PPIG Newsletter – September 2006

XP and Pair Programming practices

Sallyann Bryant, Benedict du Boulay and Pablo Romero

IDEAS laboratory, University of Sussex, Falmer, UK

1. Introduction
Over the past ten years or so Extreme Programming has been slowly gaining acceptance as a potentially beneficial software development technique both in the commercial and academic worlds. Here we review the existing literature to attempt to gain insight into the practice of pair programming across both educational and commercial contexts. We begin by considering the pair programming team and the environment in which pairing takes place. We then review a number of studies aimed at measuring the costs and benefits of programming in a pair, particularly in terms of the quality of the software produced. Finally we discuss the potential cognitive benefits of pair programming and consider when and where its use might be most appropriate.

2. Extreme Programming
Extreme programming is a methodology that has been slowly gaining acceptance and interest within the programming community over the last ten years. It is seen as an antidote to the large, up-front design methodologies commonly found within the software industry. Extreme Programming (or 'XP' as it is commonly known) stemmed from the working practices of a team of programmers working for Chrysler on a payroll system project known as 'C3'. This team is said to have amplified the working practices that they found useful and enjoyable, and dropped those that they did not. XP falls under the umbrella term of 'agile methodologies'. According to the Agile Alliance (www.agilealliance.org), an agile approach is one that values:

> "Individuals and interactions over processes and tools,
> Working software over comprehensive documentation,
> Customer collaboration over contract negotiation,
> Responding to change over following a plan".

As such, XP focuses on short development cycles with regular delivery of the new versions of software to the customer. In the most recent edition of Beck's 'white book' on XP (Beck and Andres, 2004) there are thirteen main or 'primary' practices and eleven 'corollary' practices. The primary practices mainly relate to either team working (sit together, whole team, energised work, pair programming), communication (informative workspace, stories), project management (weekly cycle, quarterly cycle, slack), or programming practices (10 minute build, continuous integration, test-first programming and incremental design). Here we consider the practice of pair programming.

3. Pair Programming
Pair programming has been defined as 'two people working at one machine, with one keyboard and one mouse' (Beck, 2000). Within the extreme Programming framework, a pair is not usually fixed, that is, a programmer does not tend to work with the same partner all the time. More usually, a pair will work together for the duration of a single task that might most often take a day or two to develop. While some projects empower the programmers themselves with responsibility for deciding when to change pair, others enforce rotating pairs on a regular basis.

3.1 The driver and navigator roles

The terms 'driver' and 'navigator' describe the role of each programmer. These roles are by no means fixed, rather a programmer may change roles several times within a programming session. The driver is the programmer who currently has control of the keyboard, while the navigator contributes to the task verbally and by other means (see Bryant, Romero and du Boulay, 2006a). The navigator role could be considered something of a mystery. Some suggest that the navigator provides a constant design and code review, others consider him/her to be working at a higher level of abstraction than the driver (Hazzan and Tomayko, 2003). This could also be described as the driver working tactically while the navigator works strategically (Williams and Kessler, 2003). Research on commercial programming teams has also shown that rather than take a 'divide and conquer' approach to a programming task, the driver and navigator work together on each aspect of the problem (Bryant, Romero and du Boulay, 2006b).

3.2 The pair programming team

Commercial pair programming generally takes place within the larger context of a programming 'team'. A number of studies have considered the environment within which pair programming takes place. In particular, Sharp and Robinson (2003) focus on XP in their ethnographic work, providing useful insights into what it means to be a member of an XP team in a number of different commercial companies. Kessler and Williams (2003) consider pair programming as providing a form of 'legitimate peripheral participation' (see Lave and Wenger, 1991). Here, a novice programmer can learn the 'craft' of programming by working as part of a programming pair, playing a useful but controlled part in the production of software while being immersed in the project environment, in which (s)he can learn through observation. Similarly, Bryant, Romero et al. (2006) discuss the advantages of the pair's conversation making their work visible to others on the team. They cite occasions where overhearing has resulted in advice and guidance from others outside the pair, or indeed, where pairs have been reformed according to whom on the team is best equipped to work on the task at hand.

3. The pair programming environment

Of course, pair programming does not happen in a vacuum. In fact, pair programming has been shown to "take place in the context of a rich environment of artifacts and talk" (Bryant, Romero and du Boulay , 2006a) where tools created for individual use are often re-appropriated by programming pairs, for example the mouse and keyboard are subtly used to help smooth driver-navigator role exchange. In addition, novel artefacts like cuddly toys are reported as used for informal locking mechanisms when integrating new code onto the test machine (Sharp and Robinson, 2003; Bryant, Romero and du Boulay, 2006).

3.4 The potential benefits of pair programming

A number of studies have considered the effectiveness of pair programming. While some have looked at team morale and attitude towards pair programming (Benedicenti and Paranjape, 2001; Rumpe and Schroder, 2002; Pulugurtha, Neveu et al., 2002; Johnson, Mao et al., 2003), the majority of studies tend to consider whether projects with pair programming show improved quality or speed (e.g. Canfora, Cimitile et al., 2005). Here we consider studies which have attempted to identify the extent to which programming in pairs affects the quality of the resultant software in commercial and academic environments.

3.4.1 The pair programming student

A number of studies have found pair programming useful academically (e.g. Macias 2002; Noll and Atkinson 2003; Tessem 2003). Probably the most cited study aimed at assessing this is the study described in Williams, Kessler et al. (2000). In this between-subjects study 13 university students chose to work on a project individually while 28 other students worked in pairs. The findings showed that code produced in pairs passed more automated tests over four different programming exercises as shown in Table 1.

Exercise	Individual	Pair
Program 1	73.4 %	86.4 %
Program 2	78.1 %	88.6 %
Program 3	70.4 %	87.1 %
Program 4	78.1 %	94.4 %

Table 1. Percentage of tests passed by individually or pair coded programs - Williams et al. (2000).

Nevertheless, one might question the extent to which can it be ascertained whether the two groups were in fact comparable. It is possible that the more successful students might be more likely to have insight into the advantages of collaborative working, or were more aware of their strengths and weaknesses, and therefore be more likely to volunteer for pair programming. Similarly it is possible that when working in pairs the most able student is doing most of the work, therefore it may not be appropriate to compare a pair score with an individual score, but might be more useful to compare the score of a pair working collaboratively with the highest score of a randomly selected 'pair' of students working individually. Finally, given that the pairs were students, learning affects should also be taken into consideration.

3.4.2 The Pair Programming Practitioner

There are also a number of positive studies concerning commercial pair programming (see Benedicenti and Paranjape 2001; Deias, Mugheddu et al. 2002; Rumpe and Schroder 2002; Ambu and Gianneschi 2003).

Lui and Chan (2003) performed experiments with experienced software developers. In their second experiment participants took algorithm-style aptitude tests on their own and in pairs. In pairs 85% of the responses were correct, compared to only 51% when working alone, however one might question the extent to which this task is comparable to an increase in software quality in a commercial computing environment. This result also underlines some of the issues raised above with regards to student pairs.

A further study by Nosek (1998) considered fifteen full-time, experienced software developers working on challenging problems in their usual working environment. The report gives a clear and reliable definition of how quality was measured (a score of 0-2 for readability and a score out of 2 for each of 3 output requirements). Inter-grader reliability was 90%. The results showed that pair teams significantly outperformed individuals on program quality. Whilst not stated explicitly in the documentation, one can assume that the participants were not already experienced in pair programming, as Nosek (1998) states that they 'were somewhat sceptical of the value of collaboration...and thought that the process would not be enjoyable'. This suggests that results were significant despite the required change in practice for those working in pairs.

Finally, Jensen (2003) describes an experiment that took place in an organisation developing a complex real-time systems in Fortran. A single project trial of pair programming produced code with a 127 percent gain in productivity and an error rate three orders of magnitude less than those on similar projects. The programmers rated the pair designed code as 'better quality', although there is no clear definition of what this referred to. Despite the study not being a direct comparison with an alternative project, the development environment and methodology were the same as the other projects, and the experience of the team was considered 'about average' for the organisation.

Taken as a group, these studies almost unanimously support the hypothesis that pair programming leads to better quality software. However, taken individually each study has its methodological limitations. Therefore, while on one hand one might suggest that the variety of environment and methodology makes the confirmatory nature of the results all the more compelling, on the other there is still call for a single, rigorous confirmatory study.

3.4.3 Cognitive benefits

There are a number of potential cognitive benefits to pair programming, which may help to understand, at least in part, how the reported gains in quality and speed may be obtained. The presence of a second programmer may help to minimise confirmation bias (Hutchins 1995). This is a phenomenon whereby an individual is more likely to filter information, focusing on that which confirms their current hypothesis, and discarding potentially useful and important information that does not. In fact, Williams and Kessler (2003) allude to this by claiming that pair programming lessens the likelihood of 'tunnel vision'. Similarly it is possible that the mere presence of the navigator forces the driver to stay 'on task' and ensures that (s)he conforms to quality standards etc.

Another cognitive benefit may simply be that working in a pair encourages a programmer to talk. There is evidence to suggest that this type of verbalisation alone may result in improved understanding. That is, it has been shown that simply by talking effectively to oneself ("self-explanation"), one might achieve a greater level of understanding and create a more correct mental model of the problem (Chi, de Leeuw et al. 1994). This is further explained by Ainsworth and Loizou (2003) who consider verbalisation a kind of 'cognitive off-load', freeing up working memory. In fact, many programmers anecdotally report occasions when they have had a 'eureka'moment on a difficult problem while explaining it to somebody else. It appears that these benefits are not dependent on the object being spoken to so much as the person talking. For example, there have been reports of the benefits of talking to a cardboard cut-out of your favourite "programming guru" (Portland pattern repository, in Williams and Kessler, 2003).

4. When to pair program?

While many studies have made progress in understanding what pair programming (and indeed XP as a whole) involve, there is still some debate as to its global applicability. Pairing is tiring (see Tessem, 2003 and Sharp and Robinson, 2003), perhaps due to the additional cognitive load of articulating one's work while one performs it. Most programmers do not pair together for the whole of the working day. In fact regular breaks and a week of limited working hours are seen as an important part of the extreme programming approach. In its 'purest' form, it is assumed that all code is written in a pair and that time outside the pair is spent on administrational or communication tasks (for example, checking and responding to email). It has, however, been suggested (Ambu and Gianneschi, 2003) that pairing might be impractical when deadlines are tight and studies by Benedicenti and Paranjape (2001), Becker-Pechau, Breitling et al. (2003) and Gittins, Hope et al. (2001) all suggest that pairing is not used exclusively, but could more usefully be introduced in a non-mandatory fashion, perhaps focusing its use on a project's most critical or complex tasks. These conflicting approaches suggest that deciding the extent to which programmers work in pairs might be dependent on a number of factors. These may include the problem domain, the company culture, deadline pressures, personal preference and problem complexity.

5. Remote pairing

There is some debate regarding the applicability and practicality of 'remote pair programming', some claim many of the benefits of pair programming are degraded if the members of the pair are not co-located, in fact in the latest version of XP, one of the primary practices is 'Sit together'. Presumably this is in part due to the fact that the talk generated when pairing has the additional benefit of making the work of the pair transparent to the whole team. In addition, it has been suggested that pair programming better caters for our human needs as essentially social creatures. However, others claim that remote pairing allows one to benefit from both the improved quality of pairing and the convenience of working 'wherever'. There are some reports of remote pairing having worked commercially (see Harrison, 2003 and Kircher, Jain et al., 2001) and some have even developed tools to support remote pairing. For example, Stotts, McC.Smith et al. (2004) produced the Transparent Video Face-top, which provides the remote pairer with an overlaid image of the face of their programming partner on the screen. This is thought to assist in providing the subtle visual cues and facial expressions which are otherwise unavailable to the remote programming pair. Similarly, Hanks (2004) provides a tool to help overcome the problem of floor control and provide a referencing mechanism when remote pairing through the use of an additional 'pointing only' mouse. This mouse (seen as an iconised pointing finger) can be used to highlight a code object under discussion on the screen without the coordination problems that would be associated with having two 'regular' mouse users. Here we have only considered the most commonly accepted form (according to the literature) – co-located pair programming.

6. Conclusion

Pair programming is one of the core practices of the XP methodology and a number of studies, although limited, almost unanimously suggest that pairing improves software quality. This is reinforced when one considers the potential cognitive benefits, some of which are discussed above. However, a number of papers report on the intense and exhausting nature of pair programming and recommend its use in a more measured fashion, perhaps making pairing non-mandatory or focusing its use on a projects most complex or critical tasks. Similarly, further studies are required if we are to fully understand pair programming, in particular its applicability to complex projects or those in which programmers work remotely.

References

Ainsworth, S. and A. T. Loizou (2003). The effects of self-explaining when learning with text or diagrams. Cognitive Science 27: 669-681.

Ambu, W. and F. Gianneschi (2003). Extreme programming at work. Fourth International conference in Extreme Programming and Agile Processes. Lecture Notes in Computer Science (2675). Springer : 347-350.

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

Beck, K. and C. Andres (2004). Extreme Programming Explained: Embrace Change - 2nd Edition. Upper Saddle River, NJ, USA, Pearson Education.

Becker-Pechau, P., H. Breitling, et al. (2003). Teaching team work: An extreme week for first year programmers. Proceedings of the 4th International conference in Extreme Programming and Agile Processes.

Benedicenti, L. and R. Paranjape (2001). Using extreme programming for knowledge transfer. Proceedings of the 2nd International Conference on eXtreme Programming and Agile Processes in Software Engineering.

Bryant, S., P. Romero and B. du Boulay (2006a). Pair programming and the re-appropriation of individual tools for collaborative software development. 7th International Conference on the Design of Cooperative Systems, Carry-le-Rouet, France. 'Cooperative Systems Design' in Frontiers in Artificial Intelligence and Applications (137), IOS Press: 55-70.

Bryant, S., P. Romero and B. du Boulay (2006b). The Collaborative Nature of Pair Programming in Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2006) - (in press).

Chi, M., N. de Leeuw, et al. (1994). "Eliciting self-explanations improves understanding." Cognitive Science 18: 439-477.

Canfora, G., Cimitile, A. and Visagio, C (2005). Empirical study on the productivity of the pair programming, Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering, Sheffield, UK. Baumeister, Machesi, Holcombe (eds). Springer-Verlag, Berlin, 2005: 92-99.

Deias, R., G. Mugheddu, et al. (2002). Introducing XP in a start-up. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering. Springer: 62-65.

Gittins, R., S. Hope, et al. (2001). Qualitative studies of XP in a medium sized business. Second International conference on eXtreme Programming and Flexible Processes in Software Engineering, Villasimius, Sardinia, Italy.

Hanks, B. F. (2004). Distributed pair programming: An empirical study. XP/Agile Universe 2004, Salt Lake City, USA.

Harrison, N. (2003). A study of extreme programming in a large company. Avaya labs, 2002.

Hazzan, O. and J. Tomayko (2003). The reflective practitioner perspective in eXtreme Programming. XP Agile Universe 2003, New Orleans, Louisiana, USA. Springer : 51-66.

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

Jensen, R. (2003). "A pair programming experience." The journal of defensive software engineering 16(3): 22-24.

Johnson, S., J. Mao, et al. (2003). Extreme makeover: Bending the rules to reduce risk rewriting complex systems. Fourth international conference in Extreme Programming and Agile Processes in Software Engineering.

Kircher, M., P. Jain, et al. (2001). "Distributed extreme programming." Second international conference on eXtreme Programming and Agile Processes in Software Engineering.

Lave, J. and E. Wenger (1991). Situated learning: Legitimate peripheral participation. New York, NY, USA, Cambridge university press.

Lui, K. and K. Chan (2003). When does a pair outperform two individuals? Fourth international conference in Extreme Programming and Agile Processes in Software Engineering. Lecture Notes in Computer Science. Goos, G., Hartmanis, J. and J van Leeuwen (eds). Springer: 225-233.

Macias, F. (2002). Empirical experiments with XP. Third international conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002), Sardinia, Italy: 225-228.

Noll, J. and D. Atkinson (2003). Comparing extreme programming to traditional development for student projects. Fourth International conference on extreme programming and agile processes in software engineering. Springer: 2675.

Nosek, J. T. (1998). "The case for collaborative programming." Communications of the ACM 41(3): 105-108.

Pulugurtha, S., J. Neveu, et al. (2002). Extreme programming in a customer services organisation. Third international conference on Extreme Programming and Agile Processes in Software Engineering, Sardinia, Italy. Springer: 193-194.

Rumpe, B. and A. Schroder (2002). Quantitative survey of extreme programming projects. Third international conference on Extreme Programming and Agile Processes in Software Engineering, Alghero, Italy. Springer: 95-100.

Sharp, H. and H. Robinson (2003). An ethnography of XP practices. in Proceedings of the Fifteenth annual psychology of programming interest group workshop (PPIG/EASE 2003), Keele University, UK: 15-27.

Stotts, D., J. McC.Smith, et al. (2004). Support for distributed pair programming in the transparent video facetop. XP/Agile Universe 2004, Calgary, Canada, Spring-Verlag.

Tessem, B. (2003). Experiences in learning XP practices: A qualitative study. Fourth International conference on extreme programming and agile processes in software engineering. Springer LNCS 2675: 131-137.

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

Williams, L., R. Kessler, et al. (2000). Strengthening the case for pair programming. IEEE software 17(4): 19-25.