The ‘Tag Team’: Tools, tasks and roles in collaborative software development

Sallyann Freudenberg (née Bryant)

Submitted for the degree of D. Phil
University of Sussex
December 2006
Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:
Acknowledgements

A belated nod to those who encouraged me to start down this road: Dr. Alan Eardley of Staffordshire University, Ian Skerrett who emailed me an article on XP because he “thought it was the kind of thing I’d be interested in” and Adam Bushell, whose meals, friendship and encouragement helped to sustain me in the early days.

Huge gratitude to my supervisors, Benedict du Boulay and Pablo Romero, for all their valuable help, support and advice. In particular they have been key in helping to make the ‘journey’ such an enjoyable one with their encouragement, enthusiasm and good humour. I could not have hoped for better mentors. Thanks to current and past members of the IDEAS laboratory with whom it has been my pleasure to work and from whom I have received so much helpful feedback and advice. In particular, grateful groveling to Dr. Amanda Harris for reviewing, statistical assistance and being an excellent listener, and to Rowanne Fleck and Katerina Avramidis for reviewing.

Now for my nearest and dearest: first my son, Isaac Yoshi, whose arrival in the middle of this work taught me so many wonderful lessons and who continues to amaze me every day; my husband, Andrew Freudenberg, for his patience, encouragement, strength, and understanding (and for often making me laugh against the odds); Nan, my mum and my two lovely sisters Jayne Brown and Tessa Bryant for their continued support in everything I do. All my wonderful friends, sorry I don’t have room to name you all.

Finally, I’d like to thank the anonymous participants in my studies, who I cannot name, and the companies for which they worked, who I can: Egg, BBC iDTV project, BNP Paribas and LogicaCMG.
Preface

This research was funded by a three-year grant from the Engineering and Physical Sciences Research Council (EPSRC). The project was supervised by Professor Benedict du Boulay and Doctor Pablo Romero at the University of Sussex.

The work described in this thesis has been presented at the following conferences and workshops:

- The 16th Annual Workshop of the Psychology of Programming Interest Group in Carlow, Ireland (Bryant, 2004b).
- The IEEE Symposium on Visual Languages and Human-Centric Computing in Rome, Italy (Bryant, 2004a).
- The 17th Annual Workshop of the Psychology of Programming Interest Group in Brighton (Bryant, 2005).
- The International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2005) at Sanibel Island, Florida (poster session) (Bryant, Romero, & du Boulay, 2005).
- The 7th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2006) in Oulu, Finland (Bryant, Romero, & du Boulay, 2006b).

It has also been published in:

- The Journal of the Computer Society of India (Bryant, du Boulay, & Romero, 2006).
- The International Journal of Human-Computer Studies (Freudenberg, du Boulay & Romero, in press).

In addition, aspects of this work have been presented to:

- The Empirical Studies of Software Development group at The Open University.
- The British Computer Society (Sussex Branch)
- The Systems and Software Research Group, Hertfordshire University
- The COGS group and the Human-Centred Technology group at The University of Sussex.

The proverbs cited at the beginning of each chapter of this thesis are taken from ‘The Little Zen Companion’ by David Schiller (1994).
The ‘Tag Team’: Tools, tasks and roles in collaborative software development

Sallyann Freudenberg

Summary

Computer programming is highly challenging. Since its inception a wide range of approaches, tools and methodologies have been developed to assist in managing its complexity. Collaborative working is a common practice in many disciplines and over the last ten years or so working in pairs has been put forward as an alternative to the traditional model of programming alone. Studies have suggested that ‘pair programming’ has a positive effect on software quality, however ideas regarding how and why have been largely speculative.

The research presented in this thesis considers how pair programming is realised and seeks evidence of factors contributing to its reported positive effects. Three main analyses of experienced pair programmers are presented. The first considers the role of the environment, the second considers the extent to which the pair collaborates and the third looks at the behaviours associated with the ‘driver’ and ‘navigator’ roles.

Findings suggest that while pairing creates a number of overheads, these are managed through the subtle use of existing tools and artefacts. In addition, the transparency provided through over-hearing pair conversations provides a mechanism for advice seeking and optimal pair organisation.

Results show that each member of a programming pair contributes to each aspect of the problem, irrelevant of role. They also suggest that speculations in the literature on the strongly differentiated nature of the driver and navigator role are unfounded.

This leads us to suggest an alternative understanding of the programming pair, which we have called the ‘tag team’. In addition, we have produced a set of ‘pair programming behaviours’ which we believe will be useful for pair programming practitioners, for those considering introducing collaborative working and for the teaching and learning of programming.

Thesis submitted for the degree of Doctor of Philosophy
University of Sussex
December 2006
## Contents

1. Introduction.................................................................................................................. 1  
   1.1. Why collaborate?.................................................................................................. 1  
   1.2. Extreme programming ...................................................................................... 1  
   1.3. Pair programming ............................................................................................ 3  
   1.4. Studies of student pair programmers .................................................................. 4  
   1.5. Studies of Commercial Pair Programmers...................................................... 6  
   1.6. Aim of this thesis ............................................................................................... 7  
   1.7. What this thesis is not about ............................................................................. 8  
   1.8. Structure of this thesis ...................................................................................... 8  
   1.9. The studies, data and analyses in this thesis ..................................................... 9  
      1.9.1. The studies .................................................................................................. 9  
      1.9.2. The data ..................................................................................................... 9  
      1.9.3. The analyses .............................................................................................. 10  
   1.10. Chronology of thesis work .............................................................................. 11  

2. Issues in the successful realisation of collaborative software development .............. 12  
   2.1. Introduction......................................................................................................... 12  
   2.2. Experience........................................................................................................... 12  
      2.2.1. General expertise......................................................................................... 13  
      2.2.2. Expertise in programming ......................................................................... 16  
      2.2.3. Expertise in Pair Programming .................................................................. 18  
      2.2.4. Rating Expertise ....................................................................................... 19  
   2.3. Tools and setting ............................................................................................... 20  
      2.3.1. Ethnography ............................................................................................ 20  
      2.3.2. Ethnography and the workplace ................................................................. 21  
      2.3.3. Ethnography and programming ................................................................. 21  
      2.3.4. Ethnography and XP/pair programming ................................................... 22  
      2.3.5. External Representations ........................................................................... 23  
   2.4. Level of collaboration ....................................................................................... 24  
      2.4.1. Defining collaboration and co-operation .................................................... 24  
      2.4.2. Collaboration ............................................................................................. 25  
      2.4.3. Co-operation ............................................................................................ 27  
      2.4.4. Collaboration/Co-operation and Pair Programming .................................. 29  
   2.5. Role .................................................................................................................... 29
5. The role of level of collaboration ................................................................. 66
  5.1. Introduction .......................................................................................... 66
  5.2. Background .......................................................................................... 67
  5.3. Method .................................................................................................. 67
  5.4. Results .................................................................................................. 70
    5.4.1. Overall session shape ....................................................................... 70
    5.4.2. Co-operation vs collaboration ....................................................... 71
    5.4.3. Collaboration and task type ............................................................ 72
    5.4.4. Collaboration, role and task type .................................................... 73
  5.5. Discussion ............................................................................................ 74
    5.5.1. Overall session shape ....................................................................... 74
    5.5.2. Co-operation vs collaboration ....................................................... 75
    5.5.3. Collaboration and task type ............................................................ 75
    5.5.4. Collaboration, role and task type .................................................... 75
    5.5.5. Study limitations ............................................................................ 76
  5.6. Conclusion ............................................................................................. 77
    5.6.1. Further work .................................................................................. 78

6. The role of role .......................................................................................... 79
  6.1. Introduction .......................................................................................... 79
  6.2. Method .................................................................................................. 80
  6.3. Background .......................................................................................... 82
  6.4. Results .................................................................................................. 83
    6.4.1. The pair programming session ....................................................... 83
    6.4.2. Overall level of abstraction ............................................................ 84
    6.4.3. Level of abstraction and role ........................................................... 84
    6.4.4. The navigator as ‘reviewer’ ............................................................. 85
    6.4.5. The navigator as ‘foreman’ ............................................................... 86
  6.5. Discussion ............................................................................................ 87
    6.5.1. The navigator as reviewer ............................................................... 87
    6.5.2. The navigator as foreman ............................................................... 88
    6.5.3. The ‘Tag Team’ – An alternative perspective ................................... 88
    6.5.4. The proliferation of talk about ‘Chunks of Code’ ............................... 89
    6.5.5. Study limitations ............................................................................ 89
  6.6. Conclusions .......................................................................................... 90
7. Pair programming behaviours

7.1. Introduction

7.2. Desirable ‘expert’ pairing behaviours

7.2.1. The ‘Cohabitor’

7.2.2. The ‘Relocator’

7.2.3. The ‘Tag Team’

7.2.4. The ‘Intermediate level of talk’

7.2.5. The ‘Re-pair’

7.2.6. The ‘Break’

7.2.7. The ‘Enforcer’

7.3. Undesirable pair programming behaviours

7.3.1. The ‘Thrasher’

7.3.2. The ‘Divider’

7.4. Absent pair programming behaviours

7.4.1. The ‘Reviewer’

7.4.2. The ‘Foreman’

7.5. Conclusion

8. Conclusions and Future Work

8.1. Introduction

8.2. Summary of this research

8.3. Findings

8.3.1. Contributions

8.3.2. Implications for future studies

8.3.3. Implications for practitioners

8.3.4. Implications for pair programming environments

8.3.5. Implications for teaching and learning collaborative software development

8.3.6. Implications regarding our understanding of pair programming

8.4. Limitations

8.4.1. Methodological limitations

8.4.2. Limitations of findings

8.5. Future work

8.5.1. Considering the effect of verbalisation

8.5.2. Comparing agile and conventional approaches

8.5.3. Investigating the ‘tag team’
8.5.4. Investigating intermediate level talk .......................................................... 126
8.5.5. Defining further pair programming behaviours ....................................... 128
8.5.6. Informing programming education ......................................................... 128
8.5.7. Fostering meta-cognition ...................................................................... 128
8.6. Conclusion ................................................................................................. 128
9. Bibliography ................................................................................................. 130
10. Appendix A – Experience questionnaires .................................................... 139
A.1 Participant questionnaire .......................................................................... 139
A.2 Peer/supervisor questionnaire .................................................................. 140
Appendix B – Sub-tasks .................................................................................. 141
B.1 Example sub-task decomposition diagram .............................................. 141
B.2 Number of contributions of new information to each generic subtask type by observed session ........................................................................................................................................... 142
B.3 Number of subtasks of each type to which new information was contributed by only one participant .................................................................................................................................. 143
B.4 Number of subtasks of each type to which new information was contributed by only one role ............................................................................................................................... 144
List of Tables

Table 1.1  Tests passed by individually or pair coded programs (Williams, Kessler, Cunningham, & Jeffries, 2000) ........................................................................................................................................... 4
Table 1.2  Readibility and functionality scores by individually or pair coded programs ........ 6
Table 3.1  Initial and refined candidate pair programming behaviours ............................... 38
Table 3.2  Table for capturing candidate pair programming behaviours ............................ 39
Table 4.1  Profiles of sessions observed .................................................................................. 47
Table 4.2  Artefacts and the pair programming activities they facilitate ............................... 64
Table 5.1  Derived generic sub-tasks ..................................................................................... 69
Table 5.2  Example coding of dialogue .................................................................................. 69
Table 5.3  Percentage of sub-tasks of each generic type that were collaborative across participants ........................................................................................................................................ 72
Table 6.1  Scheme for coding utterances by level of abstraction ......................................... 81
Table 6.2  An example section of coding .............................................................................. 82
List of Figures

Figure 3.4  Interaction frequency per one hour session according to pair programming expertise ................................................................. 40
Figure 3.5  Interaction type according to pair programming expertise ................................................................................................. 41
Figure 3.6  Example session showing interactions when each programmer is driving in a low expertise pair ........................................................................................................ 41
Figure 3.7  Example session showing interactions when each programmer is driving in a high expertise pair ........................................................................................................................................... 42
Figure 4.1  Physical layout of desks and chairs in Project B .......................................................................................................................... 50
Figure 4.2  Physical layout for Project C ......................................................................................................................................................... 51
Figure 4.3  Physical layout for Project D, Team 1 .............................................................................................................................................. 52
Figure 4.4  Physical layout for Project D, Team 2 .............................................................................................................................................. 52
Figure 4.5  Example of proximity facilitating peripheral awareness through over-hearing .......................................................................... 54
Figure 4.6  Example of proximity facilitating peripheral awareness through name-dropping ................................................................ 54
Figure 4.7  An example of dialogue during keyboard hand-over ............................................................................................................... 55
Figure 4.8  An example of navigator use of the mouse for pointing .......................................................................................................... 56
Figure 4.9  Examples of the code being used to continue conversation ....................................................................................................... 56
Figure 4.10  An example of the code being used to supplement verbalisation .............................................................................................. 57
Figure 4.11  An example informal external representation ........................................................................................................................ 58
Figure 4.12  An example exchange about creating an external representation .............................................................................................. 58
Figure 4.13  An example list ........................................................................................................................................................................... 59
Figure 4.14  The surrogate mouse ........................................................................................................................................................................ 60
Figure 5.1  Example verbalised subtask decomposition ................................................................................................................................. 68
Figure 5.2  Distribution of contributions amongst generic sub-tasks ........................................................................................................... 70
Figure 5.3  Average relative contributions for each generic subtask type .................................................................................................. 73
Figure 5.4  Percentage each role contributed to each generic subtask type ................................................................................................... 74
Figure 6.1  Average number of utterances of each level of abstraction in the sessions considered (error bars showing highest and lowest values) .................................................................................................................. 84
Figure 6.2  Normalised SY level utterances by role for each session .............................................................................................................. 85
Figure 6.3  Chart showing theoretical levels for utterances by the driver and navigator were they to work at different levels of abstraction ................................................................................................................ 86
Figure 6.4  Chart showing actual levels for utterances by the driver and navigator ............................................................................................ 87
1. Introduction

Old pond, frog jumps in – plop!
Matsuo Basho (1644-1694)

1.1. Why collaborate?

Computer programming is a cognitively taxing activity. Not only is it difficult because of a lack of direct manipulation (Blackwell, 2002) and a ‘product that no-one can see’ (Perry, Staudenmayer, & Votta, 1994, p.178) but due to many other factors. These include complexity, the need for a multi-layered, multi-dimensional model capable of supporting mental simulations, and the sheer amount of knowledge required, its suitable organisation and mechanisms for its access.

One possible method of taming the complexity of software development may be to work collaboratively. In fact, one form of collaborative programming has now been formalised as ‘pair programming’, one of the core practices of the Extreme Programming (XP) methodology. In pair programming, “all production code is written with two people working at one machine, with one keyboard and one mouse” (Beck, 2000).

1.2. 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’ (Fowler, 2006). 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 the regular delivery of new versions of software to the customer. In the most recent edition of the ‘white book’ on XP (Beck & 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).

When using XP, whenever possible the software development team work with a full-time on-site customer, developing systems in an evolutionary manner rather than using the traditional waterfall approach.

XP requires a development ‘iteration’ to be defined as a number of ‘stories’. It is recommended that an ideal iteration (including release of one or more stories) should take between one and four weeks. These stories are rough, high level specifications written on a set of story cards, each of which is given to a technical team to estimate the effort it will take to complete (if this is not possible, the story is split into smaller stories). Each story must be related to a business problem that exists today, potential future requirements are not included. This complements the views of Visser and Hoc (1990) that specifications given at the start are never complete or unambiguous. The stories are then sorted by priority and risk and the acceptance tests for each are defined by the business. If necessary a story may then be split into tasks, each of which is detailed on a task card. A task is then passed to or selected by a pair programming team (see 2.3). The pair first set about defining and automating a ‘unit’ test or set of tests that will prove the new functionality has been successfully implemented. The tests are run against the current system, which of course fails, as no development has begun.
This approach is known as ‘test first’ because the automated tests are written before the coding begins.

Once the newly developed code passes all the unit tests, it is loaded onto an integration machine that contains the most recent working version of the system. Here a full set of automated acceptance tests are run, including the new unit tests specific to this new functionality, which have been added. If any test does not pass the programming pair must either fix the code or remove their new code, leaving the integration machine in a state where the full set of tests pass once more.

1.3. Pair programming

When a pair are programming together they are typically referred to as being in one of two roles: the “driver”, who is currently using the peripherals (i.e. the mouse and keyboard) to manipulate the computer, or the “navigator”, who contributes to the activity verbally. Pairs are dynamic and can - in fact many say should - change between activities in order to maximize the spread of system knowledge throughout a project team.

While the driver seems to be fulfilling the ‘traditional’ role of typing in the code, the navigator role could be considered something of a mystery. We will be discussing some of the claims about these roles in Section 2.5 and investigating them further in Chapter 5.

A number of publications have considered the costs and benefits of pair programming with regard to development time and software quality in one form or another. The majority of these have either been studies in an academic environment or experience reports from practitioners. The studies have compared pair programming favourably with programming alone in terms of quality of software produced and side effects such as decreased ‘tunnel vision’ and positive ‘pair pressure’ (e.g. Williams et al. 2000),

Here we consider a number of studies which have attempted to identify the extent to which programming in pairs affects the quality of the resultant software. These studies have been grouped into those that consider students in an academic environment and those that consider commercial software developers working in industry. The academic studies will be considered first, however, the extent to which they generalise may be questioned in terms of the skills and experience of the participants, the nature of the tasks, and the environment in which the studies
take place. The commercial studies will then be discussed, building on the academic studies to give a broader picture.

1.4. Studies of student pair programmers

Probably the most cited study claiming to assess the benefits of pair programming is the academic study that took place at the University of Utah, which is described in Williams, Kessler, Cunningham and Jeffries (2000). In this between subjects study 13 university students chose to work on a project individually while 28 other students chose to work on the same project in 14 pairs. The findings showed that significantly (p < 0.01) more automated test cases passed in code produced in pairs over four different programming exercises. The results are shown in more detail in Table 1.1 below. Nevertheless, one might question the extent to which it can be ascertained whether the two groups were in fact comparable. It is possible that more successful students might be more likely to have insight into the advantages of collaborative working, or may be more aware of their strengths and weaknesses, and therefore more likely to volunteer for pair programming. Similarly it is possible that when working in pairs the most able student is simply doing most of the work.

Table 1.1 Tests passed by individually or pair coded programs (Williams, Kessler, Cunningham, & Jeffries, 2000)

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Individual</th>
<th>Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>73.4</td>
<td>86.4</td>
</tr>
<tr>
<td>Program 2</td>
<td>78.1</td>
<td>88.6</td>
</tr>
<tr>
<td>Program 3</td>
<td>70.4</td>
<td>87.1</td>
</tr>
<tr>
<td>Program 4</td>
<td>78.1</td>
<td>94.4</td>
</tr>
</tbody>
</table>

This study is discussed further in Cockburn and Williams (2001), which states that there were about 15% fewer defects in the programs produced by pairs, and that these programs were shorter than those produced by solo programmers, implying an improvement in internal as well as external quality.

Macias (2002) was able to access a larger pool of participants in two studies with students working on projects with real clients. All students were given equal access to the client, and at the end of the project the client chose which solution to implement. The first study consisted
of 80 second year degree students and the second study ran over two consecutive years, during which 58 students, either on Masters courses or on the final year of their degree courses, participated. This study was not solely concerned with pair programming, but with comparing eXtreme Programming to a traditional methodology, thus pair programming may be only one of the factors potentially contributing to the results. Nonetheless, two of the three clients considered that the eXtreme Programming team had produced software with better external quality, and all lecturers considered that the eXtreme Programming teams had produced software with better internal quality, although it is not clear how this was measured. It is also unlikely that the clients and lecturers were blind to the methodology used, particularly as the clients were actively involved in the development process and the lecturers were overseeing the projects.

Less statistically compelling are academic studies by Keenan and Coleman (2003) and Noll and Atkinson (2003). Note that these studies were again not solely interested in pair programming and the potential improvement in quality. Nevertheless they remain interesting in that they add to the body of evidence in support of Williams et al. (2000). In the study by Keenan and Coleman (2003), 36 students who wrote test cases in pairs were asked to assess their completeness. 54% considered their test cases felt `more complete' than those produced when working alone, 29% considered them `as complete' and only 7% considered them `less complete'. There is no discussion of what was meant by the `completeness' of the test cases and in fact this simply confirms what one would expect to find in collaborative work of any kind. In Noll and Atkinson (2003) two teams were given a number of `stories' to work on. A story is the XP terminology for a rough, high-level specification that is typically written on a small card (see Section 1.2). The two teams that used eXtreme Programming completed fewer stories (three compared to eight when using a traditional methodology) but the stories that were completed were much more robust. As we are not considering time, but merely quality, we must consider this study as further evidence towards a quality improvement when using eXtreme Programming, in which pair programming must surely play a part.

The sole study the author has found that contradicts Williams et al. (2000) findings with students is a within subject study by Puus, Seeba, Salumaa and Heiberg (2004), in which 176 students completed questionnaires after being exposed to programming both alone and in pairs. When pair programming, the students were significantly less satisfied with the resultant program than when working alone. However it is worth considering that this may not be
indicative of a difference in the quality of the software produced, but may perhaps relate solely to the student's `sense of ownership' of the program.

### 1.5. Studies of Commercial Pair Programmers

In 2003, Lui and Chan (2003) performed a number of experiments with experienced software developers. These experiments considered both time and quality. Here the second experiment will be discussed, as it was the most quality-focused and did not consider how long it took to produce a completed response. The participants took part in algorithm-style aptitude tests, both on their own and in pairs. When working in pairs 85% of the responses were correct, compared to only 51% when working alone, however one might question the extent to which this activity is comparable to an increase in software quality in commercial computing.

Despite the limited numbers considered, a within subject study by Nosek (1998) provides further evidence that pair programming does indeed improve software quality. This is perhaps the most compelling evidence because the study was of fifteen full-time, experienced software developers working on challenging problems in their usual working environment. The report also gives a clear and reliable definition of how quality was measured. The metrics used were a score of 0-2 for readability (2 being entirely readable and 0 not at all readable) and a score out of 2 for each of 3 output requirements, thus an overall potential score of 8. Inter-grader reliability was 90%. The report does not, however discuss the development approach taken, therefore it is not clear whether or not the projects were using an agile methodology. The results are shown in Table 1.2 below, and a two-sided t-test indicated 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 makes the evidence very compelling as the results were significant despite the required change in practice for those working in pairs.

<table>
<thead>
<tr>
<th></th>
<th>Individual</th>
<th>Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readibility</td>
<td>1.40</td>
<td>2.00</td>
</tr>
<tr>
<td>Functionality</td>
<td>4.20</td>
<td>5.60</td>
</tr>
</tbody>
</table>
Finally, Jensen (2003) describes an experiment that took place in an organisation developing complex real-time systems in Fortran using a conventional (i.e. not agile) development methodology. A one-project trial of pair programming produced code with a 127 percent gain in productivity (measured in lines of code per person-month) 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.

1.6. Aim of this thesis

Each of the studies discussed above has its weaknesses, either in terms of the extent to which it might generalize to commercial programmers, the lack of stringent definition of terms (e.g. quality) or the number of participants involved. However, taken as a group, these existing studies almost unanimously support the hypothesis that pair programming leads to better quality software. In fact, one might suggest that the very variety of environment and methodology makes the confirmatory nature of the results all the more compelling.

Despite these reported benefits, the cognitive aspects of pair programming are seldom investigated and little understood. The ethnographic studies reported in Sharp and Robinson (2003) and Robinson and Sharp (2004) provide an insightful story of XP in a commercial environment, but do not assess pair programming from a cognitive perspective. In fact there have been a number of calls for further investigation of the types of interactions that affect the reported benefits of pair programming (Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999) and when and why it is effective (Chong et al., 2005). Hughes and Parkes (2003, p.137) even propose a suitable method by which ‘the cognitive processes underlying productivity and quality gains can be formally mapped rather than speculated about’.

The aim of this thesis is therefore to begin to understand how pair programming might generate its reported benefits. In particular, it aims to consider how pair programming is practically realised and what is special or different about the environment, behaviours and practices of successful pair programmers. It addresses four distinct research questions:
• How can we best distinguish between ‘expert’ and ‘novice’ pair programmers?
• How does the use of tools, setting and the environment facilitate successful collaborative software development?
• What is the level of collaboration (as opposed to co-operation) in experienced pair programming sessions?
• What are the defining aspects of the reported driver and navigator roles?

1.7. What this thesis is not about
This thesis does not aim to provide evidence that collaborative software development is more effective than individual programming. In particular, it does not attempt to suggest, design or perform the ‘ultimate’ experiment regarding the effect of pair programming on software quality. The aims of this thesis are not related to the provision of a tool for collaborative software development, whether remote or collocated. Nor does it suggest a prescriptive manner in which to teach, or indeed perform, pair programming.

1.8. Structure of this thesis
This thesis is organised by gradually refining the focus of the studies presented. It consists of eight chapters. In Chapter Two we report on existing research in the field of the psychology of programming. In particular we discuss general issues in expertise, tools and setting, task and role and then focus more keenly on their application to programming and finally pair programming. We also provide background for the methods and techniques used.

In Chapter Three we derive a suitable method of delineating between ‘experienced’ and ‘novice’ pair programmers and provide some evidence that there is indeed something ‘different’ or ‘special’ about experienced pairs. Using the derived delineation, data from a set of four ethnographic studies of ‘experienced’ pairs were analysed with three different foci: In Chapter Four we look at the environment, tools and setting in which experienced pair programming is realised; Chapter Five considers the different ways in which the pair might divide up the task at hand, and Chapter Six investigates some of the claims regarding the ‘driver’ and ‘navigator’ roles. In Chapter Seven we define a set of ‘pair programming behaviours’ that summarise the behaviours we found and those that we sought, but did not
find. We conclude with a summary of the work presented, a discussion of its implications and limitations and some suggestions for further work in the area.

1.9. **The studies, data and analyses in this thesis**

1.9.1. **The studies**

The information reported in this thesis relates to four, one-week studies of pair programmers in four separate companies. All of the companies used an agile approach (Beck et al., 2006), and several of them used eXtreme Programming (Beck, 2000). The studies took place in the workplace, with the programmers working on typical activities. During these four studies a variety of information was gathered in an attempt to respond to the thesis questions identified above. Twenty-four sessions were observed, transcribed and analysed. Each session was an hour long and a total of 45 programmers participated in the studies. As pair composition switched frequently and the organization of pairs and their work was not determined by the studies, some individuals were observed in more than one pair. However, any particular pair was observed working together for at most two one-hour sessions and any individual a maximum of four times. In total 18 different pair combinations were observed. These sessions were not evenly distributed across the four studies, rather three were from the first study, six from the second, seven from the third and eight were from the fourth study.

The programmers studied were doing a variety of activities, which included providing new functionality, solving existing problems, porting functionality across to new systems and refactoring (re-structuring the design of part of the system to be more usable or intelligible or to minimise duplication, as described in Fowler, Beck, Brant, Opdyke, & Roberts, 1999). Nearly all of the sessions included both code comprehension and code generation activities as on no occasion was a solution being coded in isolation – there was always an existing system being worked on.

1.9.2. **The data**

In response to our first question regarding how best to distinguish between more and less ‘experienced’ pairs, during the first study, fourteen participants of mixed pair programming experience were observed in detail for one-hour pair programming sessions. Their behaviour
was categorised by the observer according to a number of pre-reported behaviours expected to be found in programming pairs.

Additional information was gathered from all four of the studies, including this initial pilot. This data took several forms. With a view to our second question, regarding the role of tools and the environment, a series of field notes was kept by the author for all four of the studies. These field notes included general observations on emergent themes and notes regarding informal interviews that were conducted. The field notes were augmented by further data in the form of photographs of the project environments and copies of external representations made or referred to.

Regarding our final two questions concerning the level of collaboration and the meaning of the roles in pair programming, verbal data was gathered from all four studies in the form of voice recordings of the utterances made by the pair programmers during one-hour pair programming sessions. This amounted to the collection of a total of 14,886 sentences.

1.9.3. The analyses

In response to our first question regarding the level of pair programming experience at which ‘expert’ or ‘special’ behaviours are observed, the number of occurrences of each of the expected pairing behaviours was analysed according to a classification of ‘novice’ or ‘expert’ based on our initial informal survey. In particular, changes in behaviour were assessed according to changes of role between the driver and navigator. This was performed in order to ascertain whether there appeared to be marked differences between role change and behaviour in more and less experienced pairers.

In response to our second question, regarding the impact of the tools and environment in which pair programming takes place, a different type of analysis was called for. In order that the findings emerged from the environment, rather than focusing on a set of pre-defined theories, the analysis approach was based on ethnographic techniques. In particular, and in keeping with ‘Grounded Theory’, the author attempted to view everything as ‘strange’. Field notes were kept detailing occurrences of any behaviours that seemed interesting or noteworthy. At the end of each day of study the field notes were analysed and emergent themes were identified. These themes then became the focus of the future days of observation and
analysis, in particular both confirmatory and disconfirming evidence was sought. Chapter four describes the findings of this analysis.

Due to the more specific nature of the final two questions (regarding level of collaboration and role) and the limitations of the data (in particular that in most cases recording was limited to audio) a third type of analysis was applied. As reported in more detail in Chapters five and six, verbal protocol analysis was used. Two separate sets of codings were derived that specifically targeted the types of information required by the research question at hand. Following validation of the coding schemes, all 14,886 sentences were coded according to each scheme, allowing statistical analyses to be applied.

1.10. Chronology of thesis work

The work for this thesis began in October 2003. The initial pilot study took place over one week at the end of February 2004 and the data from that study was initially coded and analysed in the six-month period that followed. The three other one-week studies took place in October and November the same year. The recorded sessions from these three sessions were transcribed in the first half of 2005, and the coding schemes were also derived during this period. The transcriptions from all four studies were coded according to these schemes at the end of that year, following a four-month break in the project.

Analysis of the derived data began at the start of 2006 and the production of this thesis document took place between June and December 2006.
2. Issues in the successful realisation of collaborative software development

Somebody showed it to me and I found it by myself
- Lew Welch

2.1. Introduction

As discussed in Chapter 1, while the benefits of pair programming are said to be plentiful, the factors involved in obtaining them remain elusive. This thesis therefore focuses on the behaviours and conversations of experienced pair programmers in an attempt to uncover the components of a ‘successful’ pair programming session.

In order to achieve this focus, it is necessary to first broadly understand the nature of expertise, and in particular expertise in software development and how it might be rated. The remainder of this chapter then considers in turn the tools and setting in which pair programming is practically realised, the potential division of tasks between the programming pair and the assigned roles of ‘driver’ and ‘navigator’. For each of these three areas we consider the existing literature, beginning with a general overview and then focusing first towards programming and then more specifically pair programming. We also discuss appropriate methods by which these might be studied and expressed. This provides the basic theoretical results, models and methodologies that will later be applied.

2.2. Experience

Before homing in on collaborative software development experts, it is necessary to consider how to define expertise. In this section, the literature on expertise in general, in software development and in pair programming is used to ascertain common themes that may be useful
when identifying and assessing expertise. We then go on to consider how one might appropriately assess an individual’s level of pair programming expertise.

2.2.1. General expertise

In this section we will discuss expertise in a general sense. However, while the generalizations and comparisons in this section assume a standard concept of expertise, one should not forget that, as reported in Bryant (2004a), each of the following contributory studies relates to a specific subject group doing a specific activity.

Expertise and the question of what constitutes expert behaviour is a subject much discussed within the Artificial Intelligence (A.I.) arena. However, much of this literature relates to expertise in a particular area or performing a particular activity or type of activity. Of particular relevance here, due to its more general nature, is the classification system suggested by Dreyfus, Dreyfus and Athanasiou (1986) in their ‘five steps from novice to expert’. The five steps they identify are:

1. Novice: Follows simple instructions and recognises various objective but context-free facts and features.
2. Advanced beginner: Begins to recognise meaningful elements, but cannot define them.
3. Competence: Organises and inter-relates elements according to a plan.
4. Proficiency: Organises and understands work intuitively, but thinks analytically about what to do.
5. Expertise: Skill has become so much a part of them, they are not aware of it. Usually ongoing and non-reflective but when required will deliberate before acting.

While this classification is useful in identifying the ‘learning curve’ by which expertise is attained, it is less helpful in identifying tangible, observable features by which an ‘expert’ may be identified in a particular field. The five stages do, however, clearly indicate the presence of five factors that we will now consider in the wider literature on expertise: General approach; Knowledge; Strategy; Practice and Meta-cognition.
**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 & du Boulay, 2002). This may stem from an underlying confidence in their ability, suggesting that self-confidence plays a prominent role in expert behaviour.

**Knowledge and practice**

Studies have identified some common characteristics regarding the nature, amount and organization of information in experts 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 & Polson, 1988). Schmidt (1993) tells of ‘encapsulated knowledge’ where the expert has a kind of ‘compiled knowledge’ that 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 the 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 and 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 (Brooks, 1983), focal line (Davies, 1994), or trigger (Vitalari & Dickson, 1983). 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.

**Strategy**

Lesgold, Rubison, Feltovich, Glaser, Klopfer and Wang (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 peculiarities of the problem to ascertain what is special about it. One might assume that this allows the expert to either further pare down the information they have and 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 & 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 than that of novices, 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) finds 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 artefacts rather than by an individual working in isolation. This implies that there is a social aspect to expertise. In particular, Hutchins (1995) cites maritime navigation as a complex activity achieved within a complex ecology of cognition, some of which is embedded in the environment, tools and techniques that are used.
Metacognition

Meta-cognitive insight has been widely observed in experts. For example, Chi (1988) shows that experts self-monitor more, Eraut and du Boulay (2002) state that they are aware of their own biases and Suwa and Tversky (2002) mention that expert architects are aware of the cycle of using external representations for the production of ideas. Ericsson and Polson (1988) also 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 they take and the tools they use in problem solving in a way that novices do not. It might also suggest that those who are more expert are more likely to give an accurate rating of their ability than those with less experience.

2.2.2. Expertise in programming

We will now consider how these general findings regarding expertise and knowledge, strategy, practice and meta-cognition relate more generally to the domain of software development.

Knowledge

Soloway, Adelson and Ehrlich (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). They showed that experts performed well on plan-like problems (those incorporating generic knowledge structures) 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 and Blackwell (1999) further define this model as being particularly rich and ‘stoppably dynamic’. These findings are consistent with general studies in expertise across domains, 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, activity, environment and the programmer herself. In addition to these types of knowledge, another key element in programming expertise is pragmatic knowledge regarding ‘how to be a good programmer’. Some aspects of this are detailed by Hunt and Thomas (1999).
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 activity characteristics and the requirements of the language. Chi (1988) shows that experts categorise more ‘deeply’ than novices, focusing on semantics or principles, rather than surface features and syntax. This categorization and refinement fits neatly with the top-down processing model. As mentioned by Adelson and 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) agree 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) shows experts managing these jumps by noting them down and then continuing their top-down processing which accords with the expert meta-cognitive abilities discussed in Section 2.2.1.

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. 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 regard to distributed cognition, software development has traditionally been considered as a type of ‘disembodied’ process or solitary activity performed by an individual. However, more recently, studies have begun to consider that this approach may only tell part of the story (Flor & 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. In turn, this strategy may be influenced by the structure of a programmer’s knowledge (Davies, 1994).
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, one 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.

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 though 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. Coupled with the selective use of tools discussed above, this suggests that experienced software developers are aware of their approach to problem solving and can select tools that are suitable to this approach.

2.2.3. Expertise in Pair Programming

Considering the literature on eXtreme Programming, it quickly becomes apparent that there is a need for a solid definition of the terms ‘experienced’ and ‘novice’ pair programmer. In fact, rather than defining the factors that delineate the ‘experienced’ from the ‘inexperienced’, the literature tends to focus on how each of these might behave, and what might happen with pairs of the same or mixed levels of experience. For example, Williams and Kessler (2003) devote an entire section of their book on pair programming to the relative merits and challenges of expert-expert, expert-average, expert-novice and novice-novice programming without defining how these classifications are decided.

Nevertheless it is clear that a particular set of behaviours or attributes are ‘expected’ in an experienced pair. In the pair programming literature, interpersonal and attitudinal skills proliferate in discussions of expertise, rather than programming knowledge. For example, in the commercial arena Dick and Zarnett (2002) consider communication, confidence and
comfort pairing to be the three most important factors when selecting pair programming personnel, and Williams and Kessler (2003) suggest seven habits of effective pair programmers, surprisingly none of which relate to technical skills and knowledge. In addition, academic studies which have considered length of programming experience are somewhat contradictory. In a study of 132 pairs, Madeyski (2006) found that the mean of a pair’s programming experience seemed to correlate with external quality, while Muller and Padberg (2004) found the reverse. The pair programming community therefore seems to assume a certain level of programming knowledge and expertise and focus on the additional skills required for pairing. Perhaps 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. Similarly little importance is given to the actual amount of pair programming practice an individual has. Reference is made to ‘experienced’ and ‘novice’ (e.g. Williams & Kessler, 2003) or ‘senior’ and ‘junior’ partners (Beck, 2000; Wake, 2002) without qualifying whether this refers to experience pairing, experience programming, experience within a particular technical environment or experience in the relevant problem domain. In addition, little attention is paid to what would qualify one to be considered ‘experienced’ in any of these measures. In fact, as communication and the mechanics of working together seem to be so important, one might assume that length of time pair programming (‘length of tenure’) may play an important role in obtaining these interpersonal and communication skills and therefore being considered an ‘expert’ pair programmer.

2.2.4. Rating Expertise

Considering rating in general, 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. Amongst other 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. Love (1981) compared peer nominations, ranking and ratings of 145 police officers and 33 supervisors. His
findings showed that the ratings displayed significant reliability, and therefore were not biased by friendship etc. These findings suggest that peer and supervisor ratings achieve a more accurate result than self-rating. In Chapter 3 we will further consider the validity of self- and peer-rating and length of tenure as measures of pair programming expertise.

2.3. Tools and setting

Software development is traditionally thought of as solitary in nature, with the enduring image of the quintessential programmer sat alone battling it out at his computer. In fact, as indicated in many of the studies discussed in Section 2.2, studies in the psychology of programming have tended to use quantitative methods to consider software development at the individual, cognitive level (e.g. Basili, Selby, & Hutchens, 1986; Davies, 1994; Kutar, Britton, & Barker, 2002; Pennington, 1987; Romero, 2003). These types of studies have provided an understanding of the complexities of programming and the methods employed and difficulties encountered by any individual undertaking this endeavour. However, there is now a wider appreciation of the collaborative nature of the production of software and the additional need to understand the social and physical context in which programming takes place (Curtis, 1986; Rogers & Ellis, 1994; Sharp, Robinson, & Woodman, 2000).

Engelmore and Tannenbaum (1990) reported that engineers spent more than fifty percent of their time communicating both formally and informally. Thus researchers have begun to consider the wider context within which computer programming takes place, focusing on on-site studies of commercial programmers doing their day-to-day work and considering the tools, setting and environment as well as the individual and the computer. In particular, the psychology of programming has begun to draw on techniques from anthropology, such as ethnography and distributed cognition (discussed in Sections 2.3.1 and 2.3.2) to gain insight into the rich environment in which programming is practically realised (Petre, 2003; Sharp, Robinson, & Woodman, 2000).

2.3.1. Ethnography

There is a long tradition of ethnographic studies in the field of social anthropology. For example, Malinowski (1922) used ethnography when he lived alongside, studied and documented the lives of islanders in the western Pacific. Such techniques have been widely used and debated ever since. Ethnographers tend to live amongst and immerse themselves within the community they are studying. They often supplement their observational field notes
with other forms of information, including interviews, photographs, the collection of artefacts and quantitative information such as population size and gender distribution. Some of the methodological concerns widely discussed in anthropology include the effect of the presence of the ethnographer and the extent to which ethnographic reports can be anything more than a highly socially and culturally subjective interpretation of events (see Denzin & Lincoln, 2003). Typically attempts to compensate for these issues include triangulating different methods of data collection and analysis to see whether they yield the same results and clearly describing and placing the writer and their position within the report itself.

**2.3.2. Ethnography and the workplace**

One particularly relevant use of ethnography is a maritime study by Hutchins (1995). In studying the distributed manner in which seamen navigate and describing the roles and artefacts involved, Hutchins coined the phrase ‘distributed cognition’, describing an ecology where ‘cognition is distributed and situated in a socio-cultural world’. Ethnographic techniques have been used in many other places of work. For example, Goodwin and Goodwin (1996) studied aircraft control centres; Heath, Hindmarsh and Luff (1999) studied the London Underground and Hutchins and Klausen (1996) took the techniques of distributed cognition and applied them to an airline cockpit.

Several studies also highlight the importance of artefacts: from their studies of a telephone call centre, Ackerman and Halverson (1998) suggest that any organisation’s memory is constructed and maintained by both people and artefacts and Schmidt and Simone (1996) highlight the use of artefacts for coordination.

**2.3.3. Ethnography and programming**

Following interest in the collaborative nature of software development, a number of studies have taken ethnographic techniques to the world of computing. In particular, Teasley, Covi, Krishnan and Olson (2000) describe the special characteristics and benefits of a co-located project team. Rogers and Ellis (1994) suggest the use of distributed cognition as offering particular benefits for the study of computing, as do Sharp, Robinson and Woodman (2000) in their discussion on the use of ethnography to study software development. Similarly, Ball and Ormerod (2000, p.148) suggest that “it remains paradoxical that the majority of existing studies of design expertise have ignored the role of situational and social factors in…preference to carrying out laboratory-style investigations”.

Of particular relevance to this thesis, Flor and Hutchins (1991) applied their distributed cognition analysis to two programmers working together on a software maintenance activity. They recorded both the screen and the two programmers and also captured the programmers’ keystrokes. The study found that even seemingly trivial activities revealed complex sets of supporting activities and identified seven main system level properties: the reuse of system knowledge; The sharing of goals and plans; Efficient communication; Searching through larger spaces of alternatives; Joint production of ambiguous plan segments; Shared memory for old plans and Divisions of labour and collaborative interaction systems. They also showed that while functional roles are defined by the pair (in this case ‘teller’ and ‘changer’), they are only very loosely followed.

#### 2.3.4. Ethnography and XP/pair programming

There are already several examples of the use of ethnographic techniques in studies of extreme programming. In an observational study at a large company, Harrison (2003), looked at which of the XP practices were used and which were not. Chong (2005) performed a set of ethnographic studies in order to consider the differences in social behaviour between XP and non-XP teams. They found that teams using XP tended to conform more to a particular set of work routines and practices and that awareness is created and maintained more easily on XP teams due to the dialogue between pairs of programmers being available to the rest of the team. Ethnographic techniques have also been used to compare the interruptions that occurred on pairing and non-pairing teams (Chong & Siino, 2006). This study resulted in the finding that breaks and intrusions on the pairing team tended to be shorter, more functional and more often face-to-face compared to those on the non-pairing team.

Perhaps most relevant here, Sharp and Robinson (2003) used ethnography to study XP teams, identifying five main themes: A shared purpose, understanding and responsibility; The coding and quality of code matters; Sustainability; Rhythm (a daily rhythm and one oriented around the three-week project iterations); and a Fluidity of boundaries. More recently Sharp and Robinson (2006) have used distributed cognition to consider the sources and transformation of information and representations, identifying as key the rich sources of information that surround the programming pair.
In Chapter 4 we present the results of four ethnographically inspired studies of pair programmers. It is hoped that the findings presented will not only provide a strong flavour of the environment within which experienced pair programmers work, but also provide evidence of the many subtle and interesting ways in which tools and techniques originally intended for solo use are re-appropriated and re-interpreted by the programming pairs to assist in subtle role changes, co-ordination and cognitive off-load.

2.3.5. External Representations

Group representations  Nearly all computer systems need to be understood by more than just the person or people who created them. In fact, a wide and varied audience may both help to create, and need to be informed of the progress of any software project at some level – end users, analysts, programmers, user interface designers, technical specialists and marketing departments to name just a few. As such, the role of diagrams and other paper-based external representations in software development is well-documented and key to many development methodologies. In fact, one could say that historically many software development methodologies have relied on formal, external representations to provide stakeholders with a ‘shared model’ of the system at any point in time. Moreover, Curtis (1986) found that having a shared model of the problem influenced a team’s decision-making rate.

Individual representations  The documented benefits of diagrams in general are many, identifying that diagrams are also useful to the individual. These include their ability to “show complexity in a simple, retainable form” (Dogan & Nersessian, 2002), to disambiguate mental representations (Cox, 1999) and to assist in offload, ease problem solving and provide constraints (Scaife & Rogers, 1996).

However, one of the core values of agile projects is the focus on working software rather than producing separate documentation. In particular, the XP methodology downplays the role of formal system architecture or design diagrams. In support of this focus away from a single architecture diagram, there is considerable evidence showing the benefits of less formal types of representation. Informal external representations may allow the externalization and representation of partially formed models, where blank or fuzzily defined areas can co-exist with clear, well-defined aspects. Norman (1983) states that mental models can include knowledge and beliefs thought to be of doubtful validity, therefore perhaps an informal representation allows this dubious information to be included in a working ‘representation in
progress’. In addition, Petre and Blackwell (1999) found that expert systems designers produced very visual, rich and multi-dimensional images, which were used to help explore the problem and produce a solution. It is likely that such images would not conform well to the constraints of a formal representation technique but rather lend themselves to the production of less formal diagrams.

The qualitative analysis in Chapter 4, in particular the section on Informal External Representations, discusses how informal external representations were created and used in the 36 pair programming sessions studied.

2.4. Level of collaboration

There is speculation in the literature regarding the different ways in which pair programmers might divide up the various aspects of a particular task according to their roles as ‘driver’ and ‘navigator’ however little is said about how they might divide up the workload itself. Here we consider two different ways in which a pair might organise their work. These approaches are characterised as: collaboration and co-operation. In this section we first define our use of the terms ‘collaboration’ and ‘co-operation’ and then consider the suggested costs and benefits associated with each approach.

2.4.1. Defining collaboration and co-operation

It has been suggested that collaborative situations may be defined in terms of three factors: interactivity, asynchronicity and negotiability (Dillenbourg, 1999). Similarly it is suggested (Roschelle & Teasley, 1995) that co-operative work is accomplished by the division of labour. That is, were a task completed co-operatively, each party would take a defined sub-set of the task to perform separately (and most likely in parallel). When each part is completed, the solutions would then be slotted back together to provide a complete solution to the sub-task. A useful example of this, given in Kerawalla, Pearce, Yuill, Luckin and Harris (2006) is two people doing a crossword together. In a collaborative approach both people would work together on each and every clue, whereas using a cooperative approach one person might take the horizontal clues while the other considers the vertical clues. Of course, co-operating in this manner necessitates some additional effort in integrating the work of both parties.

At first glance it may be difficult to distinguish between collaboration and co-operation, particularly as, in either case, talk could take place that conforms to the ‘rules’ of a
conversation. Under closer inspection this conversation may be a dialogue (with both parties discussing the same issue) or two inter-leaved monologues (where each party is essentially ‘talking to themselves’ but following the conversational norms involved in a discussion, for example, turn taking).

2.4.2. Collaboration

The potential benefits of collaborating on a task are widely reported: Suthers (2001) suggests that collaboration increases learning, productivity, time focused on the task, knowledge transfer and motivation and Jeong and Chi (2000) show that understanding improves after collaboration - those collaborating on a task learned more than those performing it alone. Here we suggest a number of mechanisms by which these potential benefits might occur and also consider manners in which they may be hindered.

2.4.2.1. Potential benefits

**Peer pressure to perform at one’s best** Working in real-time with a co-located partner is likely to increase one’s motivation. In particular, a programmer might be less likely to spend time ‘off task’, for example, checking their email or surfing the internet, when accompanied by a collaborating partner. The same programmer may also feel more obliged to conform to project standards and take fewer short-cuts when their work is being constantly surveyed by a partner. This is referred to as positive ‘pair pressure’ by Williams and Kessler (2000).

**Reduction in errors due to confirmation bias** It could be suggested that collaboration decreases the probability of confirmation bias (Hutchins, 1995). As Petre (2002) shows, memory can be subject to distortion and create bias. One particularly interesting form of this is confirmation bias, where a coincidental occurrence is incorrectly assumed to validate a hypothesis. This has been the subject of investigations in a number of different fields. Besnard and Greathead (2003) describe the Kegworth air disaster as an example where expertise negatively affected the ability to diagnose a problem. The aeroplane pilot hypothesised that Engine A was faulty when in fact Engine B was the problem. When Engine A was throttled back, there was a coincidental drop in vibration, which the pilot took as confirmation of his hypothesis. The pilot therefore shut down the healthy engine, resulting in a total loss of power and a fatal crash. The article goes on to suggest that in cognitive terms confirmation bias makes sense. Given the cognitive effort involved in creating a mental model, it would seem
logical to wait for consistent data to support it in order to avoid the cost of revising the model. Hutchins (1995) gives further evidence of confirmation bias, this time in expert naval navigators. In software development, the provision of another programmer may assist in providing a mechanism to reduce this bias. Perhaps this positive effect of pairing underlies the phenomenon to which Cockburn and Williams (2001) are alluding when mentioning that pair programming makes a programmer less likely to go for a bad design and decreases tunnel vision.

**Apprenticeship and learning** It may also be suggested that more successful knowledge transfer might occur when collaborating. That is, by being present when one’s partner performs part of a task, one is exposed to not only the task solution, but also to the method by which it was attained. This relates closely to what Lave and Wenger (1991) refer to as ‘legitimate peripheral participation’, that is, not only might a pair programmer learn through ‘doing’ it themselves, but also through ‘seeing and hearing’ those around them perform more advanced tasks in the surrounding area. Williams et al. (2000) also highlight this similarity and agree that pair programming may help provide an apprenticeship environment and Fuqua and Hammer (2003) found that new members who pair programmed ‘got up to speed quickly’ on the commercial programming project they studied.

**Reviewing work for mistakes** Finally, it has been suggested that the navigator constantly reviews the work of the driver, catching spelling and syntax mistakes so that the driver does not need to concern themselves with the activities of reviewing (e.g. Jensen, 2003; Williams & Kessler, 2000) and also making sure that the driver is ‘going in the right direction’. This is discussed further in Section 2.5.3 and explored in Chapter 6.

### 2.4.2.2. Potential costs

**Maintaining a collaborative model** Collaborating on a software development task requires the two parties to maintain a common mental model of the problem at hand. While it is necessary to ensure that both partners have some form of collaborative model, this might not simply be a sum of the complete mental models of each party (see Jeong & Chi, 2000). Work by Petre and Blackwell (1999) suggests that experienced software developers have very rich and complex, multi-dimensional mental models that show additional aspects such as ‘completeness’ or ‘complexity’. They refer to these models as ‘focal images’. In later work Petre (2003) found that when disparate focal images were used by different groups, the focal
images hindered communication and inhibited co-operation when these groups later attempted to work together. This would suggest that the production and maintenance of a common model is necessary if the pair is to collaborate effectively. Not only is this costly in terms of the cognitive processing required to maintain both an individual and collaborative model, but also potentially in terms of the meta-cognition and verbalisation required in order to ensure that the models of each programmer remain in step.

**Verbal overshadowing**

There is evidence to suggest that talking about a subject may stifle creativity. In particular, Schooler, Ohlsson and Brooks (1993) posit that creative thoughts and insights are distinct from language processing and that, as it has been shown that verbalisation can interfere with non-verbal tasks as varied as face recognition, the memory for colour and jam tasting trials, so this might be extrapolated to cover all insight problems. Insight problems are defined as those which have a high probability of an impasse followed by a ‘eureka’ moment and whose solutions often involve a sudden reorganisation of information. Schooler, Ohlsson and Brooks (1993) performed a series of studies to investigate this phenomenon further and assess the limits of its effect. The findings show that verbalisation significantly affects performance on insight problems and does so differently than other similar types of interruption. This suggests that collaborative pair programming may have a negative effect due to it forcing verbal communication even during usually 'silent' insight tasks, whereas a more co-operative approach may help to inhibit discussions that could produce this ‘verbal overshadowing’ effect.

### 2.4.3. Co-operation

#### 2.4.3.1. Potential Benefits

**Time gain**

One obvious projected benefit of a co-operative approach is that by allowing the two programmers to work on different aspects of the task in parallel there should be an associated time gain.

**Exploitation of specialisms**

If each partner takes a separate part of the problem at hand to work on, there is more scope for each to ‘play to his strengths’. That is, if one partner has a particularly high level of expertise in a certain area, they can work on an appropriate part of the problem quickly and accurately without being hindered by their partner.
**Reduction in cognitive load**  Dealing with only part of the task at hand may lower cognitive load on each individual programmer, as they can be safe in the knowledge that they do not need to consider certain issues that are being dealt with by their partner.

**Increased self-explanation**  It is possible that the presence of a partner encourages each programmer to verbalise. There is a body of evidence suggesting that simply talking to oneself helps improve understanding. For example, Chi, de Leeuw, Chiu, and Lavancher (1994) asked a group of students to self-explain each line of a text about physics and showed that self-explanation resulted in the production of a more correct mental model and a higher gain in understanding. Ainsworth and Loizou (2003) suggest that verbalization provides a form of ‘computational off-load’, perhaps putting part of the problem ‘out in the world’ rather than requiring it to be kept ‘in the head’. Ericsson and Simon (1980) state that verbalization provides an intermediate re-coding of information, and that in the process of this recoding, it is necessary to add further information for communication purposes which may itself prove useful. Cox (1999) also shows that translation between modalities (in his work from mental to diagrammatical but here from mental to verbal) improves understanding. This might well be equally applicable to the domain of computing and suggests that simply talking about a software development issue may assist in its understanding and ultimately its resolution. In fact there are a number of accounts of this effect including talking to a cardboard cut-out of your favourite ‘programming guru’ (Portland Pattern Repository, referred to in Williams and Kessler, 2003). While the positive affects related to self-explanation may at first appear contradictory in nature to the negative affects attributed to verbal overshadowing the two phenomena are in fact related to different types of problem. Evidence of positive self-explanation relates to step-by-step comprehension type problems, while verbal overshadowing occurs when working on insight type problems that require a creative solution.

**Reduction in group polarisation**  Finally, a co-operative pair programming approach may also assist in minimizing a phenomenon known as ‘group polarisation’ or more specifically ‘the risky shift’ (Stoner, 1961). This term is used to describe the fact that groups have been shown to have a tendency to make riskier decisions than individuals. For example, Myers and Bishop (1970) found that highly prejudiced students became even more prejudiced after discussing racial issues in a group, whereas unprejudiced students became even less prejudiced. Separating out the programming activity between individuals may avoid this effect and therefore lead to less extreme solutions.
2.4.3.2. Potential costs

**Extra time organising the task**  
One clear cost of a co-operative approach is the time and effort required to define and agree how the task is to be split up. It could be considered that on occasions this might require significant negotiation, in particular when one aspect of the task was seemingly more appealing or less difficult than another or when task boundaries are difficult to visualise.

**Increased integration effort**  
Of course, when a task has been split up in order for each programmer in the pair to perform a particular part, some additional effort will necessarily be involved in integrating those parts when they are complete. Not only might this involve effort in ensuring that the two component parts ‘fit together’, but additional issues, similar to those seen in larger scale systems integration, may also play a part. These include areas such as duplication, version control and standardisation.

2.4.4. Collaboration/Co-operation and Pair Programming

In order to understand how the benefits of pair programming are realised it is first necessary to consider the manner in which a pair divide up the work on their assigned task. In Section 2.4.1 above we discussed two distinct ways in which a pair might divide up their task: co-operation and collaboration. In Sections 2.4.2 and 2.4.3 we have also shown that each of these approaches has a number of related costs and benefits.

We indicated in that a surface approach to analysing the talk in which a pair engage may not be sufficient to distinguish between co-operation and collaboration. In Section 2.5.4 we describe a more detailed and systematic method of analysing talk: verbal protocol analysis. Chapter 5 is then dedicated to the endeavour of applying verbal protocol analysis to pair programming sessions in order to establish the nature by which work on a task is divided between the pair.

2.5. Role

The terms ‘driver’ and ‘navigator’ are used consistently throughout the pair programming literature to describe who currently has control of the keyboard (and often also the mouse). In fact, it is rare to find literature on pair programming that does not mention these roles and attribute some particular behaviours to them. In order to discuss these role-based assumptions
further, it is first necessary to discuss the concepts of ‘domain’ and ‘level of abstraction’ that have been applied to them.

2.5.1. Mapping between domains

Many of the difficulties encountered in commercial computer programming can be attributed at some level to the complexities involved in translating and transforming between the problem domain (that is, the real world in which the programming problem resides) and the programming domain. Of course, these domains are very different in nature, not least in terms of their language, rules, ‘values’ and constraints. Pennington, Lee and Rehder (1995) discuss the difficulties involved in maintaining these multiple levels of information at the same time. They claim that the ‘major problem for programmers is to coordinate fundamentally different problem spaces’.

2.5.2. Levels of abstraction in programming

Even the more constrained programming domain is highly complex and rich. Blackwell (2002) reports that programming is hard because of the lack of direct manipulation and the use of a notation to represent abstraction. In fact, the programmer may need to work at a number of different levels of abstraction, from considering the structure of the code of the system as a whole to the syntax of the command that is currently being typed in. These multiple levels may often need to be maintained and considered at the same time. It has been reported (Petre & Blackwell, 1999) that experienced programmers maintain very complex, ‘dynamically stoppable’ mental models of the programs they are working on in order to help them to manage this complexity.

Use of the phrase ‘levels of abstraction’ is itself somewhat problematic, in that it might be related to, for example, a single line of code, families of types of code or generalised views of programs. However, rather than characterise and classify the various different types of levels of abstraction that might exist, we used the phrase in this thesis to encompass both the problem and programming domain. In fact, Brooks (1983) takes a similar approach in describing programming as ‘constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain’.
2.5.3. The ‘driver’ and ‘navigator’ roles in pair programming

There are some suggestions that the roles of driver and navigator may be useful in managing the complexity of simultaneously considering the different levels of abstraction and moving between the problem and programming domains. While the driver seems to be performing a more conventional programming role (typing in code, manipulating the IDE etc.) the role of the navigator is more puzzling. Based on the pair programming literature we have derived two potential roles that the navigator might play:

The navigator as reviewer
The navigator as foreman

By ‘reviewer’ we mean that the navigator reviews the code that the driver is typing in, pointing out any syntax and spelling errors. By ‘foreman’ we mean that the navigator thinks about the overall structure of the code-base and whether the code is solving the business problem for which it is intended.

Note that where these are mentioned in the literature, there is no suggestion that they are mutually exclusive. In fact, they are more often cited as two of the main properties of the navigator role. In their book on pair programming, Williams and Kessler (2003, p.4) refer simultaneously to both of these when they state that “The navigator…observe(s) the work of the driver, looking for tactical and strategic defects” and Lui and Chan (2006, p.916) state that “the navigator/observer continuously observes the work with a view to identifying tactical defects and providing strategic planning. Tactical defects are syntax errors, typos, calling the wrong method, and so on. Strategic defects occur when…what is implemented just won’t accomplish what needs to be accomplished”. This implies that the navigator has to work at two different ends of the abstraction spectrum. We have assumed that this viewpoint has arisen from a mixture of the observation of pairs programming and first-hand experience of pair programming as a practice, however this is not explicitly stated.

There is other evidence to suggest the existence of the ‘reviewer’ role. Using a fictional pair programming session as an example, Wake (2002, p.73) suggests that one navigator behaviour is “The partner provides an ongoing quality boost: review(ing)”. Jensen (2003, p.2) also states that “The navigator review(ed), in real time, the information entered by the driver”. In cognitive terms, this continual ‘reviewer’ role suggests that the navigator catches spelling and
syntax mistakes so that the driver does not need to concern himself with reviewing. Essentially, the driver can ‘offload’ the reviewing activity onto the navigator.

There are also further occurrences of the ‘foreman’ role in the literature. Dick and Zarnett (2002, p.82) suggest that ‘the first is responsible for the typing of code (the driver); the second is responsible for strategizing and reviewing the problem currently being worked on (the navigator)’. In Williams and Kessler (2003, p102) Hayes, a professional programmer, suggests that ‘The tactical programmer is at the keyboard, worrying about the mechanics of getting the code into the machine…..the strategic programmer is thinking about whether the objects are cohesive and loosely coupled, thinking about whether the names are correct, and thinking more about the problem that is being solved.’ Beck (2000, p101) also says that ‘While one partner is busy typing, the other partner is thinking at a more strategic level’, later describing this further as (p.58) ‘One partner….is thinking about the best way to implement this method right here. The other partner is thinking more strategically’. In their discussion of Extreme Programming and reflection, Hazaan and Dubinsky (2003) concur that ‘the one with the keyboard and the mouse thinks about the best way to implement a specific task; the other partner thinks more strategically. As the two individuals in the pair think at different levels of abstraction, the same task is thought about at two different levels of abstraction at the same time’ (emphasis theirs). These two aspects of the navigator role will be further investigated in Chapter 6.

### 2.5.4. Verbal protocol analysis

Verbal protocol analysis is the name given to the production, gathering and analysis of talk from participants. It has been used historically in cognitive psychology, particularly the use of ‘think aloud’ protocols, where participants are asked to provide a verbal report of ‘what they are thinking’ or ‘what they are doing’. Ericsson and Simon (1980) proposed a model for the verbalisation processes of subjects instructed to think aloud, finding that verbalising information is a useful way of gaining insight into cognitive processes. They claimed that verbalising does not affect cognitive processes when participants are only required to verbalise information to which they would usually attend. However, where this is not the case, they suggest that verbalisation will add an extra step to cognitive processing, but rather than transforming it will merely slow it down, for example by requiring the participant to find understandable referents.
There is a history to the use of verbal protocol analysis for gaining insight into computer programming. A literature review on verbal protocols in software engineering is available (Hughes & Parkes, 2003), which also suggests that the analysis of verbalisation may be a useful method by which to study pair programmers. By applying this type of analysis to pair programming observational studies, one hopes to be capturing only verbalisations that would normally have been produced in a pair programming session, rather than eliciting new verbalisation especially for the study.

Of course it is possible that the production of verbalisations may be an important factor in differentiating pair from individual programming. For example, these verbalisations might have a role to play in the gains in software quality reported in the pair programming literature. In fact, there are a number of reports of verbalisation having a positive effect in other domains: Chi, de Leeuw, Chiu and Lavancher (1994) found that the production of self-explanations augmented the understanding of physics questions and Pirolli and Recker (1994) suggest that self-explanation encourages meta-cognition. Ainsworth and Loizou (2003) similarly found that self-explaining provides computational offload and helps to improve verbal declarative knowledge. This suggests that one of the reported positive effects of pairing might indeed be a function of the requirement on the programmer to talk about what he or she is doing.

In Chapters 3, 5 and 6 we have employed verbal protocol analysis in order to better understand the behaviour and cognition of pair programmers.

### 2.6. Key issues identified

While there are many reported benefits of pair programming, little is understood about how these benefits may be achieved. In order to begin to uncover some of the mechanisms at work in successful pair programming sessions, we have identified the following four key issues:

**Expertise**  As highlighted, much has been uncovered about what constitutes ‘expertise’, both generally and in software development. While the terms ‘novice’ and ‘expert’ are used frequently in the pair programming literature, there is a need for a clear definition of ‘pair programming expertise’. In Chapter 3 we begin to address this issue in order to focus on the behaviour of experienced pairs for the remainder of the thesis.
Tools and Setting  The literature discussed in Section 2.3 highlights the fact that if we are to better understand what happens in a successful pair programming session, we must first understand the environment, tools and setting in which such a session takes place. In Chapter 4 we draw on ethnographic techniques as a means of providing such insight.

Task  We have identified two ways in which a programming pair may work together: collaboration and co-operation, and outlined the relative costs and benefits of each. In Chapter 5 we investigate whether successful pair programming sessions are predominantly collaborative or co-operative in nature.

Role  In Section 2.5 we discussed two roles that prevail in the pair programming literature: ‘driver’ and ‘navigator’. In particular, we identified two commonly held beliefs regarding how these roles serve to provide the reported benefits of pairing. We named these the ‘reviewer’ and ‘foreman’ models. In Chapter 6 we analyse experienced pair programmers working together in search of evidence for the existence of these role models.

In Chapter 7 we define a set of pair programming behaviour derived from the studies in Chapters 3, 4, 5 and 6.
3. The role of experience

The notes I handle not better than many pianists. But the pauses between the notes – ah, that is where the art resides!
- Artur Schnabel

3.1. Introduction

As described in Section 2.2, the literature suggests that there are some special attributes and behaviours associated with experienced pair programmers. In this thesis we aim to explore this further, however in order to compare more and less experienced pair programmers we first need to derive a suitable method of delineating between ‘expert’ and ‘novice’ pairers. This chapter therefore comprises two main sections: in Section 3.2 we consider how to appropriately assess a participant’s level of pair programming expertise. For this we used an informal survey of 45 participants that was obtained across our four studies. In Section 3.3 we analyse and contrast the behaviours of a number of observed pairs in a single study according to this derived delineation. We then suggest a number of interesting differences in behaviour that not only highlight the differences between novice and expert pairs, but also confirm the usefulness of the assessment level derived.

3.2. Rating expertise

Evidence provided from the studies on rating discussed in Section 2.2 lead us to suggest an inverse relationship between level of experience and rating. However, as many of the skills associated with pair programming expertise are inter-personal and communication skills, the length of time a practitioner has been pair programming was also expected to play a key role. In an informal survey, 45 participants were asked to rate their own experience level and the
amount of *time* they had spent pair programming. Their supervisor or, where this was not possible, a peer was then asked to rate their level of pair programming *ability*.

In analysing the responses to these questionnaires, it should be noted that the sample used for this analysis was purely opportunistic in nature and limited to programmers at four companies who were willing to take part in the study. The comparison of self and peer/supervisor ratings should also be approached with caution, as the questionnaires used for self-rating asked about pair programming ‘experience’ while those used for peer/supervisor rating asked more generally for a ‘pairing rating’. While one might assume these to be the same thing it is possible that there is an underlying assumption that part of ‘experience’ would automatically involve their length of tenure.

In actual fact, of 29 participants rated as high ability by their peers only 6 rated themselves as highly experienced. However, of the five participants rated low ability by their peers, none rated themselves as highly experienced and only 2 rated themselves as having obtained even a medium level of experience. However, pair programmers with less than six months’ pair programming experience were consistently rated as low and that there was a general trend in likelihood to be more highly rated as the six month experience threshold was reached and surpassed.

The relatively high number of combinations of rating categories and low number of participants result in the data being unsuitable for statistical analysis with regard to significance. However, the themes and trends that emerge suggest a conformance to the literature on self efficacy and in particular show that those low in skill and experience tended to over-rate themselves when compared with cross-ratings by their peers/supervisors. A further analysis, considering programmers with pairing experience of less than or greater than/equal to six months’ and using an average of self and peer/supervisor ratings more closely follows expectations in the literature. That is, those considered by others to be lacking in experience tended to comparatively over-rate themselves and those considered highly able pair programmers by others tended to under-rate themselves. This suggests that a six-month threshold provides a clearer predictor of self and peer/supervisor rating of expertise than one year. Therefore, the next section in this chapter aims to assess more clearly whether this six month mark provides a useful delineation between ‘inexperienced’ and ‘experienced’ pair programmers.
3.3. Assessing behaviour

An initial pilot study with 14 participants took place in order to assess whether there were many differences in behaviour between more and less experienced pairs, when delineated by six-months commercial pair programming experience. Using this benchmark, half of the participants were considered as novice and half as experienced at pairing (i.e. 7 participants fell in each category). The study was exploratory, considered only a small sample size and did not rely on formal experimental design to interpret significance. During the study fourteen pair programming sessions of one hour each were observed in an internet banking company, which employs approximately 160 IT staff. As not all projects use pair programming the study had one project as its main focus and an additional two projects were considered. The main project in question was a mixture of experienced external XP consultants and the company’s own IT staff. Half of these sessions were recorded on video. Ten of the sessions were development activities, two involved fixing the test environment and two involved the creation of a flow-chart.

In the following sections we set out the method by which the utterances of these pairs were assessed and present results comparing the two groups. In the discussion section we then consider what these findings suggest with regard to the behaviours of more and less experienced pair programmers.

3.3.1. Method

An initial set of candidate codes for assessing pair programming behaviour (shown in the left-hand column of Table 3.1) were taken from a list of typical pair programming activities defined by Wake (2002). Following a pilot study of one pair in an academic environment, a number of category changes were made: first, the ‘asks for help’ category was modified to instead record the asking of questions as this was more easily delineated and did not require any attempt on the part of the coder to ascertain the implicit motivation behind the question (i.e. whether it was a request for help, or, for example, a test of knowledge). The second part of this initial category (‘and receives it’) proved difficult to separate from ‘help with detailed and strategic thinking’. During the pilot it became clear that these could occur in two ways: either agreeing with an initial explanation or suggesting (or counter-suggesting) an approach. The pre-pilot pair never disagreed without providing an alternative. The final change was the addition of a ‘rest’ category for use when the pair took time out from programming. This reflected observations in
the pre-pilot study and is in keeping with reports in other studies that pair programming is exhausting (e.g. Gittins, Hope, & Williams, 2001; Sharp & Robinson, 2003). The refined categories are shown in the right hand column of Table 3.1.

Table 3.1 Initial and refined candidate pair programming behaviours

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Refined code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A person asks for help (and receives it).</td>
<td>Question. Direct question, e.g. ‘What arguments does this function take?’ or indirect question, e.g. ‘Could you give me a hand with this?’</td>
</tr>
<tr>
<td>The partner helps with both strategic and detailed thinking.</td>
<td>Explain. Explanation of an approach or a piece of existing code.</td>
</tr>
<tr>
<td>The pair learns together; Cross training is built in.</td>
<td>Suggest/counter. Suggestion of a theory, approach or an alternative to an existing suggestion or explanation.</td>
</tr>
<tr>
<td>If one partner forgets something, the other can remind him or her.</td>
<td>Remind. Remind the partner to do something or consider something e.g. ‘Don’t forget to fill in that stub function’ or ‘don’t forget that we need to be able to see both addresses on the screen at once’.</td>
</tr>
<tr>
<td>There is a protocol for changing hands.</td>
<td>Change driver.</td>
</tr>
<tr>
<td>The partner provides permission to do the right thing, even if it is starting over.</td>
<td>Confirm/agree. Confirm or agree with a previous suggestion, course of action or explanation.</td>
</tr>
<tr>
<td>The partner provides and on-going quality boost: review, refactoring, unit testing, new ideas</td>
<td>Review/refactor.</td>
</tr>
<tr>
<td></td>
<td>Test</td>
</tr>
<tr>
<td></td>
<td>Rest</td>
</tr>
<tr>
<td></td>
<td>Other. Category for other behaviours not covered by the above.</td>
</tr>
</tbody>
</table>

The pilot study also showed that transactions took place much more quickly than originally anticipated (up to twelve interactions occurred in a single minute) and that the pattern of
interactions (that is, the order in which the behaviours occurred) seemed to be a potentially important and useful part of the data to be captured. A table of categories was then produced to allow fast data capture and for accurate sequencing information to be gathered. This is shown in Table 3.2, where each table row signifies a single minute. Sequence numbers were then hand-written into the boxes for each code in the order in which they occurred. Note that each utterance only received one code. While this might seem counter-intuitive (i.e. there might be an agreement about testing), in practice it was not problematic, as utterances were coded individually rather than analysed for the context in which they fell. That is, an agreement statement (e.g. “Yeah, I see what you mean”) was coded as an agreement even if it was within the context of testing (e.g. the previous or directly consecutive utterance was testing related).

Table 3.2 Table for capturing candidate pair programming behaviours

<table>
<thead>
<tr>
<th>Driver</th>
<th>Explain</th>
<th>Ask</th>
<th>Confirm/Agree</th>
<th>Review/Refactor</th>
<th>Test</th>
<th>Suggest/Counter</th>
<th>Remind</th>
<th>Rest</th>
<th>Look up</th>
<th>Ext Repr</th>
<th>Solo</th>
<th>Metaphor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To capture additional qualitative data, recording via video, still photography and handwritten notes were used in order to form a rich picture of the pair programming sessions. This additional information was used to provide further information regarding the socio-cognitive aspects, external representation use and any extraordinary events and behaviours. Such findings, across a number of studies, are discussed in Chapter 4.
3.3.2. Results

Figure 3.1 considers the number of verbal interactions of the types shown across the different combinations of expertise within a pair. HH denotes a pair in which both partners were high expertise, HL mixed expertise and LL low-expertise. It shows that more expert pairs averaged 27% fewer interactions per hour than novice pairs, and mixed experience pairs produced a number of utterances of these types between the other two, tending slightly more towards the high-high condition than the low-low.

Figure 3.1 Interaction frequency per one hour session according to pair programming expertise

In accordance with Figure 3.1, Figure 3.2 shows that pairs where both partners had less than 6 months commercial pair programming experience generally produced more utterances of most types than more experienced pairs. In particular, more questions, explanations and suggestions were evident, along with a higher number of confirmations. However, somewhat more surprisingly, less experienced pairs utterances suggest that they reminded each other less, rested less and reviewed each other’s work less. They also looked up information less than their more experienced counterparts, but made more used of external representations. Less experienced pairs also discussed testing less often than those with more experience.
We then consider differences in role behaviour according to pair type. Here we show example sessions for a low-low (Figure 3.3) and high-high (Figure 3.4) pair in terms of the types of utterances produced when one partner or the other was driving in each pair.

Figure 3.2 Interaction type according to pair programming expertise

Figure 3.3 Example session showing interactions when each programmer is driving in a low expertise pair
Figure 3.4 Example session showing interactions when each programmer is driving in a high expertise pair

As can be observed in Figure 3.4, the pattern of frequency of utterances of different interaction types appears much more stable in the more experienced pair, independently of who is driving. In contrast, Figure 3.3 shows a difference in the frequency of different types of interaction according to who is driving in a low-expertise pair.

3.3.3. Discussion

The findings in Section 3.3.2 above suggest a number of interesting differences between pairs consisting of more and less experienced pair programmers. First, expert pairs were 27% less verbose. That is, they produced nearly 100 fewer verbal interactions per hour than novice pairs. This suggests that expert pair programmers might be more selective about their interactions and may have a better understanding of the role and knowledge of themselves and their programming partners. This lends support to the claim by Williams and Kessler (2003, p98) that “experts don’t have to spend much time explaining things to one another….they can often communicate using few words”. In particular, Figure 3.2 clearly shows that novice pairs suggested and counter-suggested much more frequently (82.5 times per session) than experts (42.5 times per session) which accords with suggestions in the observers field notes that, rather than spend time resolving differences of opinion, novice pairs often thrashed between different strategies depending on who was driving. Similar behaviour has been reported in the use of external representations (Cox, 1999, pp., p.351), where subjects “abandon a poorly
constructed or erroneous model and try something different”. Novices also appeared to practice ‘lingo’ and ‘explained’ to one another much more.

Initial findings suggest an interesting relationship between expertise, role and behaviour. Where more experienced pair programmers work together there seems to be a defined set of behaviours that remain common whoever is driving. An example experienced pair programming session is shown in Figure 3.4 where the lines represent the overall interaction occurrences in the pair programming session according to who was driving. This might imply that when the pair change role they also change behaviour accordingly or alternatively that there is no change in behaviour irrelevant of whether a partner is driving or navigating. By contrast, less experienced pair programmers seemed to behave quite differently from one another, even when filling the same role. That is, there was no defined driver or observer behaviour. Perhaps surprisingly, this did not however mean that they maintained the same behaviours when driving as when observing. Often a novice pairer would not only behave differently when driving than when observing, but also behave quite differently to the way his partner behaved when he was driving. See Figure 3.3 for an example of a low-experience pair programming session. Less experienced pairs also changed role nearly twice as much as those with more experience (a novice pair averaged 7.5 role changes per hours compared to an experience pair averaging 4 changes per hour).

This suggests not only that six months’ experience is a useful level by which to separate pair programmers in order to contrast their behaviour, but also that there is indeed something ‘special’ about the manner in which an experience pair are able to fluidly change role with little effect on the ‘shape’ of the overall pair programming session.

3.3.4. Study Limitations

These studies have a number of limitations. In particular, the low number of participants in the pilot study and the inclusion in this study of few projects (one main project and two subsidiary projects) at a single company suggest that the data presented may not generalise over a larger population. In addition, the mixture of qualitative field notes and more quantitative analysis is rather unorthodox. Finally, given that the observer both defined the categories used for the analysis of what the pairs talked about and also coded the observed pairs for these behaviours there is a possibility that there was a bias towards fitting observed behaviours into these categories even when inappropriate. However, the differences in behaviour identified and the
confirmation of the data in the field notes of the observer suggest that the delineation of six months’ experience is useful in highlighting some interesting differences in behaviour between more and less experienced pair programmers.

3.4. Conclusion

There are two manners in which this preliminary work informs the rest of this thesis. First, the delineation of pair programmers with more and less than six months’ commercial pair programming experience would seem to be a useful one in a number of ways: it provides a predictor of self and peer/supervisor rating of expertise and it provides a useful manner by which to select those pair programmers consistently rated as being experienced and able pair programmers. Second, the data comparing the behaviours of these groups indicates that there is something ‘special’ about the behaviour of experienced pair programmers when working together. In particular, further analysis of this data highlights the surprising result that the experienced pair seems able to alternate between driver and navigator role in a fluid and seamless manner. In fact, a change in driver seems to have very little effect on the overall ‘shape’ of the pair programming session.

Much of the work that follows in this thesis attempts to more clearly ‘unpick’ the manner in which these experienced pairs of programmers work. In particular focusing on their use of settings and the environment and the role and importance of being the ‘driver’ or navigator’. The data presented in this chapter not only suggests this as a useful area of study, but also implies that it may be an interesting and fruitful experience for novice pair programmers to observe and reflect on the dynamics of a more experienced pair. By doing so they might more fully understand the driver and navigator roles and the dynamic and fluid nature in which they might be exchanged.
4. The role of tools and setting

It is not the same to talk of bulls as to be in the bullring  
- Spanish proverb

4.1. Introduction

Having considered the role of experience and uncovered some of the differences in behaviour between more and less accomplished pair programmers, we will now focus on experienced pairs. We have defined experienced pairers as those having at least six months’ commercial pair programming experience (as discussed in Chapter 3). In particular, before focusing internally on the pair themselves, it is first necessary to consider their surroundings and the environment in which they work. As discussed in 2.3, we employ ethnographic techniques and consider how experienced pair programming is practically accomplished in a commercial environment. We focus on the tools, artefacts and environment in which the pair are embedded, particularly their impact on role and knowledge co-ordination. The goal of this analysis is to understand how the project layout and the artefacts and representations in evidence are used to facilitate various pair programming activities. In particular, we focus on role management and both within pair and extra pair communication. We also consider the extent to which the pair programming environments studied allowed each programmer to personalise the workspace for their comfort and individual requirements.

4.2. Method

The methodology used for this work was ethnographically informed. The data were based on observational studies whose field notes were supplemented with informal interviews, photographic and video evidence of artefact use and the verbal protocol analysis of transcribed sessions. All of the sessions were recorded in digital audio and three captured on video. These recordings were transcribed and combined with the field notes, informal interviews and
photographs to create a rich picture of the interactions from each session. Focus was on the social environment of people and artefacts in which the pair programming was embedded. The method used was inspired by the work of Grinter (2001) and based on Grounded theory (Glaser & Strauss, 1967). Grounded theory helps to ensure a solid foundation for hypotheses by basing them on observational studies in the real world. Here, theories came directly from the observational data. In particular, any emergent themes were considered at the end of each day of study and were then focused on in future sessions in order to gather further evidence of their generality. Disconfirming evidence was also sought. The methodology has also been greatly influenced by the work of Hutchins (1995) in regards to the study of artefacts, however the approach was not truly one of ‘distributed cognition’ as the pairs were opportunistically studied for an hour each and therefore the origin and transformation of information and artefacts were often not in evidence. Instances reported in this chapter relate to themes consistently seen in the data unless otherwise specified. In addition, informal interviews were used to clarify emergent themes and verify that the observer had derived an accurate understanding of what had taken place.

The author’s background as an experienced commercial software developer was particularly beneficial in facilitating acceptance in the field. For example, taking coffee breaks and socially interacting with the participants by exchanging programming ‘war stories’ may have contributed to the fact that only two pairs were not happy to be observed. Similarly, the pairs that were observed did not seem at all restrained in their use of colloquialisms, programming jargon or social chat.

Only sessions with programmers of at least six months’ commercial pair programming experience are considered in this report. As we have shown in Chapter 3, there is evidence to suggest that this provides a good indication of an ‘experienced’ pair programmer. Where possible, examples of actual occurrences are given.

4.3. Background

4.3.1. Sessions observed

The profiles of the sessions included in this chapter are detailed in Table 4.1.
Table 4.1 Profiles of sessions observed

<table>
<thead>
<tr>
<th>Business sector</th>
<th>Number of projects considered</th>
<th>Number of sessions considered</th>
<th>Agile/XP approach?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project A Banking</td>
<td>1</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>Project B Entertainment</td>
<td>2</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>Project C Mobile communications</td>
<td>2</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>Project D Banking</td>
<td>4</td>
<td>8</td>
<td>Yes</td>
</tr>
</tbody>
</table>

A description of each of the projects observed is provided in Section 4.4.

4.3.2. A typical pair programming session

This section describes a fictional but typical pair programming session in order to provide a sense of background and context for the more detailed, within session, descriptions that follow.

The stand-up The day begins with a stand-up meeting. Each pair gives an overview of what they worked on yesterday and any issues they encountered. Areas where one activity might impact on another are identified. The pairs in the team consider the outstanding activities that are written on visible task or story cards (see Section 1.2), and decide which to work on next. A task will usually take about one full ‘ideal’ programming day to complete. In some cases this will mean continuing to work as a pair on a task not yet completed, but in other cases there may be some negotiation. Here, John and Mary continue working on yesterday’s unfinished task.

The coding Once the meeting is finished, they agree to work at John’s desk. As the team all pair program, the desks are set out with room for two chairs to fit side by side in front of the large screen. They spend some time discussing progress and decide that now that they have completed writing the automated test script that will prove their code works once it is done, they can get on with writing the code itself. Mary remembers that there was an outstanding issue and they have a discussion with the allocated business ‘customer’ in order to clarify the requirement. Once resolved, John pushes the keyboard over to Mary and suggests “you drive”. Mary starts up the Integrated Development Environment (IDE) that the team use and it opens up two initial views. One view shows the suite of automated tests, including the one that they
wrote yesterday. The second view shows the system source code, organized into classes and their methods. It is here that the new code will be written.

As they start working, they discuss the approach they are going to take on each sub-task together before continuing. Often they draw informal sketches, type a piece of example code or point at something on the screen. They switch seamlessly between views and often transfer the keyboard and mouse between them, sometimes with utterances like ‘show me what you mean’ and sometimes simply indicating their intention to change roles with a gesture. Occasionally, whoever is navigating picks out a typing mistake or syntax error.

At one stage Peter, who is working nearby, overhears the pair discussing an issue that they are having problems solving. Peter knows about this area and they have a three-way discussion. Occasionally one of the pair overhears their name being mentioned by another pair and gets involved in a different issue. When there are short breaks in the development, perhaps while the test suite is running or completed code is being integrated, they take the opportunity to have a break, a social chat or check their email. Regular coffee breaks are also taken every hour or so.

**The Integration** Once the code is complete and the test suite runs successfully, Mary picks up a fluffy toy from on top of the integration machine and places it on top of their terminal to show that they are integrating their code with the most recent version of the whole system. This signals to the rest of the team that they should not be making changes to the integration system at the same time. Once the code is copied across, a full set of integration tests are successfully run, and they place a green sticker on their paper task card and stick it back on the progress chart.

### 4.4. Results

Here we present the results of our analysis of the observational studies. In particular, we focus on those elements that seem to be connected to the support of pair programming in practice. First, we consider the physical environments and pair programming uptake on each of the projects. We then go on to consider the issues of peripheral awareness and the use of various artefacts across all of the projects observed.
4.4.1. Project A

**Physical environment** Project A’s accommodation was almost entirely open plan, apart from a number of meeting rooms and project spaces. Each desk had room for two developers to sit side by side. In addition, each desk had two large screens. Typically these displayed the code and the test system. Some developers had a third screen for internet access (although the internet was accessible from either of the other screens).

The team had a dedicated ‘project space’. This screened off room had story cards on the walls that described the project tasks. There was a large projector in the project space for use in the daily stand-up meetings or when planning was taking place. The project space also had several beanbags to sit on, a small desk whose height could be varied and one or two bar stools. It was used informally by the project for ‘show and tell’ and other ad-hoc discussions.

**Pair programming uptake** The programmers on project A all used pair programming practically all of the time. They introduced a concept they called the ‘pair stare’, which is a matrix showing who has paired with who. This matrix was used to ensure everyone paired with everyone else and was also considered to help identify certain pairing problems. The programmers themselves were given the freedom to decide when not to pair but to peer review instead. However, in practice this rarely occurred.

4.4.2. Project B

**Physical Environment** Project B also worked in a open plan office. The desks were laid out as shown in Figure 4.1 (squares are desks and circles are chairs). Unlike project A, the desks were not set up specifically for pairing. A depiction of the general system architecture was permanently displayed in an informal drawing on a whiteboard on the wall in the project team’s space, from where it could be easily referred to when required. Story cards showing the project tasks were hung in a plastic wallet on the wall.
**Pair programming uptake** Programmers on Project B did a mixture of pair and solo work due to logistics, such as what time people arrived and left, attending to other work that needed to be done or urgent requests that had to be dealt with. Almost everyone paired from 11.00-13.00 and 14.00-16.00. Some pair programming sessions were specifically designated as ‘knowledge transfer’ sessions. At these times, the programmer who was more experienced in that area tended to drive, and predominantly commentated or explained what they were doing to the navigator.

### 4.4.3. Project C

**Physical environment** Project C was also open plan but with screens between desks, over which people’s heads could be seen. While each programmer had their own ‘private’ desk area, where pairing sometimes took place, there was a special dedicated area for pairing, where interruptions from email or telephone calls could be avoided. A number of screens displayed details of the various software releases planned. There was also a set of whiteboards at one end of the room on which the story cards for each iteration were displayed. There were additional whiteboards along the ‘corridor’ that contained information like system architecture diagrams or refactoring ‘smells’. Figure 4.2 shows the ‘usual’ layout for the room (desks are squares and chairs are circles), however sometimes the pairing desks were used alone.
Figure 4.2 Physical layout for Project C

**Pair Programming uptake**  Project C actually consisted of two ‘sub-projects’. Pair programming had been introduced eleven months ago, as part of a move to an ‘agile’ approach. It was considered a big culture change and several of the programmers reported that they found pairing difficult. While pair programming was optional for both projects, on one project everyone paired most of the time, while on the other the programmers tended to pair only on ‘tricky problems’.

#### 4.4.4.  **Project D**

**Physical environment**  Initially the project was a single team, however this got too large to manage and communication began to diminish and so the team was split into two. Both teams worked in an open plan environment, with several glass-fronted meeting rooms, one of which was available as a ‘drop in’ meeting room. There was also an informal meeting corner, where information displayed included: story cards showing the project’s tasks; a board showing who was in which small team; a board showing the support rota and the iteration plan. There was no system architecture on display (although one was available on the intranet). The physical layout for team 1 is depicted in Figure 4.3 and for team 2 in Figure 4.4. Developers had their own desks and ‘moved in’ with each other to pair.
Pair Programming uptake  On project D, while pairing was at the programmer’s discretion, most people chose to pair most of the time. Pairing did not start until around 9.30, before which developers may start on or continue a task on their own. Two of the most experienced software developers paired rarely (for approximately one hour during the week), and only when a particularly difficult problem presented itself.

4.4.5.  General themes in layout and uptake

Physical layout  The first of the core practices in the latest version of XP (Beck & Andres, 2004), recommends that teams should sit together in an open plan space. This mirrors research done within the CSCW community on ‘war rooms’, for example Teasley, Covi, Krishnan and Olson (2000) found that productivity was greatly increased when developers were collocated (producing twice as many function points and cutting two thirds off the time to market). All of the projects studied followed this recommendation and worked in open plan areas. The manner in which pairing took place followed two distinct models: the ‘cohabitor’ model and the ‘relocator’ model. In the cohabitor model, one pair programmer would ‘move in’ with the other, pulling an additional chair up to their desk. In the relocator model both
programmers would move to one of a number of desks in an area specially designated for pair programming. The cohabitor model was most often in evidence, presumably for two reasons: office space is costly and the relocator space model by definition requires redundant desk space which would often not be available; the cohabitor model ensures that the pair have at the very least one of the programmer’s tools at their disposal (e.g. manuals, pens, notepads, post-it notes). However it is also possible that the cohabitor model might not be ideal, in particular because the pair will still be open to interruptions from telephone calls etc and also because the balance of power within the pair may be affected by the fact that they are working at a particular programmer’s desk.

Pair Programming Uptake

While pair programming was optional on three of the four projects observed, almost all of the programmers observed chose to pair, at least for core hours. However, few programmers were able to sustain pairing all day every day. Rather, the programming day consisted of a mixture of non-pairing activities (e.g. checking email, going to meetings, making telephone calls), core pair programming hours and regular breaks. This allowed an individual programmer to attend to activities that may not have been related to the programming activity at hand, but still needed to be completed. In addition, it compensated for the fact that pair programming is intensive and tiring.

4.4.6. Peripheral awareness

All of the teams studied worked in open-plan environments. Traditionally, the work of an individual programming at a PC has little visibility to the rest of the team. However, where a pair is actively discussing the work they are doing in an open-plan environment the project team are able to ‘overhear’ each other and pick up on useful or relevant information. This phenomenon is similar to that reported between journalists (Heath, Sanchez Svensson, Hindmarch, Luff, & Vom Lehn, 2002) and is also mentioned by Chong and Siino (2006), who found that on pair-programming teams programmers scanned the room to find the best time to interrupt their team members. This peripheral awareness is further facilitated by the fact that pair programming demands a high level of verbal communication and therefore renders transparent much information which might be hidden in a more traditional software development environment. In fact, through verbal protocol analysis of one of the four studies included in this thesis, pair programmers were shown to produce more than 250 verbal interactions per pair programming hour (Bryant, 2004a).
‘Overhearing’ a pair’s verbalizations not only allowed a third party to tune in to relevant conversations from surrounding pairs (see Figure 4.5), but also allowed a developer to highlight information that might be relevant to others (see Figure 4.6). Figure 4.6 and Figure 4.5 below provide anonymous examples from different pair programming sessions where Zoe is used as the name of the project member who is external and Andrew and Betty are the names used for members of the pair.

Andrew: Reporting requirements…oh yeah.
Betty: Whenever he’s free we’re…
Zoe (overhearing): He’s free now.
Andrew: Is he?!

Figure 4.5 Example of proximity facilitating peripheral awareness through over-hearing

Andrew: Because it’ll fail won’t it?
Betty: Yeah…that was in…(sighs)...package one wasn’t it? And it’s not here, so it needs to go into package two I think.
Andrew: OK, so that’s something we can make (raises voice) Zoe aware of.
Zoe: What’s that?
Andrew: Ummm…something which was, I think in (package name), which has just been abolished.
Zoe: Right, yeah. It’s going to be constantly evolving unfortunately, isn’t it?

Figure 4.6 Example of proximity facilitating peripheral awareness through name-dropping

On occasions overhearing triggered episodes where a third party joined the pair. In some cases, where the problem required specialist knowledge that the pair did not have, a pair change was negotiated. This allowed the developer who had overheard to become part of the pair working on that problem, while an existing member of the pair ‘swapped out’. This fluid re-pairing is contrary to the static, formal nature of pair allocation typically described in the pair programming literature.

4.4.7. Artefacts

This section identifies a number of artefacts, designed for and usually used by individuals, which were re-appropriated or augmented for collaborative use and played an intrinsic role in the pair programming sessions observed. In particular, these artefacts assisted in the dynamic negotiation of driver and navigator roles; assisted within-pair communication; rendered work visible and helped assure that the programming pair were maintaining a common mental model of the task at hand.
Keyboard
The keyboard, designed as a solo data input device, consistently became the primary token for ‘floor control’ - possession of the keyboard signalled who was in the ‘driver’ role and who was ‘navigating’. This is an example of constraints being built into the tools (Hutchins, 1995), as complications from having both programmers simultaneously editing code are avoided. The keyboard was often used to indicate intention of role change: the driver might slide the keyboard over to the navigator to suggest an exchange of roles, sometimes with an accompanying utterance (see Figure 4.7 for an example). Interestingly, although relinquishing control of the keyboard in this way seemed acceptable, initiating control of the keyboard was rare. That is, the keyboard was often ‘offered and accepted’ but very rarely ‘taken without offering’.

Andrew:    If you…go to…
Betty:     (sliding the keyboard over to him) (You) drive…it’s easier.

Figure 4.7  An example of dialogue during keyboard hand-over.

As well as being used for both its traditional role and as a token for ‘floor control’, the keyboard also assisted intra-pair verbal communication. One of the methods by which the object of conversation might be highlighted is by use of the keyboard’s cursor keys. This seemed to take place for a number of reasons including: avoiding the overhead for the driver of switching to another medium; overcoming difficulties with mouse control/dexterity; ensuring accuracy of communication and allowing multi-modal pointing (one partner could highlight with the keyboard while the other used her finger).

Mouse
Despite also being designed as a solo data-entry device, the mouse was used as a collaborative resource. Control of the mouse was less formal than the keyboard and while in the majority of cases the driver would control the mouse and the keyboard, in three of the sessions this was not at all the case. It was not uncommon across sessions for the navigator to lean over and use the mouse to ‘point’ at something on the screen, rather than pointing with their finger or describing the target of interest verbally (see Figure 4.8 for an example). Presumably this was to avoid both the physical inconvenience of finger-pointing and the time and cognitive load associated with verbally describing.
Andrew: …just test it… and that means you don’t have to start faffing about with this… (uses mouse to point at screen)

Betty: Yeah… I know.

Figure 4.8 An example of navigator use of the mouse for pointing
(Betty is driving and Andrew is navigating)

In two cases, a wireless mouse was placed on the desk between the two programmers and used as a communal resource to point at and highlight code during discussions and to position the cursor. This was possible because the pair were close enough to easily reach the mouse with the appropriate hand. Interestingly, neither pair had any difficulty coordinating mouse or cursor control although this was never discussed or mentioned during observations.

Interactive Development Environment and the Code
The code itself played an important role in communication and did not seem to be merely the driver’s ‘translation’ of the collaborative effort. For example, on occasions a period of silence did not indicate the end of an interaction. Sometimes verbal communication between the two programmers would trail off and the interaction would be continued by the driver typing at the keyboard. This was clearly the case where the navigator interjected using agreement protocols normally reserved for conversations (e.g. Uttering ‘mmmm’ or ‘yes’ or ‘uh huh’). Examples of this type of interaction are shown in Figure 4.9.

Andrew: A slightly different side. That’s got a… (types code)
Betty: Uh huh
(later)
Betty: Yeah, I think so. Yeah it makes it easier to write accessor methods as well, I think, if you do… (types code)
Andrew: Yeah
Betty: OK, so that’s cool

Figure 4.9 Examples of the code being used to continue conversation

As previously mentioned, the target piece of code being referred to would often be identified via pointing. In such cases, the distributed cognition afforded by this representation often led to underspecified statements, as reported elsewhere (Flor & Hutchins, 1991). An example of
this is given in Figure 4.10, where ‘this’ and ‘that’ are used to refer to parts of the code being pointed at in a variety of manners (emphasis added).

Andrew: Err…get this version of that….so that’s got that….so it’s come through there now.
Betty: So if you try and run that through there now.
Andrew: Is this a problem?
Betty: That should be included in the project.
Andrew: Yeah

Figure 4.10 An example of the code being used to supplement verbalisation

The Interactive Development Environment (IDE) that was being used facilitated this form of interaction by providing a readily visible and comprehensible representation of the program for both parties. The physical layout of the screen and the programming pair ensured that this representation could be easily read by both partners and referred to by gesturing either using the mouse or keyboard, or by physically pointing at the screen. On occasions the IDE actually initiated a ‘conversation’. This was particularly evident when, for example, the programmers’ attention was drawn to an error that had been introduced by a ‘red light’ appearing next to a particular automated test. This representation would trigger a conversation between the programmers and often initiate a new episode of problem solving.

Informal External Representations Some kind of informal, paper-based representation was produced or used during nearly every session observed. These were either informal sketches or lists. Sketches were widely used. They featured in 20 of the 36 sessions observed. These sketches were highly informal (e.g. Figure 4.11) and in some cases near illegible. This was considered preferable to using more formal or communal diagrams. For example, one programmer suggested “if it’s pre-drawn you feel like there’s nothing you can contribute”, and another that “it feels more comfortable than an official document”.

While the representations appeared useful in facilitating communication, the extent to which the non-sketching partner engaged differed widely. In fact, it was often hard to distinguish whether the representation was being produced for individual or for pair use. For example, on occasions these representations seemed to be produced to clarify the thoughts of the programmer doing the sketching, and in one particular case, the ‘sketch’ was merely traced on the table with a finger. In an informal interview, one programmer referred to these diagrams as
“like a brain-dump” and another stated “If I scribble it down I can find out if I’m thinking absolute rubbish”. This implies that such sketches may at best be playing a highly ephemeral role in communication with the partner, or be used as part of the pragmatics of the interaction (for emphasis, say), or may simply be acting as a cognitive aid for one member of the pair. If this is the intention, then their role may be simply to lower the load on working memory and assist in discovering inferences as documented in Suwa and Tversky (2002) or to attempt to externally work with very rich, multi-dimensional models (Petre & Blackwell, 1999).

Where both parties appeared to engage with the representation, its role seemed to be to highlight structure or logic regarding how things related to each other. In one session a timeline was drawn to show the relationship between three conceptual dates and in another a diagram was produced to show how one code method called a number of other sub-routines. Interview data suggests that these were used to assist communication: “It helps communication better than just talking”, “Some things are hard to articulate…so it gives you a common diagrammatic language”. See Figure 4.12 for an example of a verbal exchange about creating an informal representation. However, the usefulness of diagrams was considered limited, with comments such as “Between a pair it’s easier to just whack out a piece of code” or “You work in small mini-steps, so you can keep it all in your head”.

![Figure 4.11 An example informal external representation](image)

Andrew: Oh god (laughs)...Shall we draw the hierarchy?
Betty: Mmn.
Andrew: Because it’s...it’s more than just one.
Betty: It’s loads isn’t it.

![Figure 4.12 An example exchange about creating an external representation](image)

**Shared architecture** Each of the projects observed had some communal representations posted up either in the physical project space (in three of the four companies) or on the intranet
(in one of them). The role of these representations seemed to be to allow the wider implications of a pair’s work to be visible, to provide a means of facilitating communication across pairs and to ensure an understanding of the system as a whole.

**Lists** Lists were mainly produced as an aide memoire. They were produced in mutual view and both partners often contributed, either at the time the list was produced, or when additional items required adding later. Usually these lists were created in a programmer’s personal ‘day book’ (a kind of diary for each day) or on a separate sheet of paper. In one case items were noted on post-it notes and stuck to the screen. This fits findings by Adelson and Soloway (1988), indicating that experienced software engineers tend to work in a roughly hierarchical manner, taking notes if something comes to their attention which is not at the current level of detail.

Figure 4.13 shows an example of the type of list produced. As is obvious from the degree of informality, these lists do not seem to be created for use by anyone other than the pair who produce them. Informal interview data shows that they represent more of a checklist, for personal assurance that all the necessary sub-tasks are complete before a piece of work is deemed finished. However, in one case a programmer claimed that they would be useful for another pair who might later work on the same or a similar task. Interestingly, while a number of these lists were produced, they were rarely referred back to and ‘checked off’ in the sessions that were observed, and never seen to be transferred from one pair to another. This implies that their value might lie more in their creation than in their persistence. Perhaps their very existence was enough of a ‘memory jog’ without a need to refer to them.
**Surrogate mouse** In one session a small ball of paperclips was used as a very informal role control mechanism. Assume the programmer using the paperclips is called Benny and his pairing partner is Alex. When Alex was the driver, Benny (currently the navigator) would take up the paperclips and make movements and finger-twitches similar to those that the driver was making with the actual mouse. When B wished to assume the role of driver he would let go of the paperclips as a signal to Alex, who would then relinquish control of the mouse (and keyboard). Once finished as the driver, Benny then let go of the mouse and once more picked up the paperclips, at which point Alex almost immediately took up the driver role (and the mouse) once more. Use of the surrogate mouse can be seen in Figure 4.14.

![Figure 4.14 The surrogate mouse](image)

**Toys** Tokens were used as an informal ‘locking mechanism’ for integration. In fact they were so informal that their effectiveness relied entirely on members of the project understanding and conforming to the rules of their use. This is particularly interesting as some other, more formal, technology based locking mechanism might just as easily have been put in place. It is also contrary to an example in Rogers and Ellis (1994), showing that software developers were inconsistent in their use of a manual whiteboard system for file locking as this was extraneous to the work activities they were involved in. This choice of informal token could be viewed as relating to the team’s identity and sense of humour, acting as a cultural as well as a technical token.

In keeping with a number of studies in the field of CSCW (e.g. Heath, Sanchez Svensson, Hindmarsh, Luff, & Vom Lehn, 2002; Robertson, 2002), the physical presence of the toy and the manner by which it is manipulated may play an important role in alerting others to peripheral events which might be of interest (in this case use of the integration machine). This
is consistent with studies of news rooms, police operations, traffic control centres and operating theatres, studied by Heath, Sanchez Svensson et al. (2002) in which participants were seen to “design and produce actions to render features of their conduct selectively available to others” (p.318) and to “encourage another..without interrupting what they are doing, to see, detect and notice something..which may have to be dealt with” (p.342). Robertson (2002) stresses the human ability of peripheral awareness as particularly pertinent to this phenomenon. In the pair programming teams, each team member is given the opportunity to notice the change in integration machine control by the action of the developer walking over and retrieving the toy. Even if this is not attained, the toy’s placement on top of the developer’s monitor makes it continually available to the rest of the team and further assists in the maintenance of a common model of the system’s current state.

4.5. Discussion

These studies show pair programmers interacting seamlessly within a rich environment, using artefacts and verbalizations to assist in collaboration within the pair and to provide transparency outside the pair. Most interestingly, a number of the tools used to assist with collaboration were initially created for individual use and have been re-appropriated to embody additional constraints or skills which are now required. The following two sections consider these additional requirements in terms of role management and communication both within and outside the programming pair.

4.5.1. Role Management

Whereas the XP literature suggests that the roles of ‘driver’ and ‘navigator’ have specific properties, some of which will be investigated in chapter 6, little has been written about the management of these roles. The observational studies discussed have shown that these roles appear less formal in nature than is suggested. They also indicate that the roles are managed in a number of subtle ways, and practically realised via the interplay of verbalizations and the re-appropriation of traditional software development tools. The keyboard in particular has an important role to play in managing the relationship between driver and navigator. While it might be considered preferable to provide a separate keyboard for each programmer, in fact the use of a single shared keyboard facilitates role management by enforcing a method of floor control and providing an informal means of negotiating role change-over. It also becomes a common reference, embodying a set of social rules for changing role. For example, one can now relinquish the role of driver but not take it without being offered simply by following a
known social protocol for physical items - one is not accustomed to ‘snatching’ an item that is being used by someone else. More subtly, alternative tokens like the ‘surrogate mouse’ might be used to facilitate role change by implying a request to drive in a socially acceptable manner.

4.5.2. Within Pair Communication

Software development is a taxing task. One might consider that the overhead of communicating at the same time as producing software would be cognitively exhausting. The studies considered show that a mixture of verbalizations and artefact use work together to lessen this cognitive load, and in fact, produce tools that not only assist pair communication, but may also help the individual programmer.

The manner in which the programming pair combine verbalizations and gestures with the use of mouse, keyboard, code, IDE, external representations and other tokens and seamlessly weave these many artefacts together as a means of communication is impressive. In addition to using this rich array of ‘props’, the pair programmers observed had an implicit understanding of the role and appropriateness of each item and its use as a method of communication. Only on one occasion, where a partner was less able to physically manipulate the mouse, was this management of resources explicitly discussed.

4.5.3. Extra-Pair Communication

One of the additional benefits of the verbalizations required to successfully program in a pair is the transparency this lends to the work the pair is engaged in. Traditionally, the work of an individual programmer has little visibility to the rest of the team. However, where a pair is actively discussing their work in an open-plan environment they can easily be overheard by others. This provides opportunities to easily identify potential dependencies, conflicts or areas where assistance might be provided. Where additional attention needs to be drawn to an issue, a pair may raise their voices, or ‘name drop’ the person whose attention they wish to gain. On occasions a more formal mechanism for gaining attention is required. For example, when integrating new code onto the existing code base, three out of the four projects observed used a soft toy to indicate control of the integration machine.

4.5.4. Study limitations

The analysis provided here of course has its limitations. First, the observer participating in the study is an experienced, female computer scientist. Therefore, despite attempting to ‘view
everything as strange’, she may have made certain assumptions about the phenomena observed, perhaps assuming a commonality of understanding when there was none. The author is also aware that her own experience may have lead to a different focus than, for example, a social anthropologist may have had (similar issues are reported in Sharp, Robinson, & Woodman, 2000). In addition, although disruptions were kept to a minimum, the developers were being recorded in order to further analyse their interactions, therefore one should consider the potential impact on their behaviour.

Of the pair programmers considered here, only one was female, which reflects the gender bias in software development. It is possible that being observed by a female may have had some effect on the nature and dynamics of the interactions between the two programmers. However as the focus of this study was communication and role, it is unlikely that this would have a great deal of impact.

Finally, the studies reported used an opportunistic sampling method. That is, rather than ensure a typical demographic of the projects observed with regard to the overall pair programming community, the studies were merely of companies, projects and individuals who were happy to be studied at that time. As such, the analysis, discussion and conclusions raised here should be taken as illustrative examples of the type of tools and settings in which pair programming takes place, rather than as a statistical ‘norm’ of any kind.

4.6. Conclusion

The analysis in this chapter uses ethnography to begin to describe the rich ecology within which pair programming takes place in the four organisations observed. It particularly focuses on the difficulties faced by programming pairs with regard to role management and the communication of technical information. It highlights some of the ways in which pair programmers facilitate collaboration by re-appropriating or augmenting existing ‘solo’ tools or by using everyday artefacts in novel ways. It also goes some way towards dispelling the myth that all software developers are essentially solitary creatures lacking in social skills. It highlights some of the rich and subtle ways in which they communicate and shows that the verbalisations produced during pair programming can make activities more transparent and accentuate the benefits of the ‘war-room’ type environment. In addition, it comments upon the manner in which pair programming may provide a unique opportunity to facilitate cognitive offload on the part of the programmer.
Table 4.2 below is a summary of the artefacts in question, and the activities they appeared to facilitate.

Table 4.2  Artefacts and the pair programming activities they facilitate

<table>
<thead>
<tr>
<th></th>
<th>Role management</th>
<th>Within pair communication</th>
<th>Extra-pair communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verbalisation</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Keyboard</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Mouse</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tokens</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IDE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Further work in this area might consider the role of overhearing verbalisations from other pairs in more depth, perhaps providing a taxonomy for these episodes.

The re-appropriation and augmentation of tools initially designed for individual use suggests that programming pairs have some very specific extra requirements from their workstations and environments. While this ‘re-purposing’ shows ingenuity and flexibility on the part of the programmers, it suggests that there is scope for the design of more specialised tools for use when pair programming in a collocated manner. To the authors’ knowledge this has so far only been considered in distributed pair programming environments (e.g. the Additional hand cursor (Hanks, 2002) and the Transparent Video Facetop (Stotts, McC.Smith, & Gyllstrom, 2004)). One must question whether it would be more appropriate to provide specifically tailored tools for collocated collaborative software development rather than shoe-horning existing resources into collaborative use. In particular, the environments observed did not provide a means for a particular programmer to tailor the environment according to their needs. That is to say, the ergonomics of the pair programming environments left something to be desired. This is an issue that needs to be addressed if pair programmers are to avoid hazards such as repetitive strain injury.
Quite apart from the implications for pair programming tools, this analysis of tools and setting plays an important part in our appreciation and understanding of the context in which pair programming takes place. Having provided this context, we now focus in on the pair themselves and the manner in which they work together on the programming task at hand.
5. The role of level of collaboration

No-one’s mouth is big enough to utter the whole thing
- Alan Watts

5.1. Introduction

We have identified distinct ways in which a pair might work together: co-operation and collaboration. In particular, Sections 2.4.2 and 2.4.3 discussed their potential benefits and shortcomings. Here these concepts are used to more clearly understand the division of labour in pair programming. This is achieved by analysing the talk of the programming pair in order to measure the extent to which each party contributes new information to the subtasks on which the pair works. In doing so we consider a task to be collaborative when both parties are contributing similar amounts of new information, but consider it a co-operative task where only one programmer contributes. In particular, we focus on the following questions:

A. What is the task layout of a ‘typical’ one-hour pair programming session?
B. To what extent do pair programmers actually ‘collaborate’ on the same task?
C. Are certain types of task more or less collaborative in nature?
D. Does a particular role (driver or navigator) contribute more strongly to a particular type of task?

In Section 5.2 we provide the background for our studies and in Section 5.3 we discuss the methodology used both for gathering the data and for its analysis. Our findings are reported in Section 5.4 and in 5.5 we discuss the implications and limitations of these findings. We conclude in Section 5.6 by posing some outstanding questions and suggesting how further insight may be gained in this area.
5.2. Background

As in Chapter 4, this chapter focuses on the experienced pair programmers across the four studies, determined as those with a minimum of six months’ commercial pair programming experience. Thus the same twenty-four sessions amounted to the analysis of 14,886 sentences.

5.3. Method

The methodology used in this Chapter is a form of verbal protocol analysis. In particular it followed the framework set down by Chi (1997) in which protocols are produced, transcriptions are segmented and coded according to a coding schema, depicted in some manner and patterns are sought and interpreted.

Here each one-hour recording was transcribed and segmented into utterances (an utterance typically being a sentence). A coding schema was produced by reducing the work in each of the sessions into a tree of numbered subtasks (e.g. see Figure 5.1 for an example and Appendix B.1 for an actual subtask diagram). These subtasks were derived from the dialogue by considering the sub-tasks that were actually performed in order to complete the overall task for the session. It should be noted that the pair often returned to work on a subtask, but in this case no additional node was added. That is, each subtask is present only once in the structure, in the position in which it first occurred. The derived tasks were at a level of abstraction higher (i.e. less detailed) than writing a line of code but a lower level than the overall task itself. As they were based solely on participants’ utterances, the level of detail of these subtasks was automatically defined by the level of detail at which the participants spoke about the task. They were typically either:

- Things which needed to be done
- Things which needed to be understood
- Things which needed to be decided
- Things which needed to be ‘broadcast’ (outside of the pair)

Further division into verbalised sub-sub-tasks etc. was common during the process of deriving sub-tasks.

Utterances were then analysed for their contribution of new information. By ‘new information’ we mean information contributing to the sub-task at hand that had not previously been
discussed in the session (of course, it is possible that this information had been discussed by the pair outside of the observed session). Any such utterance was then coded with the number of the subtask the information was contributing to, the contributor (A or B) and their role at that time (navigator or driver - note it was usual for participants to change roles several times during a session).

![Diagram of subtask decomposition]

**Figure 5.1 Example verbalised subtask decomposition**

In order to analyze the extent to which different types of verbalised subtask fostered or inhibited collaboration, the verbalised subtasks from all sessions were then used to derive a set of generic verbalised subtask types (see Table 5.1 below). The generic verbalised subtasks were then compared with those described in the literature (e.g. Good & Brna, 2004; Pennington, Lee, & Rehder, 1995) to ensure coverage. A difference from those tasks described in Pennington, Lee and Rehder (1995) was the lack of a discrete ‘design’ category. While part of this is covered in ‘agree strategy’, the lack of a design category is not surprising in an XP environment, where there is no ‘up-front’ design task, rather design takes place as part of the coding. Table 5.1 shows the derived generic verbalised sub-tasks used in the analysis. These cover all the verbalised tasks that were identified and therefore categories such as L (Discuss the IDE) were rarely used but are included for completeness. Instances of social chat either within or outside the pair were not considered. See Table 5.2 for an example coding (note that line 4 is not coded as it is considered a continuance of line 2).
Table 5.1 Derived generic sub-tasks

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Agree strategy/conventions</td>
<td>B</td>
<td>Configure environment</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Including approach to take, coding standards and naming conventions</td>
<td></td>
<td>Setting up paths, directories, loading software etc.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 Example coding of dialogue

<table>
<thead>
<tr>
<th>Utterance Number</th>
<th>Participant</th>
<th>Role</th>
<th>Subtask</th>
<th>Generic subtask type</th>
<th>Utterance (new information in bold italics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>Navigator</td>
<td>1</td>
<td>B</td>
<td>So basically we can create a directory...and we can just use...</td>
</tr>
<tr>
<td>2</td>
<td>Z</td>
<td>Driver</td>
<td>2</td>
<td>A</td>
<td>…We put the date that we are going to put the X in.</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>Navigator</td>
<td>-</td>
<td></td>
<td>Right</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>Driver</td>
<td>-</td>
<td>A</td>
<td>So when you look at it you know that it was done on this date.</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>Navigator</td>
<td></td>
<td></td>
<td>Good</td>
</tr>
<tr>
<td>6</td>
<td>Z</td>
<td>Driver</td>
<td>2</td>
<td>A</td>
<td>…Then that’s a standard file</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>Navigator</td>
<td>3</td>
<td>B</td>
<td>I’ll just copy it all over, apart from the update.</td>
</tr>
</tbody>
</table>
5.4. Results

5.4.1. Overall session shape

In order to address the first question identified in Section 5.1 and to obtain some feel for the types of task undertaken, or at least discussed, in an average session we begin by considering the number of contributions of new information provided for each subtask (see Appendix B.2 for totals per session). The mean number of contributions per session for each generic subtask type was then calculated (shown in Figure 5.2).

![Figure 5.2 Distribution of contributions amongst generic sub-tasks](image)

Some of these categories were not normally distributed among the sessions, however, a non-parametric Friedman test showed significant differences between the mean number of utterances in a category (n=24, $x^2=91.86$ (df=11), $p<0.001$). Whilst not significant statistically, it is interesting to note that as a general trend the majority of contributions related to comprehension [G]. However, the number of these comprehension-related utterances varied widely between sessions. Second most common was Writing new code [I], followed by
Testing [C] (both writing and running tests) and Agreeing strategy/conventions [A]. All three of these subtask types were normally distributed. Least common were discussing the IDE, commenting code, corresponding outside the pair and configuring the environment, each of which accounted for less than three percent of the total interaction in an average session. Again, these were general trends rather than statistically significant results.

5.4.2. Co-operation vs collaboration

In response to question B from our introduction (To what extent do pair programmers actually ‘collaborate’ on the same task’), over all of the twenty-four sessions analysed, both partners contributed at least one item of new information to more than 93% of subtasks (see Appendix B.3 for further details). That is, using our definition, the programming pair can be said to have collaborated on 93% of the sub-tasks they performed. This is consistent whether the pair is considered as individual participants or analysed according to the roles that they performed (see Appendix B.4).

In order to analyse the nature and extent of this collaboration further, we now consider the proportional contribution by each participant. That is, we aim to ascertain whether, although both parties contributed, one programmer dominated heavily. This was achieved by totalling the number of contributions for the highest and lowest contributor in each session, and then expressing them as a percentage. Using such a method a completely even (50:50) distribution of contributions would indicate the highest possible level of collaboration. Our results showed the following averages:

- Highest contributor: 69%
- Lowest contributor: 31%

While this does not show an evenly distributed level of contributions between the two programmers in the pair, it does suggest a reasonable level of collaboration. In addition, the same person may actually have been the highest contributor for one task and the lowest for another. This use of highest and lowest contributors for each sub-task, rather than focusing on individual participants, strengthens this result by polarising the two extremes but still finding a reasonable collaboration level. It should, however, be noted that as the level at which the results could be interpreted as a collaboration is quite subjective it is therefore difficult to analyse statistically.
5.4.3. Collaboration and task type

In order to consider question C further (Are certain types of task more or less collaborative in nature), we will now take a closer look at the types of task in which higher and lower levels of collaboration took place. If we calculate the percentage of tasks of each type that were collaborative across participants (i.e. both participants contributed to a task) we obtain the percentages outlined in Table 5.3.

Table 5.3 Percentage of sub-tasks of each generic type that were collaborative across participants

<table>
<thead>
<tr>
<th>Subtask type</th>
<th>Percentage of sub-tasks collaborative across participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>A – Agree strategy</td>
<td>92</td>
</tr>
<tr>
<td>B – Configure environment</td>
<td>81</td>
</tr>
<tr>
<td>C – Test</td>
<td>92</td>
</tr>
<tr>
<td>D – Comment code</td>
<td>83</td>
</tr>
<tr>
<td>E – Correspond</td>
<td>95</td>
</tr>
<tr>
<td>F – Build, compile, check in/out</td>
<td>91</td>
</tr>
<tr>
<td>G – Comprehension</td>
<td>95</td>
</tr>
<tr>
<td>H – Refactor</td>
<td>94</td>
</tr>
<tr>
<td>I – Write new code</td>
<td>95</td>
</tr>
<tr>
<td>J – Debug</td>
<td>94</td>
</tr>
<tr>
<td>K – Find/check example</td>
<td>92</td>
</tr>
<tr>
<td>L – Discuss the IDE</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.3 clearly shows that both partners contributed at least some information to almost all tasks. Only configuring the environment and commenting code had a level of collaboration below 90% and even these were over 80%, although they were rarely performed. As in Section 5.4.2, we will now consider the extent of this collaboration by analysing the relative number of contributions made by the highest and lowest contributors for each sub-task type. Results are shown in Figure 5.3 below.
As before, only a small proportion of the data (one subtask) is normally distributed. In addition, a Kolmogorov-Smirnov test comparing the mean values of highest contributors across subtasks showed no significant difference (n=12, z = 0.615, p = 0.84). This implies that there was no particular difference in the level of collaboration between the various types of subtask. We could say that they all appear to have been conducted equally collaboratively. However, statistical analysis of this is difficult, as it is difficult to decide the threshold to distinguish collaboration from co-operation (for example a split of contributions of 100:0 is clearly unlikely).

However, there appears to be an interesting trend towards the task ‘Agreeing strategy’ [A] being the task for which contributions are least evenly distributed (averaging nearly 80:20 between participants) and the activity most evenly distributed being ‘Refactoring’ [H]. This is discussed further in Section 5.5.3.

### 5.4.4. Collaboration, role and task type

Since we have ascertained that both parties contribute something to almost every task type, in order to further understand the nature and extent of this collaboration, and in response to Question D (Does a particular role contribute more strongly to a particular type of task) we now consider the contributions to each subtask type according to participant role.
Figure 5.4 Percentage each role contributed to each generic subtask type

Figure 5.4 shows the mean percentage of contributions to each generic task type for the driver and navigator. As previously, the data was not normally distributed. Figure 5.4 suggests that contributions seemed fairly evenly distributed between roles. A Kolmogorov-Smirnov test of the normalised driver contributions showed no significant difference between the percentage of driver contributions according to subtask type (n=12, z = 0.818, p = 0.515). In fact, as illustrated above, contributions seem to generally be well distributed across roles. There also appears to be a general trend in which the driver contributes slightly more than the navigator (except in ‘Discussing the IDE’, which happened rarely).

5.5. Discussion

5.5.1. Overall session shape

The sessions analysed appeared to be typical of what one would imagine for agile pair programming sessions in terms of the types of tasks discussed. Most time was spent on comprehension (both understanding the current code base and understanding the problem), which suggests that pairs were following the XP practice of “incremental design” (Beck & Andres, 2004). That is, the pairs were taking the time to fully understand the current code and see how the new code required would fit in with and enhance it. This was followed by a large amount of time writing new code, testing and agreeing strategy, which highlights the pairs as being very task-focused and code-centric as is generally assumed in the extreme programming approach. In addition, the absence of a derived ‘design’ category shows that the focus on these projects really did seem to have been shifted away from the creation of a large, up-front design and the very few discussions regarding commenting code suggest a belief in the concept of
‘self-commenting code’, which pervades the XP literature. The low number of utterances regarding both the Interactive Development Environment that was being used and configuring the environment suggests that the project environment and tools were well established and did not need much consideration in performing general programming tasks. This is unsurprising on projects of this level of maturity.

5.5.2. Co-operation vs collaboration

The analysis of each partner’s contribution to subtask showed that 93% of all sub-tasks had new information contributed to them by both participants. This suggests an extremely high level of collaboration and shows that the programming pair really are working together on the vast majority of tasks.

5.5.3. Collaboration and task type

Considering these contributions by sub-task type (see question C in Section 5.1), it appears that there is a general trend for contributions to be evenly distributed for each type of task. The sub-task on which one partner is most likely to contribute more new information is ‘configuring the environment’ [B], which was rarely discussed. There is also a trend to suggest that the activity whose contributions of new information were most evenly distributed across participants is ‘refactoring’ [H]. This is unsurprising, given the high cognitive load associated with considering both the current and potential future organization of code at the same time.

5.5.4. Collaboration, role and task type

Contributions also appeared to be generally well distributed across roles with the driver contributing slightly more than the navigator across all but one subtask type, ‘Discussing the IDE’ [L], which happened rarely. This suggests that the driver and navigator roles are less ‘tuned to different tasks’ but more a convenience in terms of who types. Considering that they were taxed with the additional cognitive load of typing, it might seem surprising that drivers contributed verbally more to the task at hand. Perhaps this could be because they were simply commentating on what they were typing as they went along. This finding is also contrary to findings by Chintakovid, Wiedenbeck, Burnett & Grigoreanu (2006), who found that the navigator tended to talk more during sessions of spreadsheet debugging. However, there are a number of differences that could account for this disparity. First, Chintakovid et al. (2006) were counting the amount of verbalisations in a general sense, rather than contributions of new information. Second, the domain of spreadsheets is different in terms of complexity and
problem type to that of object-oriented programming. Third, the user groups were very different (students as opposed to professional software engineers) and finally, the generic task of debugging plays only a small role in the more general nature of the sessions analysed here.

5.5.5. Study limitations

Here we highlight some limitations of the study. First, the sample of companies and projects studied was purely opportunistic. That is, they were those companies who ‘fitted the bill’ in terms of software development approach and pair programming and were also willing to participate in a study. The only further selection for this chapter was the exclusion of sessions where both programmers had not been commercially pair programming for a minimum of six months. Similarly, the pairs observed were those who were happy to be watched and participate in the study (although only two pairs declined). Therefore the results reported may not generalise to programmers in other companies or with differing levels of experience, inclination to be observed or a different approach.

Another limitation of this work is the method of data collection. Ideally the transcribed verbalisations would be complemented with video and screen information and other such contextual information. Unfortunately due to the sensitive nature of the projects involved, video recording was only possible at one of the participating companies and screen-recording equipment was not available for use.

We have also assumed that much can be gleaned from the discussions between the programming pair. Although we have based our findings here on the pair’s utterances, we are aware of the many other subtle ways in which the pair (and others on the project) communicate and exhibit collaboration (for example, where their attention was on the screen, how they manipulated the IDE or when they used particular facial expressions or gestures. Some of these have been reported elsewhere (Bryant, Romero, & du Boulay, 2006a) and have been outlined in Chapter 4 but are excluded from the analysis reported here.

Finally, it was not possible to double-blind code the transcriptions for accuracy. This limitation occurred for two reasons: first, the highly contextual quality of the transcriptions made it very difficult to code for sub-task without having been present (both the analysis and the observation was performed by the author); Second, the sheer volume of transcribed data
made second coding of even 10% impractical within the project constraints.

5.6. Conclusion

The main purpose of this study was to explore the collaborative nature of pair-programming. In particular, it focused on ascertaining whether a set of observed pair-programming sessions were more collaborative or co-operative in nature. This was achieved via the transcription and analysis of the dialogue produced by experienced programming pairs during a set of one-hour sessions. It focused on four main questions.

Question A considered the typical task layout of a one-hour pair programming session. An analysis of the type of generic subtasks that were discussed by the pair programmers suggests that an agile approach favours certain types of activities over others. For example, comprehension tasks, writing new code and testing were most often discussed whereas commenting code was rarely talked about. This is in line with what is suggested in the Extreme Programming literature.

Question B considered the extent to which pair programmers collaborated on the same task. Based on the definition of collaboration as ‘both partners contributing to each subtask’ the results indicate that in these circumstances pair programming is extremely collaborative in nature.

Question C asked whether certain types of task were more or less collaborative in nature. Our findings suggest that this is indeed the case. The highest level of collaboration took place on refactoring tasks, which by their very nature are highly complex. The least collaborative types of task were seen to be those in which the pair were configuring the environment, which happened rarely on the projects concerned.

Question D asked whether a particular role contributed more strongly to a particular type of task. This was found not to be the case, rather that the driver tended to contributed slightly more across almost all task types, possibly due to their commenting on what they were doing.
5.6.1. Further work

The findings discussed in this chapter suggest that the benefits derived from ‘pairing up’ on software development tasks are not achieved through a ‘divide and conquer’ approach to the tasks at hand, but elsewhere. In Chapter 6 we now consider another potential division of aspects within a task, based on levels of abstraction.

It would also be interesting to see how similar data from naturally occurring collaborative episodes in a non-agile project compared. Such a comparison could consider both general verbosity and the generic sub-task types discussed, in order to assist in ascertaining the defining features of the two approaches and their relative strengths and weaknesses.
6. The role of role

“But the emperor has nothing on at all” cried a little child
- Hans Christian Anderson

6.1. Introduction

In Chapter 5 we have shown that rather than co-operating on a task by splitting it up into sub-tasks and working on different components, experienced pair programmers tend to work together collaboratively. We also indicated that both participants, irrelevant of role, contribute to almost every sub-task.

In this Chapter we will continue our pursuit of aiming to understand experienced pair programming by considering how a pair might work together on the same sub-task. In particular we will investigate whether there is a relationship between the driver or navigator role and the levels of abstraction at which they concentrate their efforts. We will focus on the navigator role and compare navigator behaviour with that of the driver, seeking differences.

As discussed in Section 2.5.3 evidence provided from the literature on the driver and navigator roles suggests two possible realisations of the role of the navigator:

- The navigator as reviewer
- The navigator as foreman

These suggestions actually span two different concepts. First, they delineate between two domains: the programming domain and the problem domain; second, they suggest that the programming domain may then be further defined using the model of a series of ‘levels of abstraction’. Use of the phrase ‘levels of abstraction’ is somewhat problematic. Here we refer
to ‘levels of abstraction’ to consider both level of granularity within the programming domain and a separation of program domain from problem or ‘real world’ domain. This is possible since, having first distinguished between the problem and programming domains, it is only necessary to further delineate level of granularity in the programming domain to investigate the concepts of ‘navigator as reviewer’ and ‘navigator as foreman’. It is therefore unnecessary to create two separate levels of coding.

In this chapter we will be using verbal protocol analysis to explore each of these. In Section 6.2 we discuss our application of verbal protocol analysis and define the method we have used to analyse our data in search of evidence for the existence of these role realisations. We have chosen to examine pair programmer’s utterances in the knowledge that by focusing on ‘talk’ alone we are not considering other manners in which the pair communicate (see Section 4.5. for some examples of these). We then provide study background in Section 6.3. In 6.4 we present our results on both the ‘reviewer’ and ‘foreman’ realisations of the role of the navigator. In 6.5 we discuss these findings along with the limitations of our studies and we present an alternative perspective on these roles: the driver and navigator as a cognitive ‘tag team’.

**6.2. Method**

The methodology used for this work follows the framework for verbal protocol analysis set down by Chi (1997) in which protocols are produced, transcriptions are segmented and coded according to a coding schema. They are then depicted in some manner and patterns are sought and interpreted. The coding derived is shown in Table 6.1. It was based on that used by Pennington (1987) to analyse the level of detail of programmers’ statements. In addition, based on Good and Brna’s (2004) coding scheme for programming summaries, a BRIDGE code was included for use when utterances bridged or connected the real or problem domain and the programming domain. Finally, in order to consider the hypothesis that part of the navigator role is to correct spelling and programming grammar, a code for SYNTAX was added.

The coding scheme is intended to be exhaustive, hence the inclusion of a ‘VAGUE’ category in order that every sentence has a corresponding code. In this chapter, the levels of abstraction are presented in an order from the most granular or detailed to the most over-arching. In Section 2.5.2 we discussed issues in determining levels of abstraction and domains in
programming, however here we use the term ‘levels of abstraction’ to cover both the programming and problem domain. For the purposes of this thesis, the problem domain is considered to be at a higher level of abstraction than the programming domain, and therefore appears later in our scale.

Table 6.1 Scheme for coding utterances by level of abstraction

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>SY</td>
<td>Syntax – Spelling or grammar of the program. Spelling is indicated in the transcriptions by single letter capitals. NOT semantics. Key presses.</td>
<td>S P E L L I N G, dot, F9, 7</td>
</tr>
<tr>
<td>D</td>
<td>Detailed – refers to the operations and variables in the program. A method, attribute or object which may or may not be referred to by name.</td>
<td>This condition, that return value, the list, I, the counter, what <strong>this</strong> returns or gives, getCustomer.</td>
</tr>
<tr>
<td>PR</td>
<td>Blocks of the program. Including tests and abstract coding concepts. Also strategy relating to the program and its structure. General naming standards discussions etc. This could also include cases where the subject of the sentence refers to ‘some of them’ or ‘they all’ – i.e. a group of conditions. Anything to do with refactoring. Subsystems or libraries. Directories or paths, even if named.</td>
<td>That loop, truncation, the error handling, Oracle, this issue. this part of the program, mock, Mosaic.</td>
</tr>
<tr>
<td>BR</td>
<td>The statement bridges or jumps between the real world or problem domain and the programming domain. This may be where a case or condition exists in the code and the real world.</td>
<td>So we need to add a test condition here, to see if the bank account is valid for this kind of transaction.</td>
</tr>
<tr>
<td>RW</td>
<td>Real world or problem domain</td>
<td>Savings account</td>
</tr>
<tr>
<td>V</td>
<td>Vague, including metacognitive statements and questions about progress or understanding. References to a place on the screen. References to the development environment and/or navigating its menu structure.</td>
<td>Oh, yeah, I see, that bit at the top.</td>
</tr>
</tbody>
</table>
Each one-hour recording was transcribed and segmented into utterances (an utterance typically being a sentence). Utterances of ‘social chat’ and conversations outside of the programming pair were discounted from this analysis. Four sessions (one randomly chosen from each company) were blind double-coded with an accuracy of 77%. These four sessions account for 14% of the total number of pages. An example section of coding is shown in Table 6.2.

Table 6.2  An example section of coding

<table>
<thead>
<tr>
<th>Participant</th>
<th>Role (Driver/Navigator)</th>
<th>Utterance</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>If you do a dot dot dot there…umm….and go to…</td>
<td>SY</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>You drive…it’s easier</td>
<td>V</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>It is.</td>
<td>V</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>It’s just (sub-system name)</td>
<td>PR</td>
</tr>
<tr>
<td>B</td>
<td>N</td>
<td>What’s (sub-system name) in</td>
<td>PR</td>
</tr>
</tbody>
</table>

A repeated measure analysis of variance was then performed with two variables: level of abstraction, which was considered as a within subjects variable, and role (driver or navigator) which was considered as between subjects.

6.3. Background

The findings presented in the rest of this chapter are based on the same four on-site studies of commercial pair programmers. Only sessions where both programmers had more than six months’ commercial pair programming experience are considered here. That is, the same twenty-four sessions that were analysed for sub-task contributions in Section 5.4 are here re-analysed with regard to the level of abstraction at which the programmers spoke. These 24 sessions resulted in the analysis of a total of 14,886 utterances.
6.4. Results

6.4.1. The pair programming session

We begin by considering the ‘shape’ of the programming sessions observed. Figure 6.1 shows an average for the number of utterances of each type across all sessions, with each level of abstraction shown as a percentage of the utterances in a session. As can be noted from Figure 6.1 a large number of sentences fell in the ‘vague’ category. In fact, an average of 57% of the utterances in a session were classed as ‘vague’. This is not surprising, as only sentences with a defined level of abstraction would fall outside this category.

There were two main cases where the vague category occurred: first, when utterance did not seem to refer to any level of abstraction, for example questions, such as ‘How should we do this?’, simple agreements or disagreements (‘yes’, ‘I don’t think so’) or statements about progress (e.g. ‘We’ve finished that already’). Second, there were some statements where the level of abstraction could not be ascertained simply by reading the transcription. For example, ‘that’s going to work’, which could refer to a line of code, a test, a subsystem, syntax or indeed be a bridging statement between the code and the program.

The vague category involves high levels of utterances that, while interesting as a phenomenon, are not relevant to our hypothesis. As such, this category has been removed from further analysis to avoid it having a misleading effect on our results. This categorisation of ‘vague’ is, in part, due to the post-hoc analysis of the programmer’s utterances. That is, the lack of additional contextual information (that might otherwise have been provided, for example, via video footage) made it difficult on occasions to ascertain the object of a sentence.
After verifying that its assumptions had been met, a repeated measures ANOVA was performed with level of abstraction as a within-subject variable and role as between-subjects. There were main effects for level of abstraction ($f(3) = 110.05, p < .01$) but no main effects for role nor interaction effects for the combination of these two variables.

### 6.4.2. Overall level of abstraction

The main effect described above for level of abstraction indicates that the mean occurrences of utterances at some levels of abstraction may have differed significantly from others. Sessions also tended to have fewer utterances at the extreme levels of abstraction (real world and syntax level) and more in the intermediate levels. A single post-hoc test was therefore performed to assess the significance of the increase in frequency for intermediate (‘PR’) level utterances. A T-test was used, comparing ‘PR’ level utterances with all of the other categories combined. The Bonferroni correction was then applied and the results indicated that there was a significantly higher level of utterances at ‘PR’ level ($t = 2.71, p < 0.05$) than at other levels of abstraction.

### 6.4.3. Level of abstraction and role

All utterances in every session were coded by role according to level of abstraction. Note that these results were not the same as those by participant, as within a session a participant would
often change role several times. As previously mentioned, the repeated measures ANOVA performed showed neither main effects for role nor interaction effects for the combination of level of abstraction and role. In general terms, the two roles did not significantly talk more or less than one another. In addition, the navigators observed did not talk more or less at any particular level of abstraction than the drivers.

6.4.4. The navigator as ‘reviewer’

As mentioned in Section 2.4.3, there have been suggestions that part of the navigator role might include continually reviewing the work of the driver, pointing out spelling and syntax errors (e.g. Jensen, 2003; Williams and Kessler, 2000). In order to investigate this we must first consider how often these types of utterances occur. The average number of syntax and spelling (‘SY’) level utterances per session was 14 (of an average total of 620). This amounts to only two percent of the total utterances.

Figure 6.2 shows SY level utterances made by the driver and navigator for each session as a normalised percentage of the total SY level utterances for each session. Over all sessions the driver accounted for 47% of SY level utterances and the navigator accounted for 53% (session 17 had no SY level utterances but is included in Figure 6.2 for completeness). Note that we have not coded which of these SY utterances are corrections and that they could possibly contain a mixture of talking aloud and correcting. It is also likely that the driver would review and correct their own work without saying anything. However, occurrences of SY level utterances were so rare that this is unlikely to affect our findings.

Figure 6.2 Normalised SY level utterances by role for each session
6.4.5. **The navigator as ‘foreman’**

As discussed, clues from the literature also suggest that the driver and navigator might more thoroughly cover the problem space by working at different levels of abstraction. The suggestion is that the driver is working mainly at the lower levels, typing in code and doing other tactical work while the navigator is working more strategically at the higher levels of abstraction, sitting back and considering how the system fits together as a whole and relates to the business domain. Rather like the foreman at a building site might concern himself with how the whole building is fitting together, rather than how each brick is laid. Figure 6.3 depicts how this theory might look in terms of the levels of abstraction we are considering. Note that here we are considering only the ‘navigator as foreman’. However, were we also considering the role of ‘navigator as reviewer’, the level of utterances at SY levels would be reversed for the driver and navigator roles.

![Bar chart](image)

**Figure 6.3** Chart showing theoretical levels for utterances by the driver and navigator were they to work at different levels of abstraction.

Rather than the expected chart in Figure 6.3, Figure 6.4 shows the actual average number of utterances of each level per session for each role, making it clear that in the sessions observed the driver and navigator tended to generally talk at the same levels of abstraction.
6.5. Discussion

6.5.1. The navigator as reviewer

It would seem from our findings, in particular the lack of interaction effects between level of abstraction and role, that contrary to what has previously been reported (e.g. Jensen, 2003; Williams & Kessler, 2000) the role of the navigator does not appear to be defined by their correcting syntax and grammar significantly more than the driver. In fact, utterances at this level were scarce in the pair programming sessions observed. On the infrequent occasions in which they did occur, they were relatively evenly distributed between driver and navigator roles, with the driver accounting for 47% of ‘SY’ utterances and the navigator 53% and no significant difference.

It is, of course, entirely possible that a small increase in quality is gained as although the driver more swiftly notices errors while typing, the navigator picks up those which have gone unnoticed and might otherwise have remained undetected. Nevertheless the notable scarcity of
utterances of this level suggests that the key to understanding the role of driver and navigator lies elsewhere.

6.5.2. The navigator as foreman

Also in contradiction to what has previously been suggested (e.g. Dick & Zarnett, 2002; Hazaan & Dubinsky, 2003), the pair programmers in the sessions observed did not show the navigator working at a generally higher level of abstraction than the driver in their discussions. In fact, rather than working at a higher level of abstraction, the pattern of abstraction levels of navigator’s utterances are very similar to those of the driver and do not differ significantly.

As mentioned in Section 6.4.1, there were significantly more utterances at ‘PR’ level than at the other levels of abstraction. This is surprising because if the navigator was working at a higher strategic level we might expect more navigator utterances at this level and fewer driver utterances.

This poses two further questions: first, if the roles of driver and navigator may not be defined by constant review or levels of abstraction, then how might they contribute to the production of higher quality software? Second, why do the driver and navigator speak significantly more at the level of ‘abstract chunks of code’? In Section 6.5.3 we present an alternative view of the driver and navigator roles based on the evidence presented in this chapter and in Section 6.5.4 we consider the proliferation of ‘PR’ level utterances.

6.5.3. The ‘Tag Team’ – An alternative perspective

Our findings show that, rather than working at different levels of abstraction, the driver and navigator not only work collaboratively on each task (see Chapter 5) but within that task they tend not to talk at significantly different levels of abstraction. In addition, as discussed in Chapter 4, not only do driver and navigator change role regularly, these role changes appear to be very fluid with little accompanying explanatory conversation. These findings imply that the navigator continually maintains a firm grasp of what is happening during the session at a number of levels of abstraction. In addition, our findings on collaboration, presented in Chapter 5, show that the navigator is contributing very nearly as much new information to each sub-task as the driver.
This leads us to suggest that rather than the driver and navigator roles being defined by segmenting the problem space in some way – either by task or level of abstraction – they are more simply defined by the additional physical and cognitive load of typing borne by the driver. In fact, we suggest that the driver and navigator form a kind of ‘cognitive tag team’, working together, in synchrony, at the problem at hand and then switching role to alleviate the additional cognitive load of typing and providing a running commentary, both of which fall on the driver.

6.5.4. The proliferation of talk about ‘Chunks of Code’

It has been suggested that software developers need to be operating at several different levels of abstraction simultaneously (e.g. Pennington, Lee, & Rehder, 1995), for example: considering whether they are successfully solving the business problem at hand; ensuring that they are conforming to project coding standards; making good use of existing modules of code and ensuring that what they type in is ‘grammatically correct’. Our findings show that pair programmers talk significantly more at the level of ‘chunks of code’ than at other levels. This suggests that the ‘PR’ level of utterance may possibly provide a mediating device to help tame the complexity of considering many different levels of abstraction at once. In other words, perhaps ‘PR’ utterances could provide the ‘glue’ that holds together the upper and lower levels of abstraction in order to ease the process of progressing through the programming activity.

Given the ease with which driver and navigator have been seen to change role in our ethnographic research, it is also possible that the ‘PR’ level is used as a means of ensuring the navigator is up to speed in intermediate level terms in preparation for a role change. Lower level comprehension seems to be assisted by the ready availability of the code (helped by the physical layout of the pair - they can both see the screen) and the Interactive Development Environments (IDEs) that were used often made it easy to simultaneously see the code currently being written and where it fitted into the overall code base. Perhaps through ‘PR’ level utterances the driver and navigator together provide a missing level of information that is essential if the navigator is to be able to ‘take over’ in a fluid manner.

6.5.5. Study limitations

The studies discussed in this chapter have a number of limitations in common with the previous chapter. First, the sample of companies and projects was opportunistic. Second, the
data collected was limited to audio recordings. Third, the subsequent analysis therefore focuses on the pairs ‘talk’ without considering the other ways in which they communicate (as discussed in the Chapter 5).

Somewhat unusually, role was considered as a between rather than within-subjects variable. This was due to the manner in which the data was initially coded and precluded observations about how a particular individual behaved when in the driver or navigator role.

There may also be other levels of abstraction outside of those used in this analysis. Indeed there may even be different perspectives along which levels of abstraction could be plotted which might highlight role differences more centrally or more convincingly.

Finally, while double-blind tests of the refined coding schema yielded an inter-rater reliability of 77%, a Kappa test resulted in a coefficient of K=0.64. Generally a coding scheme is considered robust with a Kappa coefficient of K=0.7 or above. In this case, disagreements in the coding were largely due to the second coder lacking the contextual understanding and specific programming language knowledge required. In test sessions all disagreements were resolved through further explanation on the part of the primary coder. The overall coding should hopefully retain accuracy as it was the primary coder, with the required contextual and programming knowledge, who performed it.

### 6.6. Conclusions

Although literature on pair programming consistently refers to the roles of driver and navigator, little is known about the mechanisms by which they are realised. In this chapter we have considered the levels of abstraction at which drivers and navigators talk to gain insights into the meaning of their roles. In particular we have used verbal protocol analysis to consider two main issues: does the navigator act as a kind of ‘reviewer’ by catching syntax and spelling errors? Do the driver and navigator work at different levels of abstraction as a way of taming the complexity of each particular sub-task on which they work?

Our findings have been contrary to suggestions in the literature: first, utterances regarding syntax and spelling are rare, and when they do occur are not predominantly made by either the navigator or driver. Second, the driver and navigator do not work at significantly different
levels of abstraction but rather remain in step through the problem working together. Most discussions take place at ‘abstract chunk of code level’.

We have suggested that the driver and navigator form a cognitive tag team, where they work collaboratively on each sub-task and the navigator is at the ready to relieve the driver of the additional loads of typing and commentating. We also posit that ‘PR’ level utterances, referring to the code in an abstract way, may assist in taming the complexity of working at many levels of abstraction at once by providing the ‘glue’ that holds these levels together and which might otherwise have been missing as it is not readily available representationally to the pair.

In the next chapter, we pool findings from this and previous chapters to derive a set of ‘pair programming behaviours’.
7. Pair programming behaviours

We must endure our thoughts all night, until
The bright obvious stands motionless in the cold
-Wallace Stevens

7.1. Introduction

This chapter summarises the findings from the previous chapters in a set of pair programming behaviours. The provision of such a set is useful in a variety of ways. First, they might assist current pair programming practitioners by: helping them to reflect upon their needs and practices; highlighting additional considerations; suggesting how their practices might evolve; helping them to identify common pair programming problems; and suggesting solutions to those common problems. Second, they are helpful to those considering the use of pair programming as they highlight particular considerations and outline key desirable and undesirable pair programming behaviours. Third, they may be useful to those using pair programming in teaching and learning as they define desirable behaviours exhibited by successful pairs.

The behaviours in this chapter are of three types: first, those which were present in the observed sessions described in this thesis and which were perceived by the author as special to experienced pair programmers. Second, they detail a set of behaviours that were seen as undesirable or symptomatic of novice pair programming. Finally, they specify a group of behaviours that were sought (following references in the literature) but not found. These behaviours are not actually patterns, which originate from a set of Design Patterns conceived by the architect Christopher Alexander (Alexander et al., 1977) and have special technical significance in terms of their identification, definition and documentation and. However,
seeking an appropriate and standard notation for the documentation of these behaviours has led to them being expressed using modified versions of the ‘gang of four’ pattern notation (Gamma, Helm, Johnson & Vlissides, 1995) and the anti-pattern notation used by Brown, Malveau, McCormick and Mowbray (1998).

7.2. Desirable ‘expert’ pairing behaviours

This section describes behaviours that were both observed repeatedly during the studies of experienced pair programmers described in this thesis, and that were perceived to be desirable.

7.2.1. The ‘Cohabitor’

Intent
Provide a suitable environment for pair programming, including all resources, without requiring additional dedicated desk space.

Also Known As
Desk-sharing or ‘moving in’.

Motivation
Programming together in a pair necessitates environmental considerations beyond those for an individual. However, a dedicated pair programming area is not always feasible in the commercial programming world, where space is often limited and in high demand. In addition, it is often useful for a pair to be ‘in the midst’ of their team when working on their assigned task, thereby maximizing peripheral awareness in the team by making it easy to overhear their conversations.

We can solve this problem by ensuring that each individual programmer has a desk suitable for both working alone and pair programming. For example, providing suitable sized desks, spare chairs and additional or enlarged computer monitors.

Applicability
Use the cohabitor when:

- desk space is at a premium.
- dedicated pair programming space is not available.
• visibility (and audibility) to the rest of the team is important.
• a programmer’s desk has the essential resources for pairing (physical space, suitable displays, chairs).

Participants
The programming pair.

Collaborators
Peripheral awareness by the rest of the project team.

Consequences
The cohabitor has the following benefits and liabilities:

* It may be less comfortable for pairing. The programming pair may have to ‘make do’ with an environment that is not dedicated solely to pair programming.
* It may be more distracting. It is more likely that the programmer at whose desk they are working will be subject to interruptions (e.g. the telephone, email, physical visits).
* It promotes peripheral awareness. The rest of the team are likely to be within earshot (and view).
* It is usually cheaper. Additional, dedicated office space for pairing is not required.

Implementation
Having decided to work together, the pair must then agree at whose desk they are to work. The programmer who is ‘moving in’ must then ensure that they take with them any additional resources that they require.

Care should be taken that the desk is appropriate for pair programming. In particular, displays appear to be most accommodating when they consist of two large, flat screen monitors side by side. Adequate personal space should also be available for each programmer and adjustable chairs should be provided.

Known uses
The cohabitor also ensures that adequate facilities are available when an individual is consulting or being visited by someone rather than actually pair programming.
Related behaviours

An alternative to the Cohabitor, that is suitable when additional dedicated desk space is available, is the Relocator (see Section 7.2.2).

7.2.2. The ‘Relocator’

Intent

Provide a suitable environment for pair programming, including necessary resources, where interruptions can be minimised.

Also Known As

Dedicated pair programming space.

Motivation

Programming in a pair necessitates environmental considerations beyond those for an individual. If additional office space is available, this problem can be solved by creating a dedicated space, tailored to pairing and devoid of distractions such as telephones. Such an area might have particularly large and suitably shaped desks, large or multiple adjustable computer monitors and resources such as whiteboards, reference materials and sketchpads within arm’s reach. In addition, this formalises the process of pairing and makes it obvious when a pair are working together on a task and do not wish to be disturbed.

Applicability

Use the relocator when:

- additional office space is available.
- visibility (and audibility) to the rest of the team is not essential.
- Individuals’ desks are not suitable for pairing or specific additional resources are required for pairing.
- distractions and interruptions need to be minimised.

Participants

The programming pair.
Collaborators
None.

Consequences
The relocator has the following benefits and liabilities:

*It may be more comfortable for pairing.* A dedicated pairing space is more likely to be tuned to pairing requirements. This might include having ample facilities for assisting collaboration, such as white boards and sketch pads.

*It is less distracting.* Telephones are not available and it is more readily obvious to others that the pair are programming together and do not wish to be disturbed un-necessarily.

*It diminishes peripheral awareness.* The rest of the team are less likely to be within earshot of the pair, and therefore are less likely to pick up on areas of overlap. The potential for spontaneous extra-pair help is also diminished.

*It is more expensive.* As individual desks are also required, additional office space is necessary, along with additional sets of equipment for the dedicated pairing area.

Implementation
Having decided to work together, the pair then both re-locate to the dedicated pair programming area.

The pairing area should be suitably furnished for working together. In particular, displays appear to be most accommodating when they consist of two large, flat screen monitors side by side. Adequate personal space should be available for each programmer and adjustable chairs should be provided. Where possible, a variety of keyboards, mice and other peripherals should be made available in order to tailor the environment for individual requirements. In addition, resources to facilitate collaboration, such as whiteboards, sketchpads and reference materials should be easily accessible.

Known uses
The relocator also provides alternative working space for individuals who require an area free from distractions; or for groups of two or more who require a space away from their individual desks for reasons other than pair programming.
Related behaviours
An alternative to the Relocator is the Cohabitor (see Section 7.2), which is suitable when no dedicated space is available.

7.2.3. The ‘Tag Team’

Intent
Minimise cognitive load by changing role between driver and navigator.

Also Known As
Role change, hand over, new driver.

Motivation
Even programming alone is cognitively very demanding. In pair programming, as well as working on the task at hand with his/her partner, the driver has two additional cognitive loads: typing and providing a verbal commentary on what he is doing. In addition, there may be times when the navigator is more suited to driving, for example when highlighting an issue with an example.

We can overcome these issues by changing over role between the driver and navigator as seamlessly as possible.

Applicability
Use the tag team when:
- the driver is cognitively overloaded.
- the navigator needs to show an example.
- the driver is tired.
- the navigator needs to be encouraged to provide input.
- both programmers would prefer a change of driver.

Participants
The programming pair.
Collaborators
The keyboard and/or mouse.

Consequences
The tag team has the following benefits and liabilities:

*Cognitive overload is avoided.* The driver can take a rest from typing and commentating.

*Navigator input can be encouraged.* The driver can hand over if the navigator is ‘drifting off task’.

*Specialist knowledge can be maximised.* The partner most able to program a particular chunk of code can do so; tangible examples can also be typed in.

*Additional verbalisation is required.* The driver must keep the navigator up to speed if he/she is to be ready to spontaneously change role.

*Driving ability must be present.* Both partners must be able to type and otherwise manipulate the computer.

Implementation
One partner, having identified a requirement to switch roles, may initiate the switch in a number of ways.

- The driver or navigator may ask verbally. While driver initiated verbal requests of this type are not uncommon, such navigator-initiated requests are rare.
- The driver may simply slide the keyboard and/or mouse over to the navigator, implying that a change of role is being requested.
- A ‘surrogate’ artefact may be used (as was seen with the surrogate mouse in Section 4.4.7). Initiating possession of this ‘surrogate’ subtly suggests that possession of the keyboard and mouse (and therefore the driver role) has been relinquished.

Known uses
The tag team approach also requires that both partners remain ‘on task’. This ensures that knowledge distribution is achieved and may also assist in procuring some of the other perceived benefits of pair programming mentioned in Section 1.3 (for example, peer pressure, confirmation bias and review).
Related behaviours
The ‘tag team’ is parent to the ‘intermediate level of talk’, described in Section 7.2.4.

7.2.4. The ‘Intermediate level of talk’

Intent
Provide information at a level not available from existing representations.

Also Known As
Talking about areas of the system (e.g. the error handling).

Motivation
Programming is a complex skill that involves integrating knowledge from a number of different levels of abstraction.
This complexity may be eased by encouraging talk at an intermediate level to provide the ‘glue’ between high level concepts (e.g. the business problem being solved) and low level details (e.g. coding syntax and spelling).

Applicability
Use the ‘intermediate level of talk’ when:
• pair programming.
• explaining how a program works.
• existing representations (i.e. The IDE and diagrams) do not provide information at this level.

Participants
The programming pair.

Collaborators
None.

Consequences
Talking at an intermediate level has the following benefits and liabilities:
Cognitive overload is avoided. The intermediate level of talk helps to provide a point of reference for high and low levels of abstraction. Thus the programmer does not need to provide some other form of cognitive ‘glue’.

A collaborative mental model can be maintained. Talk at this level may help to uncover inconsistencies in understanding between the pair.

Peripheral awareness is encouraged. The pair talk is at a level that can be more easily understood when ‘overheard’ by surrounding members of the project team.

Talk at other levels may be forfeited. Depending on the level of detail of the task at hand, talk at other levels of abstraction may be more appropriate.

Implementation
The programming pair talks primarily at an intermediate level of detail. Their discussions centre around ‘areas of the system’ or ‘chunks of code’ rather than focusing on either the programming language or the business problem.

Known uses
Talking at this level of detail also ensures that the navigator is ‘up to speed’ in intermediate terms with the work of the driver. This ensures that the navigator is ready to switch roles as and when required. It also allows the navigator to contribute (almost) equally to each of the tasks in which the pair is involved (see Section 5.4.4).

Related behaviours
Intermediate level talk is a sub-behaviour to the ‘tag team’ described in Section 7.2.3.

7.2.5. The ‘Re-pair’
Intent
Optimise the allocation of appropriate programmers to tasks by allowing fluid changes to pair allocation.

Also Known As
Changing partners.
Motivation
Commercial computer systems are often large and complex and it is unusual for a computer programmer to remain working on the same system for their entire career. As such, it is unlikely that any one programmer currently on a project will have been exposed to all the components of that system. More frequently, a project will (often informally) have a nominated ‘expert’ or two for any particularly complex part of the system. The process by which pairs are allocated to tasks, along with the opaque nature of many computer programming problems, means that the designated ‘expert’ may not always be assigned to the tasks about which they have special knowledge.

Fluid re-pairing helps to solve this problem by ensuring that pair composition can be changed mid-task to ensure optimum allocation of programmers to the problem at hand.

Applicability
Use the re-pair when:
- The task is complex.
- Assistance cannot be found by other resources (books, the internet…).
- The pair currently allocated to the task are struggling.
- Another programmer on the project has the required knowledge to assist with the task.
- This domain ‘expert’ is replaceable on their current task OR their current task is less critical.

Participants
The programming pair.
The domain ‘expert’ outside of the current pair.

Collaborators
When the ‘expert’ is currently pairing, their partner.

Consequences
The re-pair has the following benefits and liabilities:

*The task is more likely to be completed.* The most appropriate expert is working on the problem and therefore is more likely to find a solution.
The task is likely to take less time to complete. A domain expert is more likely to complete the task in an optimal manner.

Knowledge transfer. Through re-pairing, the expert knowledge brought in to the pair may now be transferred, lessening the need for re-pairing in the future and providing a safeguard against attrition.

The ‘leaver’ may be demotivated. Re-pairing means that the pair member who is re-assigned does not get to see the problem through to resolution. This may have a de-motivating effect.

The alternative pair may be less optimal than before. The allocation of pairs to tasks generally takes place at the daily stand-up meeting and is achieved through negotiating who is available and what work is outstanding overall on the project. As the expert has been removed from their original pair to work on this pairs’ task, this original pair may now be poorly resourced to solve the task at hand.

Implementation
When a pair is having difficulty solving a particular problem or completing a task there are three ways in which re-pairing may be initiated. First, the pair may explicitly request advice from another colleague. Alternatively another programmer on the project may overhear the pair having difficulty. Either of these two cases may or may not eventually lead to re-pairing. Finally, a third party (for example, the project manager) may suggest re-pairing.

Once re-pairing has been suggested, it generally does not take place unless all stake-holders agree. This includes: the domain expert agreeing to join the pair who are having difficulties, the ‘leaver’ agreeing to leave the pair and join the alternate pair, the ‘stayer’ agreeing to remain in the pair and work on the original task with the domain expert and the remaining member of the alternate pair agreeing to remain on that task but now work with the ‘leaver’. If all are in agreement the re-pairing takes place and typically the domain expert and the ‘stayer’ do not physically relocate, rather the ‘task’ remains at the original desk.

Known uses
The re-pair can be used:

- When the pair is stuck and another member of the team can help.
- When another member has previous knowledge of the particular type of problem.
- When another member has previous knowledge of the area of the system (but not necessarily the particular problem).
• When another member has superior theoretical knowledge of the problem and/or area.

**Related behaviours**
Physical implementation of the re-pair is via either the Cohabitor or the Relocator.

### 7.2.6. The ‘Break’

**Intent**
Ensure the pair remain refreshed by taking adequate breaks.

**Also Known As**
Time out, coffee time.

**Motivation**
Programming is a highly demanding task, requiring intense concentration and the maintenance of information at a number of different levels of abstraction at the same time (see Section 2.5.2). Pair programming in particular, is considered to be quite exhausting, possibly because of a combination of the additional load of verbalising and the pressure of being observed (along with the changes in behaviour it may provoke, for example not taking micro-breaks). We can ease this problem by ensuring that pair programmers take regular breaks. These may be either ‘formal’ breaks, (e.g. leaving their desks and taking a coffee break) or simply taking a few minutes off-task for a social chat.

**Applicability**
Use the break when:

- Either programmer is tired.
- A more creative solution is required.
- No progress seems to be being made on a problem or the programming pair feels like they are going round in circles.
- A higher level of ‘objectivity’ is required.
- Off-task work needs to be done.

**Participants**
The programming pair.
Collaborators
Sometimes other members of the team.

Consequences
The break has the following benefits and liabilities:

*It fosters creativity.* An ‘incubation’ period (incubation is one of four essential creative processes described by Poincare (1982)) is made available, from which more creative solutions may arise.

*It avoids burn-out.* As it ensures adequate breaks are taken, which may increase productivity by assisting the programming pair in returning refreshed to fully engage with the task at hand.

*It encourages sociability.* The pair engage in social chat sometimes with others.

*It provides a window to discuss general issues.* For example, tactical issues regarding working together or tools that might be helpful.

*It forces the pair away from the code.* Therefore, if the pair discuss the task it may foster conversation at a higher level of abstraction.

*It may be poorly perceived.* Other surrounding projects with a different culture may not be aware of the benefits of breaks, and may consider the pair to be ‘slacking off’.

Implementation
Usually one programmer in the pair suggests taking a break, however on occasions the environment may make it impossible to continue with the task at hand (for example, integration tests may take a while to run). Breaks may be taken in a number of ways:

- The pair may take a break at their desk.
- The pair may retire to another area, typically with drinks-making facilities. In this case the pair will often ask other, neighbouring pairs to join them.
- One or more of the pair may remain at or return to their personal desk to perform non-task related work, such as managing email, making telephone calls or attending meetings.

During the break the tasks on which they are currently engaged may or may not be discussed.
Known uses
The break is used to take more ‘formal’ coffee and meal breaks, usually at roughly scheduled times. There are often core hours when pair programming does not take place.

7.2.7. The ‘Enforcer’

Intent
Ensure standards are appropriate and enforced.

Also Known As
Quality assurance, review, project standards officer.

Motivation
Understanding computer programs is a difficult task and one that can be eased or complicated by many factors (see Section 1.1). In addition, code written or contributed to by several different programmers can be confusing when each programmer has a different style.

Several mechanisms can be included in a piece of program code with the aim of easing the complexity of code comprehension. These include the production of and adherence to a set of coding standards. This helps to ensure that coding standards are followed by one partner reminding the other of them and/or discussions taking place on how they are best implemented.

Applicability
Use the enforcer when:

- The current driver deviates from a known or perceived standard.
- An existing piece of code is difficult to understand and modifying it by enforcing a standard may help.
- There is no current standard of this type, but one of the pair uses it personally.

Participants
The programming pair.

Collaborators
On occasions the rest of the team.
**Consequences**

The enforcer has the following benefits and liabilities:

*Code should be more readable.* In particular, enforcing the use of standards should ensure that when reading a new piece of code the pair does not first have to understand the conventions used by a particular programmer.

*Code may be easier to write.* ‘Style’ type decisions have already been made and merely need to be enforced and adhered to.

*There may be no current standard.* Creating and broadcasting a new standard may be time consuming.

*Negotiating may be time-consuming.* Conversations regarding the meaningfulness of variable names etc. are open to personal preference, and therefore may be lengthy and fruitless.

**Implementation**

One of the pair decides that a standard needs to be enforced on either a new piece of code they are writing or an existing piece of code. Usually a discussion takes place regarding one or more of the following:

- Whether such a standard is in place (formally or informally).
- Whether a new standard is needed (and if so, how it should be defined and broadcast).
- How the standard should be enforced here.

Occasionally, the standard is then broadcast to the rest of the team.

**Known uses**

Examples of standards that are typically enforced include:

- meaningful variable names.
- verb-noun method names.
- consistent prefixes.
- the use of camelCase (capitalising the first letter of each new word in a long name).
- the use of known design behaviours.

**Related behaviours**

None.
7.3. Undesirable pair programming behaviours

This sections details behaviours that were observed during the studies described in this thesis and were considered to be either undesirable when pair programming or symptomatic of novice pair programming.

7.3.1. The ‘Thrasher’

Also Known As
Keyboard ping-pong. Over-writing your partner’s code.

Refactored solution name
Step away from the code.

Refactored solution type
Behaviour.

Root causes
Dogmatism.

Evidence
As described in Section 3.3.3, novice pairs suggested and counter-suggested almost twice as much as experienced pairs. Observer’s field notes also suggested that novices often thrashed between different strategies, depending on who was driving.

Background
Most programming problems have more than one possible solution. A necessary part of the software development process is therefore to assess the suitability of a particular strategy and select the approach that is most appropriate. The ‘thrasher’ occurs when the programming pair is unable to agree on the approach to take.

General form
The driver and navigator switch frequently between solutions.
Symptoms and Consequences

The symptoms of this include:

- Frequent role switches.
- Physically appropriating the keyboard.
- Deleting or over-writing code recently entered by the other partner.
- Rarely discussing without typing in code.
- The driver continues typing, ignoring the navigator’s suggestions.

The consequences include:

- Making no (or very slow) progress.
- Producing disjointed code.
- Irritating one another and those who are nearby.

Typical Causes

The primary causes of the ‘thrasher’ are lack of pair programming experience and lack of negotiation (and other communication) skills.

Known exceptions

There should never be any exceptions to the Thrasher.

Refactored Solution

The solution involves the pair refraining from typing until a mutually acceptable solution has been agreed. If this is not possible, or project timescales do not permit continued discussion, third party intervention should be sought. Ideally this assistance should come from either the team leader (or project manager on small projects) or a more experienced programmer.

Variations

The ‘thrasher’ can occur at a number of levels of detail, from the overall design of the solution to issues such as naming standards and the use of parentheses.

Related Solutions

None.
Applicability to other viewpoints and scales
This is applicable to other types of software development and indeed other forms of collaborative problem solving.

7.3.2. The ‘Divider’

Also Known As
Divide and conquer, ‘you do this and I’ll do that’, a co-operative approach.

Refactored solution name
Collaboration.

Refactored solution type
Process and Behaviour.

Root causes
Misunderstanding of collaboration and haste.

Evidence
As discussed in Section 2.4.1, collaboration is one method by which two people can work on one task.

Background
On many software development projects deadlines are tight. In addition, each partner in a programming pair may have different areas of expertise, which vary in their relevance to alternative parts of the problem. Therefore, it may seem less time-consuming to each focus on a sub-task requiring their particular expertise and then pool the results.

General form
The pair begin by dividing up the task into sub-tasks and allocating these between them. Each works (usually in parallel) on their assigned sub-task and the pair then re-group when they are finished to integrate the results.
Symptoms and Consequences
The symptoms of this include:

- The pair not sitting together.
- The pair not engaging together on one task at hand.
- The pair performing obviously different tasks (e.g. One typing in code while the other draws a diagram or looks up an example).
- The pair not talking to one another.

The consequences include:

- Additional effort integrating the two solutions.
- Disjointed code.
- Minimising learning opportunities and knowledge distribution.
- Failure to reap some of the benefits of pair programming discussed in Section 2.4.2.1 (peer pressure to perform at one’s best, reduction in errors due to confirmation bias, apprenticeship and learning, and reviewing work for mistakes).

Typical Causes
The primary cause of the divider is tight project deadlines. It typically occurs when this is coupled with a belief that quickly providing working code is more important than quality issues and other pair programming benefits (see Section 2.4.2.1).

Known exceptions
There should be no exceptions to the ‘divider’.

Refactored Solution
The pair should take a collaborative approach to the problem, working alongside one another on all aspects of the task at hand. In particular, each partner should contribute to, and be involved in conversations about, almost every sub-task performed at every applicable level.

Variations
None.
Related Solutions
This solution is related to the ‘intermediate level of talk’ (see Section 7.2.4).

Applicability to other viewpoints and scales
This behaviour may not be applicable to conventional software development projects, where a divide and conquer approach is generally taken.

7.4. Absent pair programming behaviours
This section details a set of behaviours that were sought during the studies described in this thesis (following references in the literature) but not found.

7.4.1. The ‘Reviewer’
Also Known As
Continual review. Navigator reviews syntax and spelling.

Refactored solution name
Working together at different levels of abstraction.

Refactored solution type
Behaviour.

Root causes
Misunderstanding.

Evidence
The notion of ‘navigator as reviewer’ is common in the pair programming literature (see Section 2.5.3).

Background
The roles of driver and navigator are difficult to define. The pair programming literature suggests that the role of the navigator might be to provide a continual review. In particular, it implies that the navigator should continually correct the syntax and spelling of the code that the driver has just typed in.
General form
The navigator continually suggests small corrections, often when the driver is about to correct them.

Symptoms and Consequences
The symptoms of this behaviour include:
- The navigator not contributing at other levels of abstraction.
- The navigator ‘pouncing’ on spelling and syntax mistakes.

The consequences include:
- The pair fails to achieve the potential benefits of fully collaborating (see Section 2.4.2.1).
- The driver becomes frustrated.
- The navigator becomes bored.

Typical Causes
The ‘reviewer’ is typically caused by a belief that the role of the navigator is to continually review the code being typed by the driver.

Known exceptions
There should be no exceptions to the ‘reviewer’. However, this is not to suggest that the navigator should not occasionally make corrections at this level. Rather, reviewing is only a small part of the collaboration that takes place between a successful programming pair and is performed almost evenly by the driver and navigator (see Section 6.5.1).

Refactored Solution
The pair could more effectively work together by both contributing at many different levels of abstraction. In particular, rather than being constrained to providing spelling and syntax corrections, the navigator could get involved in all aspects of the task at hand. This also helps to maximise learning opportunities and facilitate knowledge transfer.
Variations
None.

Related Solutions
This solution also applies to the ‘foreman’ (see Section 7.4.2). It is related to the ‘collaboration’ solution (see Section 7.3.2) and the ‘intermediate level of talk’ behaviour (see Section 7.2.4).

Applicability to other viewpoints and scales
The reviewer may not be applicable on projects taking a conventional approach, in particular where some kind of formal reviewing process is in place.

7.4.2. The ‘Foreman’

Also Known As
The navigator works strategically. The navigator focuses on the ‘bigger picture’.

Refactored solution name
Working together at all levels of abstraction.

Refactored solution type
Behaviour.

Root causes
Misunderstanding.

Evidence
The notion of ‘navigator as foreman’ is common in the pair programming literature (see Section 2.5.3).

Background
The roles of driver and navigator are difficult to define. The pair programming literature proposes that the role of the navigator could be to work at a higher level of abstraction than the
navigator. In particular, it suggests that the navigator might work strategically and focus on considering the business problem and how the solution fits together.

**General form**
The navigator relates the work being done by the driver back to the business problem and suggests strategic options for how the solution might fit together.

**Symptoms and Consequences**
The symptoms of this behaviour include:
- The navigator talking at a higher level of abstraction.
- The navigator not getting involved in the lower-level coding tasks.

The consequences are/include:
- The driver and navigator having difficulties communicating.
- Intermediate levels of abstraction not being covered.
- Lack of opportunities to realise the potential benefits of fully collaborating (see Section 2.4.2.1).

**Typical Causes**
The foreman is generally caused by an inappropriate belief that the role of the navigator is only to work strategically. The behaviour occurs when this is coupled with a belief that the pair programming literature is prescriptive in nature.

**Known exceptions**
There should be no exceptions to the foreman. However, this does not mean that the navigator should never work strategically. Ideally, the driver and navigator should work collaboratively on almost all of the tasks that they perform and at all of the appropriate levels of abstraction.

**Refactored Solution**
The pair could more effectively work together by both contributing at many different levels of abstraction. In particular, rather than being constrained to only working strategically and at the higher levels of abstraction, the navigator could get involved in all aspects of the task at hand. This also helps to maximise learning opportunities and facilitate knowledge transfer.
Variations
None.

Related Solutions
This solution also applies to the ‘reviewer’ (see Section 7.4). It is related to the ‘collaboration’ solution (see Section 7.3.2) and the ‘intermediate level of talk’ behaviour (see Section 7.2.4).

Applicability to other viewpoints and scales
The ‘foreman’ may not be applicable to conventional programming, where a system designer may be concerned with being ‘foreman’ while the programmer is concerned with coding the design with which they have been provided.

7.5. Conclusion
The behaviours outlined in this chapter are based on the observational studies that took place and are described in Chapters 3, 4, 5 and 6. In particular, they further detail those behaviours seen as unusual or special to experienced pairing, symptomatic of novice pair programming or present in the literature but not observed.

These behaviours do not provide an exhaustive account of the different aspects of pair programming, many of which are also present when programming individually or using a particular methodology. Other behaviours took place during the observational sessions studied that could usefully be defined. These were omitted here, as they were not seen as especially symptomatic of experienced or novice pair programming.
8. Conclusions and Future Work

Water which is too pure has no fish
- Ts’ai Ken K’an

8.1. Introduction
In this final chapter we begin by providing an overview the research that was undertaken for this thesis. Section 8.3 then summarises the findings reported in this thesis, and discusses their implications with regard to future studies, practitioners, pair programming environments, teaching and learning and our general understanding of pair programming. In Section 8.4, some limitations of this thesis are highlighted and Section 8.5 suggests areas of future work.

8.2. Summary of this research
Amongst other claims, collaborating on software development has been consistently reported to improve code quality and shorten delivery times. Despite calls for its investigation (Chong et al., 2005; Hughes & Parkes, 2003; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999), little research has been done to further our understanding of how and why this may be the case. In this thesis we looked for evidence that experienced pair programmers behave in ‘different’ or ‘special’ ways that could contribute to these reported benefits. In particular, we focused on four research questions:

- How can we best distinguish between ‘expert’ and ‘novice’ pair programmers?
- How does the use of tools, setting and the environment facilitate successful collaborative software development?
- What is the level of collaboration (as opposed to co-operation) in experienced pair programming sessions?
- What are the defining aspects of the reported driver and navigator roles?
We began by considering how best to delineate between ‘experienced’ and ‘novice’ pair programmers. In particular, we used questionnaire data from 45 commercial pair programmers and their supervisors/peers to consider disparities in these programmers’ ratings of their own experience when compared to their supervisors and peers rating of their ability. Following the literature on rating, we predicted an inverse relationship. That is, those with little experience would over-rate themselves and those with a high level of experience would under-rate themselves. We also predicted that the length of time a programmer had been commercially practicing pair programming would be the most consistent predictor of both self- and peer-ratings. We derived a delineation that we then ‘tried out’ in a pilot study with 14 participants (7 considered ‘novice’ and 7 ‘experienced’ according to our six-month benchmark). We analysed the conversations of these ‘novices’ and ‘experts’ both in general, and according to who was driving – in order to assess whether there appeared to be differences in behaviour between the programmers in each category. Using our delineation and the results from the initial study, we performed a set of four observational studies of commercial, experienced (according to our benchmark) pair programmers.

In our first analysis we employed ethnographic techniques in order to consider how pair programming is practically accomplished. We focused on the tools, artefacts and environment in which pairing took place in an attempt to understand how these aspects facilitated two activities key to pair programming: role management and the communication of technical information (both within and outside of the pair). We provided a description of the ecology in which pair programming takes place and highlighted a number of rich and subtle ways in which the pairs communicated. We also considered the extent to which the pair-programming environment facilitated or inhibited personalisation of each programmer’s workspace and the potential impact of this on their comfort and individual requirements.

We then began to focus on the pair themselves. We considered the content of a ‘typical’ pair-programming session in terms of key activities or ‘task types’. We then used the contribution of new verbal information to a task as a means of measuring the extent to which the pair ‘collaborated’ as opposed to ‘co-operated’. A pair were said to have co-operated on a task if they both contributed new information to it. We also considered co-operation between the driver and navigator roles and investigated whether or not one of these roles was more likely to contribute information to a particular type of task.
Given the results from our collaboration/co-operation analysis we then performed further verbal protocol analysis in order to ascertain how a programming pair might work together on the same sub-task. In particular we sought evidence of the existence of two possible realisations of the role of the navigator as suggested in experience reports in the literature. We called these realisations ‘the navigator as reviewer’ and ‘the navigator as foreman’. When acting as reviewer, we would expect the navigator to focus on correcting syntax and spelling errors. When acting as foreman, we would expect the navigator to focus on higher levels of abstraction and on the business domain. The results led us to suggest an alternative perspective on the driver and navigator roles, namely that the programming pair form a kind of cognitive ‘tag team’ in order to share the load of typing in code.

Finally, we used documented some of the behaviours that were in evidence across our studies. We produced a set of seven behaviours seen as unusual or special to experienced pair programmers. We also produced four behaviours, detailing behaviours we sought (following references to them in the literature) but did not find. We believe that these behaviours will be useful for existing pair programming practitioners (by way of fostering awareness of their practices) for potential pair programmers (by highlighting some of the additional issues associated with pairing) and for students of computer programming (by showing how pair programming is practically realised).

8.3. Findings
Based on the studies and analyses described in this thesis, we reached a number of conclusions about the successful realisation of collaborative software development. In this section we will further develop these conclusions and assess their implications for further studies, for the practitioners of collaborative programming, for the environments provided for pair programmers and for those teaching and learning programming in an academic environment. Finally we will consider their more general impact on our current understanding of collaborative software development.

8.3.1. Contributions
The main contributions of this work are as follows:
• It highlights key differences in pair programming behaviour between those less and more experienced at programming collaboratively. In particular, it demonstrates that a more experienced pair is able to change between driver and navigator roles in a fluid manner.

• It provides a detailed account of the manner in which pair programming is facilitated by the novel use of tools and the setting and environment in which it takes place.

• It provides an understanding, along with concrete examples, of the role of overhearing in pair programming.

• It highlights the need for more tailored tools for use when programming collaboratively in order to avoid such hazards as, for example, repetitive strain injury.

• It provides evidence that more experienced pairs work in a collaborative, rather than a co-operative, manner.

• It provides evidence that neither the driver nor the navigator focuses on a particular level of abstraction. Rather, it suggests that the proliferation of talk at an intermediate level of abstraction may be key to achieving some of the benefits attributed to programming in pairs.

• It presents an alternative model of pair programming, known as ‘the tag team’. It finds little evidence for the commonly occurring distinctions between the driver and the navigator role in pairing, rather considers the pair as more equal in status, sharing the overhead of typing by swapping roles.

8.3.2. Implications for future studies

Chapter Three concluded that after six months’ commercial pair programming, a programmer would be rated as ‘experienced’ by themselves and ‘able’ by their peers and supervisors. In addition, analysis of the behaviour of programmers of these different abilities confirmed the usefulness of this delineation by suggesting a different set of behaviours for each group. This provides a useful method for dividing participants in further studies of pair programmers. In
describing the environment in which pair programming took place in four different companies, Chapter Four provides a context for future studies of collaborative software development and an understanding of the commonalities and differences between a variety of pair programming projects.

**8.3.3. Implications for practitioners**

The six-month ‘probationary period’ in which commercial pair programmers are considered novice either by themselves and/or by their peers/supervisors may assist in decisions regarding pair composition. It provides a clear method of assigning programmers of suitable abilities to tasks. For example, a project may require an ‘expert-expert’ pair to work on a particularly difficult problem, or an ‘expert-novice’ pair where knowledge transfer and learning is particularly important. Beginning to define a set of differences in behaviour between novice and experienced programmers may also allow a practitioner to reflect on their performance and their progress according to this trajectory and identify deficiencies and training needs as well as providing a more measurable way of focusing on positive performance.

We also found that the verbalisations produced while programming collaboratively can be combined with the physical layout of a project to optimise peripheral awareness. That is, by providing optimum circumstances for selective overhearing by members of a project, help can be requested and provided, bottlenecks avoided, progress informally reported and tracked and fluid re-assignment of programmers to pairs achieved. This has implications for the physical design and layout of projects where pair programming will be taking place in order to augment the positive effects of overhearing but provide an environment that is not so aurally busy as to prove impossible to work in.

The pair programming behaviours defined in Chapter 7 may be helpful to both current and potential future practitioners of pair programming. For current pair programmers, they may be used as a tool to reflect more closely on their needs and practices. In addition, they may highlight some additional environmental considerations. For potential future practitioners thinking of trying pair programming, they document some of the additional decisions and behaviours that should be considered in implementing a pairing approach.
8.3.4. Implications for pair programming environments

In Chapter Four we highlighted the way resources designed for individuals are re-appropriated and augmented by pair programmers to facilitate their collaboration. Our findings implied that collaborating on programming tasks adds some very special requirements, in particular relating to the pair’s working environment and especially relating to role management and within- and extra-pair communication. While the innovative use of existing tools can be helpful in fulfilling these requirements, it suggests a requirement for more specialised facilitative tools. In particular, the novel way in which programmers make use of informal external representations could be encouraged and supported, and alternative methods of suggesting and effecting role change could be provided.

Most evident from these studies was the need for a tailorable or ‘personalisable’ environment for each programmer. Across all four studies the workspace set-up for the programming pair was never able to provide variation according to preferences on issues such as optimum screen height, desk height and position, mouse and keyboard preferences (e.g. Trackball mice, ergonomic keyboards) and font size. In fact, when the driver and navigator changed role, the environment was never modified accordingly. Not only does this violate guidelines regarding ergonomics and the avoidance of work-place injuries such as R.S.I., it may also have an impact on the comfort and therefore productivity of the programmers involved. In addition, pair programming as it is currently practiced may be seen as excluding programmers with particular special needs.

8.3.5. Implications for teaching and learning collaborative software development

The comparative study of ‘novice’ and ‘expert’ pairs highlighted a number of issues that are useful in helping us to begin to understand aspects of pair programming that have not previously been considered. A number of these issues are particularly relevant for the teaching and learning of collaborative software development.

Experts were seen to talk less (nearly 100 fewer interactions per hour) than novices. In particular, novice pair programmers were seen to ‘thrash’ between solutions. This implies that an important component in learning to pair program is learning and practicing the communication skills associated with negotiation. There are a number of ways in which such skills could be nurtured. In one example, students might be given non-programming tasks
whose primary focus is resolving differences in opinion. In another example, students may be taught some specific skills related to negotiation and conflict resolution. In a third example, students may be provided with a method of scaffolding their learning of this skill while working on programming tasks together. Finally, tutors in practical sessions might draw attention to, and assist in the resolution of, ‘thrashing’ incidents, thus nurturing skills in both meta-cognition (realisation that such an episode is occurring) and resolution (coming to a mutually agreeable solution).

Expert pair behaviours appear to stay very similar irrelevant of who is driving, whereas novices change behaviour dramatically then they change role. While it might prove difficult to ‘teach’ this seamless role change, it may nevertheless be useful to inform students collaborating on software tasks that this is the case, and to have students reflect on the extent to which they accomplished this in their programming sessions.

In addition to these specific implications, there are also more general collaborative skills that are not particular to programming. Some of these skills might be gained or better understood through the medium of pair programming.

8.3.6. Implications regarding our understanding of pair programming

This section outlines the impact of our findings on the current understanding of pair programming. They are divided into two main areas: those relating to each person in the pair, and those relating to role.

We found that pair programming is highly collaborative in nature. That is, that rather than dividing up the problem and tackling a part of it each, both people contribute key verbal information to almost all (93%) of the tasks undertaken. In particular, contributions are most evenly distributed between the two programmers when they are discussing ‘refactoring’ tasks. This implies that this type of highly cognitively taxing task requires both programmers to be fully engaged and actively contributing.

The same was true according to role. That is, both the driver and navigator contribute almost equally to each task. In fact, somewhat counter-intuitively, the driver contributes just slightly more, although this may simply be the effect of the driver commentating on what she is doing.
Some further investigations were made according to claims about the driver and navigator roles. The analysis described in Chapter Six found that the navigator neither acted as reviewer (correcting syntax and spelling) nor as foreman (working at a higher level of abstraction, and considering the appropriateness of the solution to the business problem). This is in direct contradiction to claims in the literature, which have been made and then repeatedly referred to without (as far as the author is aware) corresponding evidence.

These findings lead us to suggest that, rather than performing different functions, experienced programmers who are collaborating on a problem tend to work both on the same part of the problem and at the same level of abstraction at any given time. That is, the programming pair finds a kind of ‘synergy’ of working together, in synchrony, on each aspect of the task at hand. As such, it seems that the roles of ‘driver’ and ‘navigator’ are useful only in determining which programmer is typing (and therefore has an extra cognitive burden) at any particular time.

In fact, as the pair not only collaborate on each aspect of the task, but tend to do so at the same level of abstraction, we have suggested that the driver and navigator roles are a practical means of not only simply typing in code, but also of effecting a type of ‘cognitive tag team’ to help share the load of typing in the code. We suggest that in order to change role in the fluid manner necessary to not compromise productivity, the navigator and driver must be continually ‘in sync’ and that this is assisted by the proliferation of references made at the level of ‘chunks of code’ in their talk. In particular, we suggest that talking at this level may help to tame the complexity of thinking at different levels at the same time (which is one of the known and documented difficulties of computer programming).

The importance of talking (and perhaps thinking) at this intermediate level has implications on a number of levels. In one case, fostering working at this level may assist in easing the teaching and learning of programming. In another, representations focusing on or highlighting this level may be beneficial to programmers both in a commercial and academic environment. In a third, encouraging or requiring talk at this level may assist in solving complex programming problems. Finally, this finding may transfer or be useful across other domains involving complex problem solving in a whole range of arenas from mathematics to architecture.
8.4. Limitations

The over-arching limitations of the work presented in this thesis are in two main areas: methodological limitations and limitations relating to findings.

8.4.1. Methodological limitations

The mixture of qualitative and quantitative methods used in the studies described in this thesis might be considered somewhat unorthodox and the analysis of programmers at four different companies could be criticised by traditional ethnographers. However, the author believes that by harnessing the power of these different types of approach it becomes possible to provide a more rounded account, not only of the details of particular behaviours but also of the context within which they took place. In addition, the use of four different sites was used to verify findings by applying the same coding scheme across sites. This approach is in line with ‘cognitive ethnography’ (Ball & Ormerod, 2000). It should also be noted that the sample of companies was opportunistic in nature rather than statistically representative as the studies were of projects in companies that were not only ‘publicising’ their use of pair programming, but that were also happy to be studied.

In Chapter Four we highlighted the role of tools and setting in pair programming, stressing the importance of the environment in which collaborative software development takes place. However in Chapters Five and Six we focus entirely on the pairs talk rather than other forms of communication and engagement with the task. This was necessary for two reasons: first, the sensitive nature of the projects involved meant that video recordings and screen capture were not feasible. Second, the research issues covered in these chapters required the analysis of data at a level of detail which observation alone could not provide.

8.4.2. Limitations of findings

The primary limitation of the findings reported in this thesis is the extent to which they might generalise. While verification of findings has taken place via the consistent use of a single set of codings across four companies, the data collected was opportunistic in nature and therefore could not necessarily be said to be ‘representative’ of all programmers currently practising pair programming commercially.

Our analyses in Chapters Five and Six do not show statistically significant findings. While this may be considered a limitation, in fact, the main contribution of these two chapters is the
absence of findings related to phenomena suggested in the literature – particularly the lack of a significant difference in the levels of abstraction discussed by programmers when assuming the driver or navigator role. These findings regarding what pair programming roles were not lead us to make suggestions about what the nature of the driver and navigator roles might be. In particular, we outline a ‘cognitive tag team’ approach, however, further work is required to corroborate this suggestion (see Section 8.5.3 for some suggested studies).

8.5. Future work

There are seven main areas of future work that could be inspired or informed by this thesis: consideration of the effects of verbalisation on software development; comparative studies of agile and conventional programmers; further investigations into and refinements of the tag team concept; studies of intermediate level talk; the production of additional pair programming behaviours; studies regarding pair programming and education; and investigations into pair programming and meta-cognition. In this final sub-section, these are discussed in turn.

8.5.1. Considering the effect of verbalisation

It is possible that verbalisation may be key to the improvements associated with collaborative software development. In fact, there are suggestions that talking to oneself or even to an inanimate object may assist in solving programming problems (Portland Pattern Repository cited in Williams & Kessler, 2003). These suggestions are in line with domain and activity specific studies in other areas that suggest that self-explanation has a positive effect on understanding (Chi, de Leeuw, Chiu, & Lavancher, 1994).

Further work on verbalisation and pair programming could help to identify the optimum verbalisation conditions for learning and practicing programming. In particular, such studies could highlight the effects of different types of verbalisation on programming outcome, the programming process and affective state.

8.5.2. Comparing agile and conventional approaches

A useful area of research would be a set of comparative studies, contrasting agile development projects and practices with those of more conventional software development. Areas of focus might fruitfully include: type of tasks covered in an hour; amount and type of talk; amount of consultation (comparing talk outside of the pair with talk across programmers on conventional
projects); and physical layout of project space. The outcome of this work might be a road-map, mapping practices to other factors including problem type and complexity, project size, programmer ability, phase in the development lifecycle, type of problem domain, programming environments and languages. It might also highlight some overall ‘best practises’ and commonalities between the two approaches. Finally, such studies may also consider issues of transfer between solo and pair programming (and vice versa). For example, whether pair programming might imply a crisis of confidence when those used to the ‘comfort’ of pairing are required to program alone.

8.5.3. Investigating the ‘tag team’

In Section 6.5.3 we presented our interpretation of the driver and navigator roles as a ‘cognitive tag team’. In particular, we suggested that the navigator maintained a firm grasp of, and contributed equally to, the task at hand. We also suggested that the driver role might be more a symptom of sharing the load of typing than any other division of labour. As mentioned in Section 8.4.2, further work is required to confirm the notion of the ‘tag team’. Here we suggest two studies with this aim.

First, in order to check navigator understanding, a number of pair programming sessions could be interrupted randomly. During these intervals the navigator should be asked to explain both the current state of the solution and the forward plan. These explanations could then be cross-checked with the driver and the code itself.

In order to verify that role change assists in dividing up the work of typing, a further set of experiments could be undertaken. These would enforce a set of ‘single driver’ pair programming sessions, where one partner is forced to continually type. These sessions could be compared with other sessions, where the driving is more evenly shared. Thus one could ascertain perceived overload by the driver (how tired the driver feels) and navigator drift (how engaged the navigator remains), as well as considering any possible changes in the nature of their talk over time.

8.5.4. Investigating intermediate level talk

Central to the idea of the ‘cognitive tag team’ is the proliferation of intermediate level talk. In Section 6.5.4 we suggested that this level of utterances may ‘provide the glue’ that holds
together the upper and lower levels of abstraction. We also suggested that they could help keep the navigator up to speed by providing information at this missing level. It is possible that the proliferation of talk, and perhaps thought, at this level is the essential factor promoted by pair programming that gives rise to the reported increase in software quality. To investigate this further several studies are suggested.

First, studies of novice student pair programmers might take place. The conversations of the pairs might be recorded and analysed for intermediate level utterances. The ‘tag team’ concept would assume that these novice pair programmers talk less at an intermediate level than more experienced and successful pairs.

Second, commercial pair programmers of varying experience might be recorded performing their usual work. At the end of each session, the pair could be asked to rate the success of the session and the quality of code they have produced. The conversations of the pair might also be analysed according to the methodology described in Section 6.2 to ascertain the level of abstraction of their utterances. A comparison can then made between the session success and code quality ratings and the degree of intermediate level talk in the session. The ‘tag team’ concept would assume that intermediate level utterances are most common in the sessions rated as ‘high quality’ and successful.

Further studies might also take place, with the aim of ascertaining conditions that might encourage or optimize intermediate level talk. In one example, there may be a frequency of driver-navigator role change that encourages talk at this level, as the navigator feels the need to be able to take over driving ‘at any moment’. A variety of enforced role change frequencies could be experimented with to ascertain whether there is an optimum ‘driving time’ that maximises ‘PR’ utterances.

Alternatively, some form of intermediate level representational support could be provided in an attempt to foster ‘PR’ level talk. Of course, results could show that providing these representations had a negative effect on the level of ‘PR’ talk. Such findings might indicate that it is the very lack of representations in the Extreme Programming methodology that encourages compensatory intermediate talk. In particular, it is possible that the production of this compensatory talk has the pleasing side effect of improvements in software quality.
8.5.5. **Defining further pair programming behaviours**

Another further step, already mentioned in Chapter Seven is the identification and documentation of a full set of programming behaviours. A comprehensive set would be invaluable to the software development community, both in terms of a set of ‘best practices’ and ‘practices to avoid’ for current practitioners of collaborative software development, as a reflective tool by which to consider and evolve practices and as a decision support tool for those considering introducing pair programming. Such an endeavour could be informed both by this thesis, by suggestions in the existing literature, and by further studies.

8.5.6. **Informing programming education**

One direction in which the work in this thesis could evolve would be in its application to the academic arena. In particular, comparative studies of the behaviours of ‘successful’ professional pair programmers and students may identify key requirements and shortfalls and could yield a number of suggested additions to the programming curriculum. It may also suggest ways in which to scaffold student programmers’ learning of pair programming skills (especially those more subtle skills involved in role change, intermediate level talk and negotiation). Finally, it could imply changes to the physical environment in which programming is taught and learnt.

Another learning-related objective might be investigation into the features of a learning activity that can promote successful collaboration.

8.5.7. **Fostering meta-cognition**

One final burning question stemming from this thesis is whether collaborative software development fosters meta-cognition, reflection and retrospection and to what extent. That is, does talking to a programming partner make one more aware of one’s progress, capability and cognitive processes than talking to oneself or working alone. The results from such experiments may again have implications for both the teaching, learning and practicing of software development both academically and in the commercial world.

8.6. **Conclusion**

As discussed in Chapter 1, collaborative software development has many reported benefits, but until now little empirical work had investigated how these benefits were obtained. This
thesis aimed to begin to provide an account of how pair programming is practically realised, in particular by commercial programmers experienced in its use.

The findings reported upon the surprising role of tools and artefacts in assisting pair programming. It also found pair programming to be highly collaborative. Most surprisingly, Chapter 6 reported that the roles of ‘driver’ and ‘navigator’ did not appear to be as meaningful as is suggested in the literature, rather that they seem to simply be a convenient term to distinguish who bears the additional cognitive load of typing. We suggested an alternative perspective, which we named the cognitive ‘tag team’ and provided a set of initial desirable and undesirable pair programming behaviours.

However, while this work furthers our understanding of collaborative software development, and has both theoretical and practical implications for a number of areas, considerable additional studies are required before pair programming can be understood in its entirety.
9. Bibliography


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis, IN, USA: Addison Wesley.


Kerawalla, L., Pearce, D., Yuill, N., Luckin, R., & Harris, A. (2006). "I'm keeping those there, are you?" The role of a new user interface paradigm - Separate Control of Shared Space (SCOSS) - in the collaborative decision-making process. *Computers and Education*.


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


10. Appendix A – Experience questionnaires

A.1 Participant questionnaire

<table>
<thead>
<tr>
<th>Question</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>How much pair programming experience do you have?</td>
<td>□ A lot</td>
</tr>
<tr>
<td></td>
<td>□ Quite a bit</td>
</tr>
<tr>
<td></td>
<td>□ Not much</td>
</tr>
<tr>
<td></td>
<td>□ None</td>
</tr>
<tr>
<td>How much do you enjoy pair programming?</td>
<td>□ A lot</td>
</tr>
<tr>
<td></td>
<td>□ Quite a bit</td>
</tr>
<tr>
<td></td>
<td>□ Not much</td>
</tr>
<tr>
<td></td>
<td>□ Not at all</td>
</tr>
<tr>
<td>How long have you been in IT?</td>
<td>□ &gt; 5 years</td>
</tr>
<tr>
<td></td>
<td>□ 2-5 years</td>
</tr>
<tr>
<td></td>
<td>□ 1-2 years</td>
</tr>
<tr>
<td></td>
<td>□ 6-12 months</td>
</tr>
<tr>
<td></td>
<td>□ Less than 6 months</td>
</tr>
<tr>
<td>How long have you been programming?</td>
<td>□ &gt; 5 years</td>
</tr>
<tr>
<td></td>
<td>□ 2-5 years</td>
</tr>
<tr>
<td></td>
<td>□ 1-2 years</td>
</tr>
<tr>
<td></td>
<td>□ 6-12 months</td>
</tr>
<tr>
<td></td>
<td>□ Less than 6 months</td>
</tr>
<tr>
<td></td>
<td>□ Never</td>
</tr>
<tr>
<td>How long have you been programming in this language/environment?</td>
<td>□ &gt; 5 years</td>
</tr>
<tr>
<td></td>
<td>□ 2-5 years</td>
</tr>
<tr>
<td></td>
<td>□ 1-2 years</td>
</tr>
<tr>
<td></td>
<td>□ 6-12 months</td>
</tr>
<tr>
<td></td>
<td>□ Less than 6 months</td>
</tr>
<tr>
<td></td>
<td>□ Never</td>
</tr>
<tr>
<td>How long have you been pair programming?</td>
<td>□ &gt; 5 years</td>
</tr>
<tr>
<td></td>
<td>□ 2-5 years</td>
</tr>
<tr>
<td></td>
<td>□ 1-2 years</td>
</tr>
<tr>
<td></td>
<td>□ 6-12 months</td>
</tr>
<tr>
<td></td>
<td>□ Less than 6 months</td>
</tr>
<tr>
<td></td>
<td>□ Never</td>
</tr>
<tr>
<td>Do you have an IT relatedÉÉ.</td>
<td>□ Degree</td>
</tr>
<tr>
<td></td>
<td>□ A level</td>
</tr>
<tr>
<td></td>
<td>□ O level/ GCSE</td>
</tr>
<tr>
<td></td>
<td>□ Other: _____________</td>
</tr>
</tbody>
</table>
# A.2 Peer/supervisor questionnaire

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Team:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Team:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pairing rating</strong></td>
<td>☐ High</td>
<td>☐ Low</td>
</tr>
<tr>
<td></td>
<td>☐ Medium</td>
<td></td>
</tr>
<tr>
<td><strong>Communication assessment</strong></td>
<td>☐ High</td>
<td>☐ Medium</td>
</tr>
<tr>
<td></td>
<td>☐ Low</td>
<td></td>
</tr>
<tr>
<td><strong>Comfort in pairing assessment</strong></td>
<td>☐ High</td>
<td>☐ Medium</td>
</tr>
<tr>
<td></td>
<td>☐ Low</td>
<td></td>
</tr>
<tr>
<td><strong>Confidence assessment</strong></td>
<td>☐ High</td>
<td>☐ Medium</td>
</tr>
<tr>
<td></td>
<td>☐ Low</td>
<td></td>
</tr>
<tr>
<td><strong>Ability to compromise</strong></td>
<td>☐ High</td>
<td>☐ Medium</td>
</tr>
<tr>
<td></td>
<td>☐ Low</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B – Sub-tasks

B.1 Example sub-task decomposition diagram
B.2 Number of contributions of new information to each generic subtask type by observed session

<table>
<thead>
<tr>
<th>Session number</th>
<th>Generic subtask type</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>I</th>
<th>j</th>
<th>k</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>15</td>
<td>24</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>34</td>
<td>41</td>
<td>19</td>
<td>24</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>61</td>
<td>0</td>
<td>43</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>41</td>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>14</td>
<td>8</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>42</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>2</td>
<td>20</td>
<td>17</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>5</td>
<td>1</td>
<td>20</td>
<td>0</td>
<td>4</td>
<td>91</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>25</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>35</td>
<td>3</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>19</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>26</td>
<td>5</td>
<td>27</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>7</td>
<td>0</td>
<td>10</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>4</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>20</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>13</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>50</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>46</td>
<td>10</td>
<td>28</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>20</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>18</td>
<td>3</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>8</td>
<td>1</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>29</td>
<td>30</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>0</td>
<td>6</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>12</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>7</td>
<td>0</td>
<td>16</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>13</td>
<td>37</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>30</td>
<td>2</td>
<td>12</td>
<td>5</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>7</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>13</td>
<td>0</td>
<td>59</td>
<td>25</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>36</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>9</td>
<td>8</td>
<td>29</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### B.3 Number of subtasks of each type to which new information was contributed by only one participant

<table>
<thead>
<tr>
<th>Subtask type</th>
<th>Total subtasks of this type</th>
<th>Subtasks of this type with only one participant contributing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A AGREE STRATEGY</td>
<td>322</td>
<td>26</td>
</tr>
<tr>
<td>B CONFIGURE ENVIRONMENT</td>
<td>74</td>
<td>14</td>
</tr>
<tr>
<td>C TEST (write, run, debug)</td>
<td>359</td>
<td>29</td>
</tr>
<tr>
<td>D COMMENT CODE</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>E CORRESPOND</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>F BUILD, COMPILE, CHECK IN/OUT</td>
<td>161</td>
<td>15</td>
</tr>
<tr>
<td>G COMPREHENSION (INCL. CODE)</td>
<td>613</td>
<td>30</td>
</tr>
<tr>
<td>H REFACTOR</td>
<td>210</td>
<td>12</td>
</tr>
<tr>
<td>I WRITE NEW CODE</td>
<td>416</td>
<td>21</td>
</tr>
<tr>
<td>J DEBUG</td>
<td>233</td>
<td>15</td>
</tr>
<tr>
<td>K FIND/CHECK EXAMPLE (INCL. CODE)</td>
<td>266</td>
<td>20</td>
</tr>
<tr>
<td>L DISCUSS THE IDE</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
### B.4 Number of subtasks of each type to which new information was contributed by only one role

<table>
<thead>
<tr>
<th>Subtask type</th>
<th>Total subtasks of this type</th>
<th>Subtasks of this type with only one role contributing (driver or navigator)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A AGREE STRATEGY</td>
<td>322</td>
<td>27</td>
</tr>
<tr>
<td>B CONFIGURE ENVIRONMENT</td>
<td>74</td>
<td>14</td>
</tr>
<tr>
<td>C TEST (write, run, debug)</td>
<td>359</td>
<td>28</td>
</tr>
<tr>
<td>D COMMENT CODE</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>E CORRESPOND</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>F BUILD, COMPILE, CHECK IN/OUT</td>
<td>161</td>
<td>15</td>
</tr>
<tr>
<td>G COMPREHENSION (INCL. CODE)</td>
<td>613</td>
<td>31</td>
</tr>
<tr>
<td>H REFACTOR</td>
<td>210</td>
<td>10</td>
</tr>
<tr>
<td>I WRITE NEW CODE</td>
<td>416</td>
<td>22</td>
</tr>
<tr>
<td>J DEBUG</td>
<td>233</td>
<td>15</td>
</tr>
<tr>
<td>K FIND/CHECK EXAMPLE (INCL. CODE)</td>
<td>266</td>
<td>20</td>
</tr>
<tr>
<td>L DISCUSS THE IDE</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>