

## **Program for Workshop on Human and Social Factors of Software Engineering 2005 (HSSE 2005), May 16, St. Louis, Missouri, USA**

Papers marked in yellow have longer presentations about 30 min. All other papers have presentations about 10 min.

### **Session 1: Social processes in Software development / Software process improvement**

Focus: Acceptability, client-programmer communication, eXtreme Programming

1. S. Michelle Young Helen M. Edwards Sharon McDonald J. Barrie Thompson, Personality Characteristics in an XP Team: A Repertory Grid Study
2. Paul S Grisham and Dewayne E. Perry, Customer Relationships and Extreme Programming
3. Amy Law, Raylene Charron, Effects of Agile Practices on Social Factors
4. A. Günes. Koru, A. Ant Ozok, and Anthony F. Norcio, The Effect of Human Memory Organization on Code Reviews under Different Single and Pair Code Reviewing Scenarios

### **Session 2: Personality types and behavioural patterns in Software Engineering**

Focus: Behavioural patterns, role types in Software Engineering, personality characteristics

5. Georgine Beranek, Wolfgang Zuser, Thomas Grechenig, Functional Group Roles in Software Engineering Teams
6. Eve MacGregor, Yvonne Hsieh, Philippe Kruchten, Cultural Patterns in Software Process Mishaps: Incidents in Global Projects
7. Kevin C. Desouza, Yukika Awazu , Managing Radical Software Engineers: Between Order and Chaos
8. Ashraf Gaffar, Ahmed Seffah, John A. Van der Poll, HCI Pattern Semantics in XML: a Pragmatic Approach

### **Session 3: Configurations of Social networks in Software development**

Focus: Social network analysis, different pragmatics and lessons learnt in Software Engineering projects, community-based approaches

9. Helen Sharp and Hugh Robinson, Some Social Factors of Software Engineering: the maverick, community and technical practices
10. Yvonne Dittrich , Kari Rönkkö, Olle Lindeberg, Jeanette Erickson, Christina Hansson, Co-operative Method Development Revisited
11. Carol A. Wellington, Thomas Briggs, C. Dudley Girard, Examining Team Cohesion as an Effect of Software Engineering Methodology
12. Chintan Amrit, Coordination in Software Development: The problem of Task Allocation

# The Effect of Human Memory Organization on Code Reviews under Different Single and Pair Code Reviewing Scenarios

A. Günes Koru, A. Ant Ozok, and Anthony F. Norcio  
Department of Information Systems,  
University of Maryland, Baltimore County, UMBC  
Baltimore, MD, 21250  
{gkoru, ozok, norcio}@umbc.edu

## ABSTRACT

Human memory organization has been shown to be related to how programmers understand programs. In recent years, agile methods brought the emphasis back on human and social aspects of software engineering with a set of new principles and practices. One of them, pair programming has been shown to improve quality and reduce the development costs. In this position paper, we propose a controlled experiment to evaluate the effect of human memory organization through chunking on code reviews under different single and pair code reviewing scenarios.

## Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Code inspections and walkthroughs*; D.2.9 [Software Engineering]: Management—*Software quality assurance, Programming Teams, Productivity*

## 1. INTRODUCTION

Human memory organization was defined by Miller [4] as the “life-blood of the thought process”. According to the memory organization theory, the logically related information put into distinct units called *chunks* enable organization in memory. Such chunks can be easily understood by programmers.

In the psychology literature, chunking was defined by Miller as “the process of organizing the input into familiar units or chunks”. In an earlier study, Norcio and Kerst demonstrated that programmers can more easily understand and recall the programs that are written in chunks, preferably in small chunks [5]. In computer programming, loop structures, grouped input and output statements, well known algorithmic constructs (e.g. a recursive binary search, a recursive tree traversal), and subroutines are the examples of program chunks.

Agile software development has emphasized a set of practices related to human and social aspects of software engineering which

were sometimes overlooked by the process-oriented software development approaches [3]. For example, extreme programming suggested pair programming as one of its main principles [1]. In pair programming, source code is written, tested, inspected, and debugged by two programmers who sit together in front of one computer.

The concept of pair programming has an intuitive appeal because it is easily perceived that two programmers working on the same problem is better than one. In addition, from a project management perspective, a number of previous studies empirically investigated the effectiveness of pair programming on software quality, project schedule, etc. and obtained supporting results [6], [7]. However, the underlying factors affecting the efficiency of pair programming and the reasons behind its usefulness have not been studied intensively [2]. Findings obtained through empirical studies in this direction could help developers to better manage and exploit the potential benefits of pair programming.

When considered within the context of extreme programming, several questions naturally rise about chunking and pair programming. How does chunking affect the efficiency of pair programming? Do small chunks have the same benefits in pair programming as they have in single programming? Can we reduce or remove the need for pairs if programs consist of small and easy to understand chunks? Does pair programming make it possible to deal with larger chunks or unchunked programs? If so, pair programming can be utilized more on the unchunked programs and less on the programs that consist of small and easy to understand chunks.

Some software managers still perceive pair programming as an expensive development approach for many projects. The answers to the above questions supported by empirical evidence could allow developers and project managers to do a trade-off analysis and make judgments about their use of pair programming by also looking at the characteristics of their programs.

In this position paper, we report a series of experiments that we plan to conduct in order to understand the effect of chunking on code reviewing under different single and pair programming scenarios. We choose to work on code reviewing because we find it easy to quantify and assess the efficiency and outcomes of this activity. For example, the number of defects discovered at the end of the experiment and defect discovery time can be used as measures. In pair programming, code reviews usually happen in a continuous manner. However, for the sake of conducting controlled experiments, we plan to perform our experiments only for the code reviewing activities. Therefore, the subjects will only be given pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Human and Social Factors of Software Engineering (HSSE)* May 16, 2005, St. Louis, Missouri, USA.

Copyright 2005 ACM ISBN # 1-59593-120-1/05/05 ...\$5.00.

```

void affiche_string(int i, int j, char *data)
{
    int k;
    if (i+strlen(data) > 79) {
        for(k=i;k<80;k++) {
            mvaddch(j,k,data[k-i]);
        }
    }
    else {
        for(k=0;k<strlen(data);k++) {
            mvaddch(j,k+i,data[k]);
        }
    }
}

```

**Figure 1: Sample Chunked Program**

grams with their specifications and asked to find the defects.

In addition, we consider different programming scenarios because we observe that the efficiency of code reviews can be affected by a number of other factors. For example, the expertise level of the individuals will also have an effect on the code reviewing efficiency of pairs. Expert programmers can recognize and remember program chunks more easily compared to novices. Therefore, a team that consists of two expert programmers can be expected to perform better than a team of two novices. Another factor is the visibility of defects. Some defects are easy to catch whereas some others are elusive. Therefore, we plan to involve expertise and defect visibility as the other independent factors.

Similarly, when code reviews are considered, problem complexity, documentation, indentation, and commenting can be other related factors. However, in our initial study, we plan to keep them constant in order to reduce the complexity of our experiments and to make them feasible.

## 2. METHODOLOGY

A 4-factor, 24-subject, and *within-subject* design will be used in the experiment. The experimental groups will consist of paired and single groups of programmers.

### 2.1 Dependent and Independent Variables

The experimentation will mainly be focused on measuring the effect of chunking on defect detecting performance.

The first factor will be *chunking factor*, categorized as chunked program and unchunked program. The second factor will be *defect visibility* with three levels, low, medium, and high. All of the participants will receive both chunked and unchunked programs seeded with defects from all three visibility levels.

The chunked program code will consist of program statements that are clustered into logically related groups. The unchunked program codes will consist of sequential program statements where no clusters will be able to be identified. Figure 1 presents a chunked program sample. This program piece includes an if statement with two branches. Each branch includes a for loop. Figure 2 is an example of an unchunked program. This unchunked program only consists of sequential statements.

The defect visibility will be determined by the experimenters prior to the experimentation. An easy defect will be highly easily detectable, for example an incorrect boolean condition. The participants will have a low level of difficulty finding a defect with medium visibility. An off-by-one error made in changing an array index can be an example of this. The difficult defects will require high attention and expertise in order to be detected. Some subtle

```

static void history_search () {
    history_search_pos = where_history ();
    rl_history_search_len = rl_point;
    prev_line_found = (char *)NULL;
    if (rl_history_search_len >= 5) history_string_size =
        rl_history_search_len + 2;
    history_search_string = xrealloc (history_search_string, his-
        tory_string_size);
    history_search_string[0] = '^';
    strncpy (history_search_string + 1, rl_line_buffer, rl_point);
    history_search_string[rl_point + 1] = '\0';
}

```

**Figure 2: Sample Unchunked Program**

errors related to the program logic can be the examples of this category.

The third factor will be “pair versus single”. Therefore, in our experiments we will observe the effects of the other factors on both one-person and two-person teams.

The last factor will be programmer expertise. The programmer participants will be categorized as novice and expert. Our plan is that 12 participants will be expert programmers, and the other 12 will be novice programmers. The level of the expertise of the programmers will be determined as follows:

- The participants who have been doing computer programming as a full-time job in the computer language that the experiment is presented in will be categorized as expert programmers.
- The participants who have at least taken one class in the computer language the experiment is presented in and have used the language on an occasional basis will be categorized as novice programmers.

Considering the expertise, we will have five types of groups:

1. One novice programmer
2. One expert programmer
3. Two novice programmers
4. Two expert programmers
5. One novice and one expert programmers

Each of the 24 participants will be used in one of the above groups. There will be three replications in the study (one replication requires four expert and four novice participants). Each participant will fulfill a defect detection task by themselves, with a participant of their expertise, and with a participant who has a different expertise level than theirs. Participants will be distributed into each group by balancing the gender.

The dependent variables will be defect detecting performance and satisfaction. For performance, defect detection times in each group and detection accuracy will be measured. Accuracy will be measured in two types of programmer errors: failure to detect defects, and false positives. If a participant or a participant team fails to detect a certain defect or fails to identify a certain defect, this will be counted as failure to detect. If a participant or a participant team thought there was a defect at a certain location in the program where in reality there was no defect there, this will be counted as a false positive error. Because the participants will mark the defects

via pencil on paper, the number of errors will be determined via post-experiment visual inspection. Defect detection times will be measured via video recording and post-experiment time measurement.

At the end of each session, the participants will fill out a survey indicating the satisfaction with their defect detection procedure.

## 2.2 Procedure

The tasks will consist of detecting and correcting defects in pre-written programs. The programs will be in either C or Java language, and expert and novice user distinction will be based on the participants' knowledge of the selected programming language. For practicality purposes, the software code will be handed to the participant in paper format and they will be asked to mark the defects with pencil. Participants will be tested in a controlled experimental room. Performance time will be measured via cameras in the experimental room, and errors will be measured via post-experiment visual inspection. In the two-participant groups, participants will be allowed to interact with each other in any way they would like. In single-participant groups, the participants will be encouraged to think-aloud and explain their mental procedures during the code reviews. The participants will be video-recorded and the dialogues and think-aloud monologues of the participants will be later analyzed qualitatively to determine the mental procedures and between-participant interactions for defect detection. All experiments will be approved by the Institutional Review Board (IRB) of UMBC.

## 3. SUMMARY

The experiment will allow the researchers to empirically determine the advantages and possibly disadvantages of chunking from the perspective of code-review performance and satisfaction under different programming scenarios. These scenarios also include programmer expertise level and defect visibility level. This controlled experimental environment will allow the researchers to determine whether presenting chunked program code improves single and pair code reviewing performance and satisfaction.

One potential weakness of the study is the isolation of other potential factors that may affect code reviewing performance, such as problem complexity and documentation. Future studies may explore the effects of the other factors on code reviewing performance within the same context.

This experiment will reveal some patterns about the efficiency of the code reviewing process. The design of the experiment is also suited to collect some qualitative information from the participants which can invoke further research questions. Especially, the communication between the partners during defect finding activities will allow us to model the cognitive processes of defect discovery.

## 4. REFERENCES

- [1] Kent Beck. *Extreme Programming Explained - Embracing Change*. Addison-Wesley, 2000.
- [2] Alistair Cockburn and Laurie Williams. *The costs and benefits of pair programming*, pages 223–243. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, 2001.
- [3] Watts Humphrey. *Managing the Software Process*. Addison-Wesley, Reading, MA, 1989.
- [4] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.
- [5] Anthony F. Norcio and Stephen M. Kerst. Human memory organization for computer programs. *Journal of The American Society for Information Science*, 34(2):109–114, September 1983.
- [6] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):17–25, July/August 2000.
- [7] Laurie Williams, Charlie McDowell, Nachiappan Nagappan, Julian Fernald, and Linda Werner. Building pair programming knowledge through a family of experiments. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering, ISESE*. IEEE, 2003.