Expertise

Discussion of guideline related issues

Derek M. Jones
derek@knosof.co.uk

Copyright ©2006 Derek M. Jones. All rights reserved.

1 Introduction

The following are some of the ways in which expertise is of relevance to coding guidelines:

- Amount of expertise that users of the guidelines (i.e., software developers) are assumed to have.
- Amount of expertise required to decide whether a deviation to a particular guideline is appropriate.
- How is expertise to be measured...

2 What is expertise?

It is a commonly held belief that experts have some innate ability or capacity that enables them to do what they do so well. Research over the last two decades has shown that while innate ability can be a factor in performance (there do appear to be genetic factors associated with some athletic performances), the main factor in acquiring expert performance is time spent in *deliberate practice*.[?]

Deliberate practice is different from simply performing the task. It requires that people monitor their practice with full concentration and obtain feedback[?] on what they are doing (often from a professional teacher). It may also involve studying components of the skill in isolation, attempting to improve on particular aspects. The goal of this practice being to improve performance, not to produce a finished product.

Studies of the backgrounds of recognized experts, in many fields, found that the elapsed time between them starting out and carrying out their best work was at least 10 years, often with several hours of deliberate practice every day of the year. For instance, a study of violinists[?] (a perceptual-motor task), by age 20 those at the top level had practiced for 10,000 hours, those at the next level down 7,500 hours, and those at the lowest level of expertise had practiced for 5,000 hours. They also found similar quantities of practice being needed to attain expert performance levels in purely mental activities (e.g., chess).

People are referred to as being experts, in a particular domain, for several reasons, including:

- Well-established figures, perhaps holding a senior position with an organization heavily involved in that domain.
- Better at performing a task than the average person on the street.
- Better at performing a task than most other people who can also perform that task.
- Self-proclaimed experts, who are willing to accept money from clients who are not willing to take responsibility for proposing what needs to be done.[?]

There are domains in which those acknowledged as experts do not perform significantly better than those considered to be non-experts.[?] For instance, in typical cases the performance of medical experts was not much greater than those of doctors after their first year of residency, although much larger differences were seen for difficult cases.

2.1 Creating experts

To become an expert a person needs motivation, time, economic resources, an established body of knowledge to learn from, and teachers to guide.

One motivation is to be the best, as in chess and violin playing. This creates the need to practice as much as others at that level. Ericsson found[?] that four hours per day was the maximum concentrated training that people could sustain without leading to exhaustion and burnout. If this is the level of commitment, over a 10-year period, that those at the top have undertaken, then anybody wishing to become their equal will have to be equally committed. The quantity of practice needed to equal expert performance in less competitive fields may be less. One should ask of an expert whether they attained that title because they are simply as good as the best, or because their performance is significantly better than non-experts.

An established body of knowledge to learn from requires that the domain itself be in existence and relatively stable for a long period of time. The availability of teachers requires a domain that has existed long

enough for them to have come up through the ranks; and one where there are sufficient people interested in it that it is possible to make as least as much from teaching as from performing the task.

The domains in which the performance of experts was not significantly greater than non-experts lacked one or more of these characteristics.

2.1.1 Transfer of expertise to different domains

Research has shown that expertise within one domain does not confer any additional skills within another domain.[?] This finding has been duplicated for experts in real-world domains, such as chess, and in laboratory-created situations.

2.2 Knowledge components of expertise

A distinction is often made between different kinds of knowledge. Declarative knowledge are the facts; procedural knowledge are the skills (the ability to perform learned actions). Implicit memory is defined as memory without conscious awareness— it might be considered a kind of knowledge.

2.2.1 Declarative knowledge

This consists of knowledge about facts and events. For instance, the keywords used to denote the integer types are **`char`**, **`short`**, **`int`**, and **`long`**. This kind of knowledge is usually explicit (we know what we know), but there are situations where it can be implicit (we make use of knowledge that we are not aware of having[?]). The coding guideline recommendations in this book have the form of declarative knowledge.

It is the connections and relationships between the individual facts, for instance the relative sizes of the integer types, that differentiate experts from novices (who might know the same facts). This kind of knowledge is rather like web pages on the Internet; the links between different pages corresponding to the connections between facts made by experts. Learning a subject is more about organizing information and creating connections between different items than it is about remembering information in a rotelike fashion.

One study[?] found that developers with greater experience with a language organized their knowledge of language keywords in a more structured fashion. Education can provide the list of facts, it is experience that provides the connections between them.

2.2.2 Procedural knowledge

This consists of knowledge about how to perform a task; it is often implicit.

Knowledge can start off by being purely declarative and, through extensive practice, becomes procedural; for instance, the process of learning to drive a car.

Experiments have shown[?] how subjects' behavior during mathematical problem solving changed as they became more proficient. This suggested that some aspects of what they were doing had been proceduralized.

2.2.3 Education

What effect does education have on people who go on to become software developers?

> *Education should not be thought of as replacing the rules that people use for understanding the world but rather as introducing new rules that enter into competition with the old ones. People reliably distort the new rules in the direction of the old ones, or ignore them altogether except in the highly specific domains in which they were taught.*

Page 206 of Holland et al.[?]

Education can be thought of as trying to do two things (of interest to us here)— teach students skills (procedural knowledge) and providing them with information, considered important in the relevant field, to memorize (declarative knowledge). To what extent does education in subjects not related to software development affect a developer's ability to write software?

Some subjects that are taught to students are claimed to teach general reasoning skills; for instance, philosophy and logic. There are also subjects that require students to use specific reasoning skills, for instance statistics requires students to think probabilistically. Does attending courses on these subjects actually have any measurable effect on students' capabilities, other than being able to answer questions in

an exam. That is, having acquired some skill in using a particular system of reasoning, do students apply it outside of the domain in which they learnt it? Existing studies have supplied a *No* answer to this question.[?, ?] This *No* was even found to apply to specific skills; for instance, statistics (unless the problem explicitly involves statistical thinking within the applicable domain) and logic.[?]

Good education aims to provide students with an overview of a subject, listing the principles and major issues involved; there may be specific cases covered by way of examples. Software development does require knowledge of general principles, but most of the work involves a lot of specific details: specific to the application, the language used, and any existing source code, while developers may have been introduced to the C language as part of their education. The amount of exposure is unlikely to have been sufficient for the building of any significant knowledge base about the language.

2.2.4 Learned skills

Education provides students with *learned knowledge*, which relates to the title of this subsection *learned skills*. Learning a skill takes practice. Time spent by students during formal education practicing their programming skills is likely to total less than 60 hours. Six months into their first development job they could very well have more than 600 hours of experience. Although students are unlikely to complete their education with a lot of programming experience, they are likely to continue using the programming beliefs and practices they have acquired. It is not the intent of this book to decry the general lack of good software development training, but simply to point out that many developers have not had the opportunity to acquire good habits, making the use of coding guidelines even more essential.

3 Guideline related expertise

Given the observation that in some domains the acknowledged experts do not perform significantly better than non-experts, we need to ask if it is possible that any significant performance difference could exist in software development. The following discussion breaks down expertise in software development into five major areas.[?]

1. *Knowledge (declarative) of application domain.* Although there are acknowledged experts in a wide variety of established application domains, there are also domains that are new and still evolving rapidly. The use to which application expertise, if it exists, can be put varies from high-level design to low-level algorithmic issues (i.e., knowing that certain cases are rare in practice when tuning a time-critical section of code).

2. *Knowledge (declarative) of algorithms and general coding techniques.* There exists a large body of well-established, easily accessible, published literature about algorithms. While some books dealing with general coding techniques have been published, they are usually limited to specific languages, application domains (e.g., embedded systems), and often particular language implementations. An important issue is the rigor with which some of the coding techniques have been verified; it often leaves a lot to be desired, including the level of expertise of the author.

3. *Knowledge (declarative) of programming language.* The C programming language is regarded as an established language. Whether 25 years is sufficient for a programming language to achieve the status of being established, as measured by other domains, is an open question. There is a definitive document, the ISO Standard, that specifies the language. ... we cannot expect any established community of expertise in the C language to be very large.

4. *Ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms.* Procedural knowledge is acquired through practice. While university students may have had access to computers since the 1970s, access for younger people did not start to occur until the mid 1980s. It is possible for developers to have had 10 years of software development practice.

5. *Development environment.* The development environment in which people have to work is constantly changing. New versions of operating systems are being introduced every few years; new technologies

are being created and old ones are made obsolete. The need to keep up with development is a drain on resources, both in intellectual effort and in time. An environment in which there is a rapid turnover in applicable knowledge and skills counts against the creation of expertise.

Although the information and equipment needed to achieve a high-level of expertise might be available, there are several components missing. The motivation to become the best software developer may exist in some individuals, but there is no recognized measure of what *best* means. Without the measuring and scoring of performances it is not possible for people to monitor their progress, or for their efforts to be rewarded. While there is a demand for teachers, it is possible for those with even a modicum of ability to make substantial amounts of money doing (not teaching) development work. The incentives for good teachers are very poor.

Given this situation we would not expect to find large performance differences in software developers through training. If training is insufficient to significantly differentiate developers the only other factor is individual ability. It is certainly your author's experience— individual ability is a significant factor in a developer's performance.

Until the rate of change in general software development slows down, and the demand for developers falls below the number of competent people available, it is likely that ability will continue to the dominant factor (over training) in developer performance.

3.1 Level of developer expertise

How much knowledge are software developers assumed to have? If they had sufficient knowledge then coding guidelines would not be needed (coding guidelines might be viewed as nuggets of expertise), while if they have very little then training rather than guidelines would probably be more appropriate.

Economic incentives driving hiring practices

Factors driving developers to improve their level of expertise

The culture of information technology appears to be one of high staff turnover[?] (with reported annual turnover rates of 25% to 35% in Fortune 500 companies).

It can be argued that a regular turnover of staff creates the need for coding guidelines so that source code software does not require a large investment in upfront training costs. While developers do need to be familiar with the source they are to work on, companies want to minimize familiarization costs for new staff while maximizing their productivity. Source code level guideline recommendations can help reduce familiarization costs in several ways:

- Not using constructs whose behavior varies across translator implementations means that recruitment does not have to target developers with specific implementation experience, or to factor in the cost of retraining— it will occur, usually through on-the-job learning.
- Minimizing source complexity helps reduce the cognitive effort required from developers trying to comprehend it.
- Increased source code memorability can reduce the number of times developers need to reread the same source.
- Visible source code that follows a consistent set of idioms can take advantage of people's natural ability to categorize and make deductions based on these categorizes.

It is your author's experience that very few companies use any formally verified method for measuring developer characteristics, or fitting their skills to the work that needs to be done. Project staffing is often based on nothing more than staff availability and a date by which the tasks must be completed.

3.2 Is computer language expertise worth acquiring?

People often learn a skill for some purpose (e.g., chess as a social activity, programming to get a job) without the aim of achieving expert performance. Once a certain level of proficiency is achieved, they stop trying to learn and concentrate on using what they have learned (in work, and sport, a distinction is made between

training for and performing the activity). During everyday work, the goal is to produce a product or to provide a service. In these situations people need to use well-established methods, not try new (potentially dead-end, or leading to failure) ideas to be certain of success. Time spent on this kind of practice does not lead to any significant improvement in expertise, although people may become very fluent in performing their particular subset of skills.

Many developers are not professional programmers any more than they are professional typists. Reading and writing software is one aspect of their job. The various demands on their time is such that most spend a small portion of their time writing software. Developers need to balance the cost of spending time becoming more skillful programmers against the benefits of possessing that skill. Experience has shown that software can be written by relatively unskilled developers. One consequence of this is that few developers ever become experts in any computer language.

When estimating benefits over a relatively short period of time, time spent learning more about the application domain frequently serves one than honing programming skills.

3.3 Deviation creation expertise

How much expertise, if any, over and above that expected of a developer is considered to be needed to create a deviation? The situations under which a deviation might be proposed include:

- It is not possible to perform a needed operation using any other construct.
- The cost (e.g., runtime performance) of using all other constructs is not acceptable.

The expertise needed is a knowledge of language constructs and knowledge of the application domain.

Small project are much less likely than large projects to have local access to a developer with a great deal of language knowledge. Having possible deviations documented along side a guideline gives developers access to the expertise of the authors of that guideline....

Deviation agreed after code review, more input increases likelihood that more possibilities will be considered, better decision???

A study of workers producing cigars by Crossman[?] showed performance improving according to the power law of practice for the first five years of employment. Thereafter performance improvements slow; factors cited for this slow down include approaching the speed limit of the equipment being used and the capability of the musculature of the workers.

4 Measuring software developer expertise

Having looked at expertise in general and the potential of the software development domain to have experts, we need to ask how expertise might be measured in people who develop software. Unfortunately, there are no reliable methods for measuring software development expertise currently available. However, based on the previously discussed issues, we can isolate the following technical competencies (social competencies[?] are not covered here, although they are among the skills sought by employers,[?] and software developers have their own opinions[?, ?]):

- Knowledge (declarative) of application domain.
- Knowledge (declarative) of algorithms and general coding techniques.
- Knowledge (declarative) of programming languages.
- Cognitive ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms (a specialized form of general analytical and conceptual thinking).
- Knowledge (metacognitive) about knowledge (i.e., judging the quality and quantity of ones expertise).

A study at Bell Labs[?] showed that developers who had worked on previous releases of a project were much more productive than developers new to a project. They divided time spent by developers into discovery time

(finding out information) and work time (doing useful work). New project members spent 60% to 80% of their time in discovery and 20% to 40% doing useful work. Developers experienced with the application spent 20% of their time in discovery and 80% doing useful work. The results showed a dramatic increase in efficiency (useful work divided by total effort) from having been involved in one project cycle and less dramatic an increase from having been involved in more than one release cycle. The study did not attempt to separate out the kinds of information being sought during discovery.

Another study at Bell Labs[?] found that the probability of a fault being introduced into an application, during an update, correlated with the experience of the developer doing the work. More experienced developers seemed to have acquired some form of expertise in an application that meant they were less likely to introduce a fault into it.

A study of development and maintenance costs of programs written in C and Ada[?] found no correlation between salary grade (or employee rating) and rate of bug fix/add feature rate.

There is a group of people who might be expected to be experts in a particular programming languages—those who have written a compiler for it (or to be exact those who implemented the semantics phase of the compiler, anybody working on others parts [e.g., code generation] does not need to acquire detailed knowledge of the language semantics). Your author knows a few people who are C language experts and have not written a compiler for that language. Based on your author's experience of implementing several compilers, the amount of study needed to be rated as an expert in one computer language is approximately 3 to 4 hours per day (not even compiler writers get to study the language for every hour of the working day; there are always other things that need to be attended to) for a year. During that period, every sentence in the language specification will be read and analyzed in detail several times, often in discussion with colleagues. Generally developer knowledge of the language they write in is limited to the subset they learned during initial training, perhaps with some additional constructs learned while reading other developers' source or talking to other members of a project. The behavior of the particular compiler they use also colors their view of those constructs.

Expertise in the act of comprehending and writing software is hard to separate from knowledge of the application domain. There is rarely any need to understand a program without reference to the application domain it was written for. When computers were centrally controlled, before the arrival of desktop computers, many organizations offered a programming support group. These support groups were places where customers of the central computer (usually employees of the company or staff at a university) could take programs they were experiencing problems with. The staff of such support groups were presented with a range of different programs for which they usually had little application-domain knowledge. This environment was ideal for developing program comprehension skills without the need for application knowledge (your author used to take pride in knowing as little as possible about the application while debugging the presented programs). Such support groups have now been reduced to helping customers solve problems with packaged software. Environments in which pure program-understanding skills can be learned now seem to have vanished.

Ability to estimate the effort needed to implement the specified functionality.[?]

A study by Jørgensen and Sjøberg[?] looked at maintenance tasks (median effort 16-work hours). They found that developers' skill in predicting maintenance problems improved during their first two years on the job; thereafter there was no correlation between increased experience (average of 7.7 years' development experience, 3.4 years on maintenance of the application) and increased skill. They attributed this lack of improvement in skill to a lack of learning opportunities (in the sense of deliberate practice and feedback on the quality of their work).

Job advertisements often specify that a minimum number of years of experience is required. Number of years is known not to be a measure of expertise, but it provides some degree of comfort that a person has had to deal with many of the problems that might occur within a given domain.

References

Citations added in version 1.0b start at 1449.

1. J. R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, fifth edition, 2000.
2. J. L. Bailey and G. Stefaniak. Industry perceptions of the knowledge, skills, and abilities needed by computer programmers. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research (SIGCPR 2001)*, pages 93–99. ACM Press, 2001.
3. C. F. Camerer and E. F. Johnson. The process-performance paradox in expert judgment: How can the experts know so much and predict so badly? In K. A. Ericsson and J. Smith, editors, *Towards a general theory of expertise: Prospects and limits*. Cambridge University Press, 1991.
4. P. Cheng, K. J. Holyoak, R. E. Nisbett, and L. M. Oliver. Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, 18:293–328, 1986.
5. E. R. F. W. Crossman. A theory of the acquisition of speed-skill. *Ergonomics*, 2:153–166, 1959.
6. J. W. Davison, D. M. Mand, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, Apr.-June 2000.
7. K. A. Ericsson and N. Charness. Expert performance. *American Psychologist*, 49(8):725–747, 1994.
8. K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100:363–406, 1993. also University of Colorado, Technical Report #91-06.
9. R. M. Hogarth, C. R. M. McKenzie, B. J. Gibbs, and M. A. Marquis. Learning from feedback: Exactness and incentives. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(4):734–752, 1991.
10. J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. *Induction*. The MIT Press, 1989.
11. S. A. J. The seer-sucker theory: The value of experts in forecasting. *Technology Review*, pages 16–24, June-July 1980.
12. M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.
13. M. Jørgensen and D. I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance: Research and Practice*, 14(2):123–146, 2002.
14. T. C. Lethbridge. What knowledge is important to a software professional? *IEEE Computer*, 33(5):44–50, May 2000.
15. P. Lewicki, T. Hill, and E. Bizot. Acquisition of procedural knowledge about a pattern stimuli that cannot be articulated. *Cognitive Psychology*, 20:24–37, 1988.
16. K. B. McKeithen, J. S. Reitman, H. H. Ruster, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.
17. J. McMillan. Enhancing college student's critical thinking: A review of studies. *Research in Higher Education*, 26:3–29, 1987.
18. A. Mockus and D. M. Weiss. Predicting risk in software changes. *Bell Labs Technical Journal*, Apr.-June 2000.
19. J. E. Moore and L. A. Burke. How to turn around 'turnover culture' in IT. *Communications of the ACM*, 45(2):73–78, 2002.
20. K. M. Nelson, H. J. Nelson, and M. Ghods. Understanding the personal competencies of IS support experts: Moving towards the E-business future. In 34^{th} *Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 8*. IEEE, Jan. 2001.
21. R. S. Nickerson, D. N. Perkins, and E. E. Smith. *The Teaching of Thinking*. Erlbaum, Hillsdale NJ, 1985.
22. S. Sonnentag. Excellent software professionals: experience, work activities, and perception by peers. *Behaviour & Information Technology*, 14(5):289–299, 1995.
23. T. R. Stewart and C. M. Lusk. Seven components of judgmental forecasting skill: Implications for research and improving forecasts. *Journal of Forecasting*, 13:579–599, 1994.
24. J. Sweller, J. F. Mawer, and M. R. Ward. Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General*, 112(4):639–661, 1983.
25. S. F. Zeigler. Comparing development costs of C and Ada. Technical report, Rational Software Corporation, Mar. 1995.