In search of practitioner perspectives on ‘good code’

Gail Ollis
Postgraduate researcher
School of Design, Engineering and Computing
Bournemouth University
gollis@bournemouth.ac.uk

Keywords: POP-I.B. Barriers to programming; POP-II.A. Individual differences; POP-II.C. Working practices; POP-IV-A. Approaches to software design.

Abstract

Much of a software developer's job involves working with existing code. The comprehensibility of code therefore has a significant and ongoing effect which can continue long after it was written. Personal experience has shown that some programmers' code is frustrating and time consuming to work with, while others write code that is crystal clear. This paper sets out the basis for a definition of 'good programmer' which emphasises the powerful but invisible productivity consequences for others, rather than the more readily measurable performance of the individual. The conjectured role of personality in shaping such characteristics is also discussed.

Craftsmanship matters

The differences in performance between software developers have long been considered in terms of personal productivity. There is, for example, a pervasive idea in the software industry that individual differences of an order of magnitude exist between high and low performers. This dates back to an incidental finding by Grant and Sackman (1967), who set out to investigate the effects of direct and indirect computer access on programmer performance. In those early days of computing, the norm was for programmers to submit programs to a separate computer room to be run in their absence. Direct access to the computer was a new trend. The access method, though, proved to play only a small part, strikingly overshadowed by large individual differences.

As in many other studies, the task was one of production: the writing and debugging of a piece of code. Speed of production is important to a commercial project's success and profitability. It is also easily measured. The code produced also needs to be sufficiently correct - a characteristic which is measurable less directly, through testing and bug reports. However, these measures reflect only a fraction of the lifecycle of software systems.

A considerable part of software development calls for comprehension of someone else's code. Some of this is contemporaneous. A developer on the same project, engaged for example in integration activities or a change to common code, has the benefit of both a shared knowledge of the project's goals and the opportunity to ask questions of the original writer. The task is different for future readers who later have to understand code in order to fix, extend or reuse it. The only completely reliable information on how the program works comes from the source code; there is no guarantee that other documents, if they exist, have kept pace with the realities of the implementation.

If the code is well crafted, this task can go very smoothly. One experienced developer summed up such craftsmanship thus: "If they're brilliant, they can walk away". There is no need to be able to find the author to ask how the code works; it is clear to see. When this is not the case, the task for reader is altogether more difficult, frustrating and error-prone. It is also more time consuming. Badly written code has commercial consequences, not just aesthetic ones. Unfortunately, unlike pure production from scratch, the time that poor craftsmanship adds to the task of working with existing code cannot be isolated and measured. Its effect is apparent to developers struggling to make enough sense of things to move forward, and all but invisible to others.

It is unfortunate that the word "craftsmanship", which denotes mastery of a skill, it is also closely associated with costly physical artefacts. It does not follow that the same should apply to intellectual work, but the association is nonetheless made and even taken as read. Take Spolsky (2003), for example: "Craftsmanship is, *of course*, incredibly expensive." (my emphasis). To extend a carpentry analogy used by Spolsky, this viewpoint frames the kind of workmanship where "all the screws line up" (ibid.) as something of a luxury. It neglects the significant but intangible expense of puzzling over software that is as confusing as a floor scattered with flat-pack furniture components, a sheet of cryptic diagrams and an allen key.

What is software craftsmanship?

Since the concern here is software development work which involves working with existing code, a peer definition of craftsmanship is the most relevant. Eliciting these opinions is the current focus of the author's research. Many aspects of other team members' behaviour can affect a software developer's job, including such things as the approach taken to testing, build management, version control and bug reporting. These aspects will be recorded along with views about features of good and bad software craftsmanship that affect the usability of the source code. Just as there are individual differences among writers of code, so it is among readers. The criteria are unlikely to be the same for each and every one, but any areas of consensus are of particular interest.

There is already plenty of published guidance on how to produce good code (e.g. Goodliffe, 2006). It will be interesting to explore the extent to which the advice from such material coincides with the aspects that practising developers report as a help or a frustration in working with existing code. Some problems may, for example, be difficult to express in the form of a rule or instruction. Returning to the analogy of physical artefacts, an apprentice learns craft skills not from a book but through practice over a long period in order to achieve mastery.

The efficacy of rules and instructions also depends on a sufficiently deep understanding of the concept they are trying to convey. The decisions of interest here are those at the micro level: not architectural concerns, but the everyday tiny design decisions made when implementing a broader design plan. An example would be the advice to use comments which describe not "how" but "why" the code does something. Its efficacy depends on an appreciation of how to communicate a useful "why" message to an experienced reader.

Why does personality matter?

It may be helpful, first, to be clear that many factors are involved in these micro decisions. Expertise, for example, will play a part. Nonetheless, there are also differences between equally experienced programmers; years of practice can account for some of differences between groups, but not for the variation within them.

Social context is also a powerful influence: programming takes place within a multi-layered social environment including the team, office, organisation and culture. Saha (2011) gives a comprehensive overview of environmental factors which can contribute to the presence of unreadable code. As an extreme example of social reasons for incomprehensible code, Goodliffe (2006) cites the case of a Greek programmer refusing to write comments in English, the company's lingua franca.

There is, however, also a significant cognitive dimension to programming. The ability to operationalise a problem into computable steps is prerequisite. It is clear from experience of teaching first year undergraduates tackling their first ever programming that this mapping of a real-world problem into programming language syntax comes much more easily to some than to others. The need for this systemising ability (Wray, 2007) is reflected in the preponderance among software engineers of 'Thinking' personality types (who tend to make objective, analytical judgements) over 'Feeling' types (who give weight to human and personal concerns) (Capretz, 2003).

Such a skill is sufficient to produce a working program. Many different solutions can achieve the same outcome; apart from being able to run some more quickly than others, the computer is utterly indifferent to the design of the program. This is not the case for human readers. Individual differences in the problem solving and decision making approaches of the programmer have profound effects on how readily the program can be understood by others. Although Bernstein (cited in Tien, 2000) suggests that programming is "best regarded as the process of creating works of literature, which are meant to be read", these works are not always accessible to the reader.

Conclusion

A future stage of research will explore individual differences between programmers to see whether there is a correlation between specific psychological personality dimensions and the programming behaviours, good or bad, about which some consensus has been found among peers. Systemising has already been mentioned as a prerequisite for communicating the programmer's intentions to the machine; perhaps a certain degree of empathising is helpful alongside this, in order to see things from a reader's point of view and so communicate more clearly to them. Since it is possible to create a perfectly functioning program that is not comprehensible (as illustrated by extreme examples such as obfuscated C programs; Broukhis, Cooper, & Noll, 2012), perhaps an element of conscientiousness (Costa & McCrae, 1992; cited in Halkjelsvik & Jørgensen, 2012) may also prompt programmers to cater for humans as well as machines.

Understanding approaches which tend to produce usable, 'team friendly' source code, or otherwise, is a necessary step in order to intervene effectively to improve standards. Just as programmers need to cater for the fellow programmers who are 'users' of their source code, designers of tools, processes and advice need to understand the perspective of their users if these interventions are to work.

Acknowledgements

Thank you to Keith Phalp and Kevin Thomas for their comments.

References

Broukhis, L., Cooper, S., & Noll, L. C. (2012). The International Obfuscated C Code Contest. Retrieved September 13, 2012, from http://www.ioccc.org/

Capretz, L. F. (2003). Personality types in software engineering. *International Journal of Human-Computer Studies, 58*(2), 207.

Goodliffe, P. (2006). *Code craft : the practice of writing excellent code*: San Francisco, Calif. : No Starch Press.

Grant, E. E., & Sackman, H. (1967). An Exploratory Investigation of Programmer Performance Under On-Line and Off-Line Conditions. *Human Factors in Electronics, IEEE Transactions on, HFE-8*(1), 33-48.

Halkjelsvik, T., & Jørgensen, M. (2012). From origami to software development: A review of studies on judgment-based predictions of performance time. *Psychological Bulletin, 138*(2), 238-271. doi: 10.1037/a0025996

Saha, A. (2011). *Origins of poor code readability*. Paper presented at the PPIG 2011, 23rd annual workshop of the Psychology of Programming Interest Group, University of York.

Spolsky, J. (2003). Craftsmanship. Retrieved September 12, 2012, from *http://www.joelonsoftware.com/articles/Craftsmanship.html*

Tien, L. (2000). Publishing software as a speech act. *Berk. Tech. LJ, 15*, 629.

Wray, S. (2007). *SQ Minus EQ can Predict Programming Aptitude*. Paper presented at the PPIG 2007, 19th annual workshop of the Psychology of Programming Interest Group, University of Joensuu, Finland.