) would consider:

- Module Length - average of non blank lines
- Identifier Length - average
- Comments - percentage of program
- Indentation - ratio of initial spaces to chars
- Blank Lines - percentage of program
- Line Length - average of non blank lines
- Embedded Spaces - average number per lines
- Constant Definition - percentage of user identifiers that are constants
- Reserved Words - number of different reserved words and standard functions used
- Included Files – number of occurrences.

If we look at indentation as an example, here a stylistic standard should indicate a level from 2 to 8 (the normal upper limit). However specifying an exact level is more problematic. This is because overly indented programs hinder comprehension, due to the associated increase in both the horizontal and vertical costs of reading the program [13, 16]. Miara et al discovered that indentation is needed in a program, as no indentation makes a program difficult to comprehend. Indentation is therefore desirable in a program and should be at a moderate level, i.e. 2 or 4 spaces. However an important factor regarding comprehensibility, is that whatever indentation style is used it should be consistent throughout [17].

The degradation of a program's stylistic quality, derived by the measurement of deviance from the standard used in version X+1 against version X, could be used to predict when groomative maintenance should be applied to the code to improve its falling stylistic quality. However there is a requirement to demonstrate that the degradation in quality, measured through this deviance, is related to increased difficulty in comprehending the code. This would also provide the necessary business benefits to undertake the work.

To model the changes in comprehensibility we are exploring the use of an automated method: hypothesis-based concept assignment (HB-CA) [18]. This is a method for automatically recognising concepts (descriptive terms nominated by the programmer, e.g. updating a policy), within a program and matching them to sections of the code to help the maintainer rapidly build an initial understanding of the program [19]. The number of concepts identified or, in particular, the number that are not, could be used as a measure of program comprehension, i.e. a degree of difficulty modeller. HB-CA is particularly suitable for this task because it uses those clues in the source code (e.g. comments and identifiers) that maintainers use when forming hypotheses about a program [18]. HB-CA has a knowledge-base defined by the maintainer containing concepts of interest from the application and software engineering domains, and possible source code clues to these (e.g. words that might be used in identifiers or comments). The method creates hypotheses for the appropriate concepts when it finds a clue, identifies areas of source code where hypotheses for similar concepts are found (using a flexible concept-oriented rather than locationoriented approach, see [20]), and assesses the evidence in each area to provide the most likely description for that code.

The establishment of the relationship between programming style and comprehensibility via concept assignment is to be achieved by measuring the deviance in stylistic coding quality of version X from version X+1 and relating this to the concepts with each version. This evolutionary measurement could then predict an effect upon program comprehension using the stylistic quality or at least highlight offending areas of code that have caused the effect (we have detailed a framework on concepts and the comprehensibility of evolving programs using a version of HB-CAS with initial case studies, see [21]).

4. Final Remarks

The ability to predict an effect upon program comprehension using the degradation of code quality may indicate the need for groomative maintenance to reinforce the quality standard upon the software. This is analogous with the process of rejuvenating software to prevent or reverse the effects of software aging [22, 23]. If concept assignment can be used to model the effect of program style upon program comprehension, during the evolution of that program, then HB-CAS can be used as a modeller of certain aspects of software quality. Indeed the evolution of the concepts themselves could be capable of indicating to maintainers a way of producing more comprehensible code by, for example, indicating candidates for refactoring [24].

The position presented in this paper is that the automatic modelling of software quality, given both a measurable stylistic coding standard and a relationship to comprehensibility, has the potential to contribute to reducing the program comprehension burden associated with evolving software. This is achieved by ensuring that by adherence to its stylistic quality standard, the evolvability of the code is maintained or even improved.

References

1. Oman, P. & Cook, C. (1990). Typographic style is more than cosmetic. Communications of the ACM, vol.33, nos.5, pp506-520.
2. Chapin, N., Hale, J.E., Khan, K.Md., Ramil, J.F. & Tan, W. (2001). Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, vol13, pp 3-30.
3. Lientz, B.P. & Swanson, E.B. (1980). Software Maintenance Management; Addison-Wesley, Reading M.A..
4. Takang, A.A. & Grubb, P.A. (1996). Software Maintenance – Concepts and Practise. Int. Thomson Computer Press.
5. Brooks, R. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, pp543-554, June 1983.
6. Mayrhauser, A.von & Vans, A.M. (1995). Program Comprehension during Software Maintenance and Evolution; IEEE Computer, vol.28, pp44-55.
7. Pennington, N. (1987). Stimulus Structure and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19, pp295-341.

8. O’Brien, M.P. & Buckley, J. (2001). Inference-based and Expectation based Processing in Program Comprehension. Proceedings 9th International Workshop on Program Comprehension, IEEE Computer Society, pp71-78, Toronto, Ont., Canada, 12-13 May, 2001.
9. Dromey, R.G. (1995). A Model of Software Product Quality; IEEE Trans. of Software Engineering, vol.21, nos.2, pp146-162.
10. Lehman, M.M. & Belady, L.A. (1985). Program Evolution, Processes of Software Change, Academic Press Inc. Ltd..
11. Krsul, I. & Spafford, E. (1997). Authorship analysis: Identifying the author of a program. Computers & Security, vol.16, nos.3, pp248-259.
12. Exton, C. (2002). Constructivism and Program Comprehension Strategies: Proc. 10th International Workshop on Program Comprehension, Paris, France, 27th-29th June 2002, pp281-284.
13. Shneiderman, B. (1980). Software Psychology - Human Factors in Computer and Information Systems; Little, Brown and Company.
14. Berry, R.E. & Meekings, B.A.E. (1985). A style analysis of C programs; Communications of the ACM, vol.28, nos.1, pp80-88.
15. Oman, P. & Cook, C. (1990). A taxonomy for programming style. 18th ACM Computer Science Conference Proceedings, pp244-247.
16. Kernighan, B. & Plauger, P.J. (1978). The Elements of Programming Style; McGraw Hill.
17. Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B. (1983). Program Indentation and Comprehensibility; Communications of the ACM, vol.26, nos.11, pp861-867.
18. Gold, N.E., Bennett, K.H. (2002). Hypothesis-Based Concept Assignment in Software Maintenance, IEE Proceedings – Software, vol. 149, no. 4, pp103-110.
19. Biggerstaff, T.J., Mitbander, B.G. & Webster, D.E. (1994). Programming Understanding and the Concept Assignment Problem; Communications of the ACM, vol.37, nos.5, pp72-83.
20. Gold, N.E. & Bennett, K.H. (2001). Flexible Method for Segmentation in Concept Assignment. Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC) 2001, pp. 135-144, 12-13 May 2001, Toronto, Canada.
21. Gold, N.E. & Mohan, A.M. (2003). A Framework for Understanding Conceptual Changes in Evolving Source Code. To appear in Proc. International Conference of Software Maintenance, IEEE Computer Society, Amsterdam, Netherlands, 22nd-26th September, 2003.
22. Castelli, V., Harper, R., Heidelberger, P., Hunter, S., Trivedi, K, Vaidyanathan, K. & Zeggert, W. (2001). Proactive management of software aging; IBM Journal of Research & Development, vol.45, nos.2.
23. Bobbio, A., Sereno, M. & Anglano, C. (2001). Fine grained software degradation models for optimal rejuvenation policies. Performance Evaluation, vol.46, pp45-62.
24. Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Inc.