Rating expertise in collaborative software development

Sallyann Bryant
IDEAS Laboratory, University of Sussex.
S.Bryant@sussex.ac.uk

The literature on expertise is wide-ranging, both across many domains and within software development, however when findings from these are contrasted with literature on experience in pair programming, some startling differences become apparent. For example, knowledge seems to be a key feature in obtaining expertise, however there is little mention of core programming knowledge as a measure of expertise in the pair programming literature. This paper discusses these discrepancies, along with findings on the reliability of various types of rating, to provide context for the presentation of data from a survey of commercial pair programmers which aims to help clarify what factors are considered good indicators of pair programming expertise by different groups.

Keywords: Pair programming, expertise, rating, self-rating

1. Introduction

Research has addressed the concept of expertise across many different fields. While this paper is primarily focused on collaborative software design, it begins by considering expertise across various domains, including medicine, table-waiting, chess, physics and law. Although expertise has been seen to depend on many factors, it has been shown that experts tend to have similar goals and perspectives (Lawrence 1988) and therefore it would seem that building up a rich picture of some of the common factors in expertise is not only possible, but desirable. Having considered expertise in general, expertise in software development and then specifically in pair programming is considered. Section three of this paper looks at studies on the rating of expertise. For example, considering the relative accuracy of ratings of self, peer and supervisor. Section four presents the results of a survey of 45 pair programmers, comparing their own expertise ratings with those of their peers/superiors and considering which factors most influenced these ratings. Finally, the paper concludes by identifying key factors in the recognition of collaborative programming expertise, and their relative importance according to rater type.

2. Expertise

Here the literature on general, software development and pair programming expertise is used to ascertain common themes that may be useful to identifying and assessing expertise.

2.1 Expertise in other domains

While the generalizations and comparisons in this section assume a standard concept of expertise, one should not forget that each of the contributory studies relates to a specific subject group doing a specific task.

2.1.1 General approach

Experts seem to have a distinct approach to their work. They recover more gracefully from mistakes (Johnson 1988) let the story unfold around them and are prepared to do nothing or to settle for good enough (Eraut and du Boulay 2002). This may stem from an underlying confidence in their ability, suggesting that self-confidence plays a prominent role in expert behaviour.

2.1.2 Knowledge and practice

Studies have identified some common characteristics regarding the nature, amount and organization of information in expert's memory, the schema by which they are accessed and the representations which are made. Experts have been shown to maintain more detailed information in chunks in long-term memory (Ericsson and Polson 1988). Schmidt (1993) tells of 'encapsulated knowledge' where the expert has 'compiled knowledge' which does not need to be understood in depth to be accessed at the highest level. Kintsch (1998) voices the popular view that in short term memory chunks of knowledge are stored and accessed hierarchically, with higher levels being easier to access.

The method by which expert knowledge is accessed has also been the focus of a number of studies, amongst which Larkin (1983) finds that expert schema for accessing knowledge are much more complex than those of novices. In fact, experts have been shown to often identify meaningful patterns (Chi 1988) and then arrive at a solution without the need for a time-consuming and exhaustive search of their knowledge. Similarly Lawrence (1988) finds that judges focus on patterns in order to minimise their workload, and are excellent at identifying ideas of what to look for or follow up. This implies that certain selection rules are triggered by certain items. Chase & Ericsson's (1982) skilled memory theory states that experts use long-term memory more efficiently to remember detailed information that can then be retrieved by the appropriate cue. In a variety of studies, this meaningful addressing mechanism is known as a beacon, focal line, clue, trigger or condition. Gick and Holyoak (1985) state that analogies helpful in problem solving might be found by abstracting the problem to a suitable structure that can be used as a retrieval cue, presumably a solution might be found in the same manner.

This research suggests that knowledge and its organisation and access are key issues in defining expertise. Thus one might assume that the pre-requisites for obtaining and organising information in the most expert manner are exposure to that knowledge and practice in its use. This in turn suggests that length of tenure performing the activity in question may be key.

2.1.3 Strategy

Lesgold et al (1988) suggest that experts very quickly identify the right problem space. Similarly Kintsch (1998) suggests that experts find the appropriate category and then use the problems peculiarities to ascertain what is special about it. One might assume that this allows the expert to either further pare down the information they have to find that which is appropriate, to detail the differences between this and previously encountered cases, or to identify and suggest values for variables which are not yet specified (Voss and Post 1988). This accords with findings by Trafton, Marshall et al (2002) and Freedman and Shah (2002) that whilst novices ignore anomalies (on graphs) experts actually focus on them.

Significant work has also addressed the issue of the production of mental models of the problem and potential solution. Voss and Post (1988) mentions that expert problem solving is much more like comprehension, with problem structuring and categorization playing a larger role than finding a solution. He also notes that experts are skilled in structuring ill-structured problems. Similarly, Chi (1988) find that experts spend more time building representations and that these are more detailed than those made by less expert colleagues.

Finally, work on distributed cognition (Hutchins 1995) has shown that tricky problems are often solved through the interaction of individuals, tools and artifacts rather than by an individual working in isolation. In particular, Hutchins (1995) cites maritime navigation as a complex task achieved within a complex ecology of cognition, some of which is embedded in the environment, tools and techniques that are used.

2.1.4 Metacognition

Meta-cognitive insight has been widely observed in experts. For example, Chi (1988) shows that experts self-monitor more, Eraut & du Boulay (2002) state that they are aware of their own biases and Suwa & Tversky (2002) mention that expert architects are aware of the cycle of using external representations for the production of ideas. Ericsson & Polsom (1988) showed that an expert waiter was aware of the 'memory tricks' he used when encoding information. This implies that experts are more able to identify and comment on the approach and tools they take in problem solving in a way that novices do not. It might also suggest that those who are more expert are also more likely to give an accurate rating of their ability than those with less experience.

2.1.5 The practice-strategy-knowledge-metacognition model

It appears that several different factors interplay in the concept of expertise and that at a very high level, expertise could be considered as a function of the four key elements of Practice, Strategy, Knowledge and Meta-cognition.

Section 2.2 will explore each of these in relation to expertise in software development.

2.2 Expertise in software development

2.2.1 Knowledge

Soloway, Adelson et al. (1988) noted that expert programmers have two types of knowledge: Programming plans (syntactical knowledge) and rules of discourse (over-arching rules on how to use that knowledge usefully) and showed that experts performed well on plan-like problems but the same as novices on unplan-like problems. Chi (1988) indicates that experts form a much more detailed problem representation than novices, which Adelson (1984) identifies as an internal model through which the expert can run simulations. Petre & Blackwell (1999) further define this model as being particularly rich and stoppably dynamic. These findings are consistent with general studies in expertise across domain, in that knowledge is considered key. Detienne (1990) describes the two main schools of thought regarding programming knowledge – those which focus on schema-based knowledge (semantic knowledge about what the program does) and those focusing more on control-flow knowledge (syntactical knowledge about how the program works). She goes on to note that there is good evidence for the existence of both, and that their importance may be seen as relative to the language, task, environment and the programmer herself.

2.2.2 Strategy

While acknowledging that knowledge plays a role, Gilmore (1990) suggests that strategy is more important than knowledge in programming expertise and that experts have the ability to select the most appropriate strategy according to the situation, the task characteristics and the requirements of the language. Chi (1988) shows that experts categorise more 'deeply' than novices, focusing on semantics or principals, rather than surface features and syntax. This categorization and refinement fits neatly with the top down processing model. As mentioned by Adelson & Soloway (1988), it makes intuitive sense for aspects of a solution to be at roughly the same level of definition if a simulation is to be performed and Voss and Post (1988) agrees that experts need to be skilled in decomposition. However, there are some exceptions, for example Davies (1991) shows that opportunistic jumps sometimes take place particularly where the subject is familiar with the problem domain. Early in the process these jumps tend to take place between the same levels of abstraction, but later vertical jumps are more in evidence. Adelson (1984) show experts managing these jumps by noting them down and then continuing their top-down processing which accords with the expert meta-cognitive abilities discussed above.

Expert software developers' use of tools appears to differ considerably from that of novices. Davies (1993) shows that experts use tools more strategically than novices and Detienne (1997) finds that experts in system design choose their strategy according to the problem. These studies imply that experts are selective about their use of tools and are able to make decisions based on the features and applicability of a particular tool, rather than taking a prescriptive approach.

With regards to distributed cognition, software development has traditionally been considered as a type of 'disembodied' process or solitary task performed by an individual. However, more recently, studies have begun to consider that this approach may only tell part of the story (Flor and Hutchins 1991)

Again, the strategy by which developers store and access information and participate with the environment around them is seen to be a key factor in the development of programming expertise.

2.2.3 Practice

Adelson (1984) quotes occasions where software experts do not appear to have access to a detailed representation of how they come to know what they do. Presumably this is a symptom of having practiced a skill to such an extent that access of a detailed step-by-step model is not required. Moreover, performing 'automatically' might assume that where a skill is well-practised it may be performed almost 'on auto-pilot'. In studies of object-oriented programmers, Pennington, Lee and Rehder (1995) found that experts were so well practiced that they were able to identify the classes

required directly from the problem domain without resorting to the gradual refinement of detail required by novices.

2.2.4 Metacognition

Petre (2002) notes that on occasions experts perform 'automatically' or 'intuitively' but are able to retrospectively reason about how they have reached a decision. In these cases there must be awareness of their knowledge (Payne, 1988) even thought they do not appear to have gone through a linear, logical process of deduction. This seems similar to the ability to reference certain key pieces of knowledge by recognizing the relevant pattern or trigger to access it and, alongside the selective use of tools discussed above, suggests that experienced software developers are aware of their approach to problem solving and can select tools that are suitable to this approach.

2.2.5 The practice-strategy-knowledge-metacognition model

The different aspects of practice, strategy and knowledge are clearly referenced in the literature on the psychology of programming, in particular where programming expertise is considered. As such, at a high level, this model seems as applicable to expertise in software development and expertise in general.

2.3 Expertise in pair programming

2.3.1 Knowledge

Considering the literature on pair programming in extreme Programming, it quickly becomes apparent that much is written about interpersonal and attitudinal skills, rather than on programming knowledge. For example, Dick and Zarnett (2002) consider Communication, Confidence and Comfort pair programming are the three most important factors when selecting pair programming personnel, and Williams and Kessler (2003) suggests seven habits of effective pair programmers, surprisingly none of which relate to technical skills and knowledge. This might be due to the pair programming community making assumptions about programming expertise, and simply considering the additional skills required. This is not surprising when we consider that it is not possible to be an expert pair programmer without also being considered an experienced programmer, although the reverse is certainly true. However, this makes it unclear what is means when pair programmers consider 'novices' – are these novices inexperienced at programming in general or simply not yet expert at pairing? Auer and Miller (2002) suggests that the benefits of pair programming include all design decisions involving at least two brains and spreading knowledge throughout the team, however little attention is paid to what this means in cognitive terms, how it might be achieved, nor indeed where this knowledge comes from in the first place.

2.3.2 Strategy

It is often stated that when pair programming the driver (who currently has control of the keyboard) and the navigator (who contributes verbally) work at different levels of abstraction (e.g. (Beck, 2000). This suggests that working with two different strategies assists in developing quality code, and is an interesting contrast with findings by Chi (1988) discussed above, which suggests that focusing on semantics, rather than syntax is a more expert strategy. However, a pilot study (Bryant, 2004) suggests that this might not be indicative of the interactions that might be observed when a pair program together – in fact one might question how it would even be possible for two people working at different levels of abstraction to successfully sustain a conversation at all.

Influenced by the work of Hutchins (1995), studies of experienced pair programmers 'in the wild' have shown software development as taking place within a rich ecology of distributed cognition composed of the programmers themselves working amongst and overhearing other members of their team and making use of a wide variety of tools and artifacts within the environment in which they work (Bryant, 2005).

2.3.3 Practice

Little importance seems to be given to amount of practice in the extreme Programming literature, although reference is often made to the 'experienced' and 'inexperienced', without qualifying whether this refers to experience pairing, experience programming, experience within a particular technical environment or experience in the relevant problem domain and what would quality one to be

considered ‘experienced’ in any of these measures. One can only assume that length of time programming or pair programming might certainly play a role.

2.3.4 Metacognition

Very little has been written regarding pair programming and meta-cognition. However, within extreme Programming reference is often made to software development as a ‘reflective practice’ (Schon, 1983) and one often finds references to Myers-Briggs personality tests on pair programmers blogs and websites. Myers Briggs types have even been investigated as a method of choosing compatible pairs in education (Katira, Williams et al. 2004). As previously mentioned, confidence has been considered a key factor in successful pair programming. One might assume that without a reasonable level of confidence the exposure of ones work and methods required when working collaboratively would be uncomfortable at very least. Similarly, lack of confidence on either partner’s part might lead to an ‘unbalanced pair’ in which the less confident partner is retiscent, or indeed unwilling to contribute. Further investigation is required in order to ascertain the role of meta-cognition in successful pair programming and whether a deeper understanding of ones own working style, perhaps bia Myers Briggs or similar inventories, might assist.

2.3.5 The practice-strategy-knowledge-metacognition model

The different aspects of practice, strategy, knowledge and meta-cognition are much less clearly referenced in pair programming literature. In fact, strategy in terms of communication and the mechanics of working together seems to be the main aspect of focus. Perhaps one assumes a certain level of programming competency when considering ability to pair program, or alternatively maybe interpersonal skills and attitude are considered so important that programming skills are considered only secondary.

3 Rating expertise

This section will consider different manners in which expertise might be rated – either by self, peer or superior. It draws on literature from psychology to suggest some expected characteristics of each kind of rating.

3.1 Self-rating

Studies by Dunning and Kruger (1999) across three domains (humour, logical reasoning and grammar) suggest that those who are less skilled are more likely to grossly over-estimate their ability and experts are more likely to underestimate theirs. Their studies suggest that novices lack the experience - and therefore the insight - to recognise their own limitations, while experts do not under-estimate their own performance, but rather over-estimate that of their peers. This concurs with findings which suggest that those who are experienced have greater meta-cognitive skills than those who are not – that is, expert pair programmers may be more aware of the limitations in their knowledge, but more able to compensate for them and select tools and techniques accordingly. Novice pair programmers will over-estimate their skill-level because, not only are they unclear about the skills required, they lack insight into their own performance. Note that when exposed to the work of their peers, experts are more likely to re-assess their rating favourably, bringing it more closely in line with their actual performance. This suggests that experts are not under-rating their own abilities, but rather assuming a higher level of competence from their peers.

3.2 Peer and superior rating

Amongst studies on rater bias, Holzbach’s (1978) survey of 107 managerial and 76 professional employees in a manufacturing company found that self-ratings were more lenient (i.e. significantly different to the ratings from different sources) than either peer or supervisor ratings, whilst peer and supervisor ratings did not differ appreciably either from other such ratings of the same type or across groups. However, peer and supervisor ratings were more likely to be global judgements, whereas self-ratings were more likely to differentiate between rating items. Love (1981) compared peer nominations, ranking and ratings of 145 police officers and 33 supervisors. His findings showed that

the ratings showed significant reliability, and therefore were not significantly biased by friendship etc. Given the studies by Dunning and Kruger (1999) discussed above, these findings suggest that peer and supervisor ratings achieve a more accurate result than self-rating.

4. Rating and collaborative programming

This section discusses a study of rating and pair programming in light of the literature discussed above. First, it considers whether patterns in self-rating according to expertise follow those generally found when contrasted with peer or supervisor rating. Second, it considers how self and peer/supervisor ratings relate to experience in terms of length of tenure (a direct measure of 'practice' from the practice-strategy-knowledge model) and confidence (one of the measures from the pair programming literature).

4.1 Hypotheses

Evidence provided from the studies on rating discussed above lead us to the following predictions:

Prediction 1: The self-assessments of ***experience*** of pair programmers rated high ***ability*** by others will be considerably lower than the ***ability*** rating attributed to them.

Prediction 2: The self-assessments of ***experience*** of pair programmers rated low ***ability*** by others will be considerably higher than the ***ability*** rating attributed to them.

Prediction 3: The pair-programming ***ability*** ratings of peers/supervisors will more accurately reflect the length of ***time*** software developers have been pair programming than their own ratings of their ***experience***.

Prediction 4: Given findings on self-efficacy, one would expect that pair programmers' own ***experience*** rating reflected the ***confidence*** rating they were assigned by their peers/supervisors.

4.2 The study

These predictions were explored in a study across four commercial software development projects. In each study participants were asked to rate their own pair programming ***experience*** level and the amount of ***time*** they had spent pair programming. A colleague or supervisor was then asked to rate their level of pair programming ***ability*** and their level of ***confidence***. In total 43 software developers responded. As peer and supervisor assessments have been seen to be more reliable than own ratings, they will be considered the 'benchmark' rating, therefore when pair programmers are said to be 'high', 'medium' or 'low' ***ability*** this will refer to their supervisor/peer assessment.

4.3 Self, peer and supervisor assessments

Following prediction 1, and in line with findings on expertise and meta-cognition, 'high ***ability***' pair programmers' (as rated by peer or superior) were expected to rate themselves as medium or low ***experience***. Similarly, according to prediction 2, 'low ***ability*** pair programmers' were expected to rate their own ***experience*** level as either medium, or high. The actual findings regarding peer/supervisor ***ability*** ratings and own ***experience*** ratings are given in the table 1 below.

Self-rating \ Peer/supervisor rating	High ***ability***	Medium ***ability***	Low ***ability***
High experience	6	1	0
Medium experience	15	9	2
Low experience	8	1	2

Table 1 – Self, peer and supervisor ratings of pair programming ability and experience

As can be seen from in Table 1, the study findings are in line with the first prediction, as 79% of 'high ***ability***' pair programmers under-rated their level of ***experience*** compared to the rating of their peer or supervisor. Regarding the second prediction, only fifty percent of 'low ***ability***' rated pair programmers rated themselves higher than they were rated by others. One possible explanation for this might be that the improved feedback and confidence gained by working in pairs may help to compensate for the lower ***ability*** pairer's overconfidence. That is, insight into the complexities and skills involved in attaining competence in pair programmers are rendered visible through working with a partner. Alternatively, ***experience*** alone may not have been considered the most key factor in rating ***ability***.

4.4 Length of tenure and ability rating

In our third prediction we suggested that ***ability*** ratings made by peers or supervisors would closely relate to the amount of time an individual had been pair programming commercially (their 'length of tenure' as a pair programmer). Exposure to a variety of pair programming experiences and time in the job are often considered important factors in gaining experience in pair programming. In particular, one of the benefits of pair programming that is often cited is its facilitation of 'learning by doing' and 'learning through observation' (e.g. Williams and Kessler, 2003, p.5). As such, one would expect length of tenure to have a strong impact on ***abilty*** rating by peers and supervisors.

As shown in Figure 1, findings from the study do not concur with predictions. Perhaps most surprising is the fact that, of 27 programmers with less than one year pairing, 19 were already considered to be highly able. This would seem to relate more to their time programming in general (15 had been programming more than five years and all longer than two) than the time spent pair programming. This fits with the general literature on expertise, in terms of repetition and gaining, storing and accessing information than with the attributes considered important in pair programming.

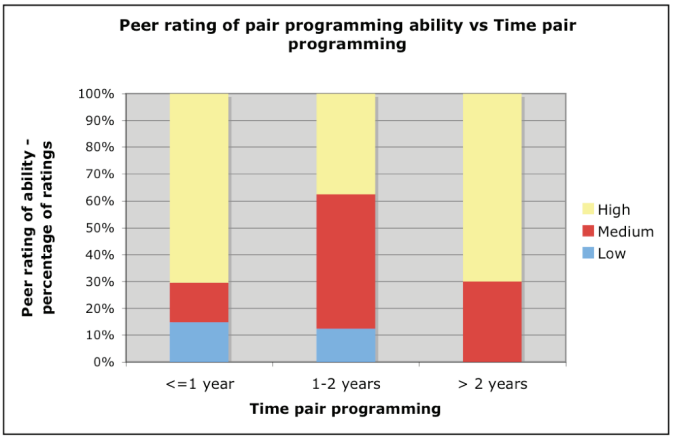

Figure 1 – Peer rating of ability and length of tenure

When compared with the impact of length of tenure on self-rating of pair programming ***experience*** (Figure 2) it is interesting to see that the amount of ***time*** spent pairing had greater impact on self-rating than on peer or supervisor. In fact, most programmers with over two years pairing considered themselves highly ***experienced***, all of the 'medium duration' pairers rated their ***experience*** as medium, and even those with less than a year's pairing rated themselves more consistently than their peer or supervisor ***ability*** ratings. Thus, length of time pairing would seem to be a more important factor when self-rating than for rating others, where general programming tenure seems more important. This is surprising, as length of pair programming tenure is perhaps a more easily quantifiable 'fact' on which a reliable rating could be based. This suggests that in the case of pair programming either self-ratings are

more reliable than peer or supervisor ratings or length of tenure alone is not a reliable indicator. Evidence in other areas, as previously discussed, suggests that it is unlikely that self-ratings are more reliable. Therefore, it would seem that, although it is often used, length of tenure alone is not a 'good' (i.e. reliable) indicator of 'expertise' in this case.

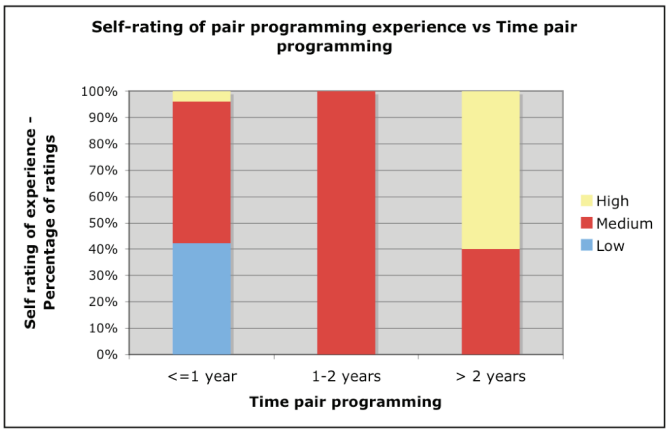

Figure 2 – Self-rating of experience and length of tenure

4.5 Confidence and self-rating

According to the theory of self-efficacy, one would expect that pair programmers of a high level of ***confidence*** were more likely to be successful and therefore more likely to consider themselves more ***experienced*** at pair programming. In order to thoroughly consider this prediction, it is necessary to ascertain the level of congruence between own rating of ***experience*** and peer/supervisor rating of ***confidence***. This may allow us to start to differentiate between high or low self-rating (which could be considered a good measure of self-confidence) and high or low ***confidence*** assessment from a third party (measuring perceived self-confidence). As seen in figure 3 below, there are discrepancies between these two forms of confidence rating in the study results.

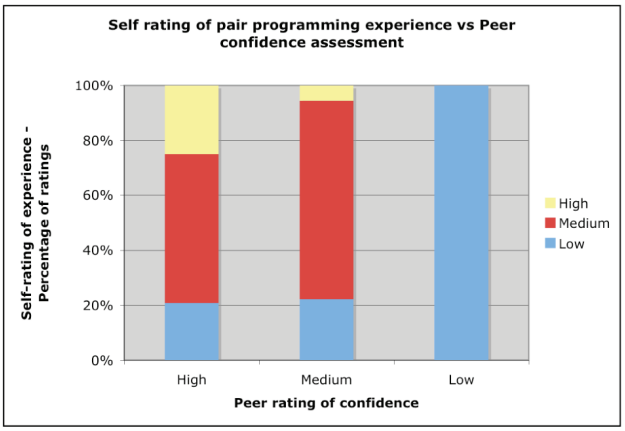

Figure 3 – Self-rating of experience vs Peer rating of confidence

As seen in figure 3, participants in the 'low' and 'medium' confidence rated groups rated their own level of ***experience*** quite consistently with the ***confidence*** rating of their peer or supervisor. However, those considered highly confident by others usually did not consider themselves highly ***experienced***. This is consistent with the data discussed under length of tenure, where a high self-rating was less evident than a high peer-rating. Perhaps this also concurs with Dunning and Kruger's (1999) suggestions that those with high ability tend to over-estimate the ability of those around them. Perhaps

indeed, this encourages them to ‘raise the benchmark’ of what they consider highly able, in some cases so high that it reaches a level they can no longer attain.

5. Conclusion

Knowledge, strategy and practice can clearly be identified from the literature as some of the key components in attaining expertise across a number of domains. however they are referenced surprisingly rarely in the literature on pair programming. Discussions of pair programming expertise are generally limited to ‘more’ and ‘less’ experienced individuals, without clarification of what is meant by these classifications and by whom they are rated. Rather the pair programming literature considers communication and attitude related skills much more dominantly. The survey of commercial pair programmers discussed in this paper showed that, consistently with cross-domain studies, pair programmers considered highly able by others tended to under-rate their level of experience, whereas 50% of those rated low ability over-rated themselves in comparison. This is perhaps due to the rather poor definition of what makes an experienced pair programmer. Thus two further variables were considered: Length of tenure and confidence.

It was somewhat surprising that self-ratings related more closely to the amount of time spent pair programming than ratings by peers or supervisors. This implies that despite not being present in the literature on pair programming, practice is considered a relevant indicator of expertise to pair programmers when rating themselves. Conversely, confidence rating was not a good indicator of self-rated experience in highly able pair programmers, despite being highlighted as an important factor in the pair programming literature. This implies that there are very different rating models being used by peers and supervisors than by pair programmers self-rating. Further studies are required to ascertain in more detail what these models are and how they relate to the key factors of knowledge, strategy and practice.

References

Adelson, B. (1984). "When novices surpass experts: the difficulty of task may increase with expertise." Journal of experimental psychology: Learning, memory and cognition **10**(3): 483-495.

Adelson, B. and E. Soloway (1988). A model of software design. The nature of expertise. M. Chi, R. Glaser and F. MJ. Hillsdale, New Jersey, Lawrence Erlbaum Associates**:** 185-208.

Auer, K. and R. Miller (2002). Extreme Programming Applied. Indianapolis, IN, USA, Addison-Wesley.

Beck, K. (2000). Extreme programming explained: Embrace change, Addison Wesley.

Bryant, S. (2004). Double Trouble: Mixing quantitative and qualitative methods in the study of extreme programmers. Visual languages and human centric computing, Rome, Italy, IEEE COmputer Society.

Bryant, S. (2005). Distributed cognition in pair programming. In preparation.

Chase, W. G. and K. A. Ericsson (1982). "Skill and working memory." The psychology of learning and motivation **16**: 1-58.

Chi, M. (1988). Introduction. The nature of expertise. M. Chi, R. Glaser and M. Farr. Hillsdale, New Jersey, Lawrence Erlbaum Associates**:** 185-208.

Davies, S. (1991). "Characterizing the program design activity: Neither strictly top-down nor globally opportunistic." Behaviour & Information Technology **10**(3): 173-190.

Davies, S. (1993). Expertise and display-based strategies in computer programming. People and Computers VIII - HCI '93 conference.

Detienne, F. (1990). Expert programming knowledge: A schema-based approach. Psychology of Programmin. J. Hoc, T. Green, R. Samurcay and D. Gilmore. London, Academic Press**:** 205-222.

Detienne, F. (1997). "Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams." Interacting with Computers **9**: 47-72.

Dick, A. and B. Zarnett (2002). Paired programming and personality traits. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering.

Dunning, D. and J. Kruger (1999). "Unskilled an unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments." Journal of personality and social psychology **77**(1999): 1121-1134.

Eraut, M. and B. du Boulay (2002). Developing the attributes of medical professional judgement and competence: a review of the literature. Cognitive science research papers, University of Sussex.

Ericsson, K. and P. Polson (1988). A cognitive analysis of exceptional memory for restaurant orders. The nature of expertise. M. Chi, R. Glaser and M. Farr. Hillsdale, New Jersey, Lawrence Erlbaum Associates.

Flor, N. and E. Hutchins (1991). Analyzing distributed cognition in software teams. Empirical studies of programmers: Fourth workshop, Ablex publishing corporation.

Freedman, E. and P. Shah (2002). Towards a model of knowledge-based graph comprehension. Diagrammatic representation and inference. M. Hegarty, B. Meyer and Narayanan**:** 18-30.

Gick, M. and K. Holyoak (1985). Analogical problem solving. Cognitive Modelling. A. Aitkenhead and J. Slack. Hillsdale, New Jersey, Lawrence Erlbaum Associates.

Gilmore, D. (1990). Expert programming knowledge: A strategic approach. Psychology of programming. J. Hoc, T. Green, R. Samurcay and D. Gilmore. London, UK, Academic press**:** 224-234.

Holzbach, R. L. (1978). "Rater bias in performance ratings: Superior, self- and peer ratings." Journal of applied psychology **63**(5): 579-588.

Hutchins, E. (1995). Cognition in the wild. Cambridge, MA, The MIT Press.

Johnson, E. (1988). Expertise and decision-making under uncertainty: Performance and progress. The nature of expertise. M. Chi, B. Glaser and M. Farr. Hillsdale, New Jersey, Lawrence Erlbaum Associates**:** 209-228.

Katira, N., L. Williams, et al. (2004). On understanding compatibility of student pair programmers. SIGCSE technical symposium on Computer science education.

Kintsch, W. (1998). Comprehension: A paradigm for cognition. Cambridge, UK, Cambridge University Press.

Larkin, J. (1983). The role of problem representation in physics. Mental models. D. Gentner and A. Stevens. Hillsdale, New Jersey, Lawrence Erlbaum Associates.

Lawrence, J. (1988). Expertise on the bench: Modelling magistrates judicial decision making. The nature of expertise. M. Chi, B. Glaser and M. Farr. Hillsdale, New Jersey, Lawrence Erlbaum Associates**:** 229-260.

Lesgold, A., H. Rubison, et al. (1988). Expertise in a complex skill: Diagnosing x-ray pictures. The nature of expertise. M. Chi, B. Glaser and M. Farr. Hillsdale, New Jersey, USA, Lawrence Erlbaum Associates**:** 311-342.

Love, K. G. (1981). "Comparnison of peer assessment methods: Reliability, validity, friendship bias and user reaction." Journal of applied psychology **66**(4): 451-457.

Payne, S.J. (1988). Methods and mental models in theories of cognitive skill, in "Characterizing the prgram design activity: neither strictly top-down nor globally opportunistic", Davies, S.P. (1991), Behaviour and Technology 10(3): 173-190.

Pennington, N., Lee, A. and Rehder, B. "Cognitive activities and levels of abstraction in procedural and object-oriented design", Human-Computer Interaction 10: 171-226.

Petre, M. (2002). Mental imagery, visualisation tools and team work. Second program visualisation workshop, Hornstrup centret, Denmark.

Petre, M. and A. Blackwell (1999). "Mental imagery in program design and visual programming." International Journal of Human-Computer Studies **51**: 7-30.

Schmidt, H. (1993). "Problem-based learning: An introduction." Instructional Science **22**(4): 247-250.

Schon, D. A. (1983). The reflective practitioner: How professionals think in action. USA, Basic Books, Inc.

Soloway, E., B. Adelson, et al. (1988). Knowledge and processes in the comprehension of computer programs. The nature of expertise. M. Chi, B. Glaser and M. Farr. New Jersey, USA, Lawrence Erlbaum Associates**:** 129-152.

Suwa, M. and B. Tversky (2002). External representations contributing to the dynamic construction of ideas. Diagrammatic representation and inference. M. Hegarty, B. Meyer and N. Narayan**:** 341-343.

Trafton, J., S. Marshall, et al. (2002). Extracting explicit and implicit information from complex visualisations. Diagrammatic representation and inference. M. Hegarty, B. Meyer and N. Narayan**:** 206-220.

Voss, J. and T. Post (1988). On the solving of ill-structured problems. The nature of expertise. M. Chi, B. Glaser and M. Farr. Hillsdale, New Jersey, USA, Lawrence Erlbaum Associates**:** 261-286.

Williams, L. and R. Kessler (2003). Pair programming illuminated. Boston, Addison-Wesley.