Programming Process Visualizer: A Proposal of the Tool for Students to Observe Their Programming Process

Yoshiaki Matsuzawa
Faculty of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku,
Hamamatsu
Shizuoka, Japan
matsuzawa@inf.shizuoka.ac.jp

Ken Okada
Faculty of Environmental
Information, Keio University
5322 Endo, Fujisawa
Kanagawa, Japan
turkey@crew.sfc.keio.ac.jp

Sanshiro Sakai
Faculty of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku,
Hamamatsu
Shizuoka, Japan
sakai@inf.shizuoka.ac.jp

ABSTRACT

We have developed a tool that enables learners to observe the process by which they program through visualization of data that are recorded in the source code editor. One purpose of the tool is to assist learners by using the Personal Software Process (PSP) to allow them to analyze the process by which they program by using the tool after completing a programming task. The proposed tool has functions for A) replaying a process using animation; B) automatic calculation of metrics; C) support for inputting subtasks; and D) process analysis report generation. An evaluation experiment was conducted with participants from the second-level introductory programming course at our university. The results were that 1) the accuracy of effort estimation dropped, although we clearly found that the reason for the drop was the difficulty of the second assignment; 2) according to a questionnaire, students reported both the effectiveness of the observation task and the effectiveness of the tool; and 3) there was large differences between students in terms of the description level of subtasks.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Design, Human Factors, Measurement

Keywords

Programming Education, Process, Visualize, PSP

1. INTRODUCTION

In recent research on programming education, improvements in either the programming language or the programming environment have been proposed in order to enable learners to focus not just on coding, but also on programming by thinking logically. However, the effects on programming education have been evaluated based on only the output from learners, that is, the rate of correct answers. The process by which the learners solve logical problems has been treated as a black box.

In actual practice, there are few applications of formal education to the programming process covering the topic of "How to solve problems". The theme of this research is to attempt to measure, analyze, and improve the programming process employed by beginners during actual programming education.

To measure and analyze the programming process, we use the Personal Software Process (PSP) by Humphrey [6] which is a framework for measuring and analyzing the personal programming process. In this paper we propose a tool for observing and recording the programming process that enables us to apply PSP in the classroom and enables learners to conduct PSP analysis by themselves.

2. RELATED WORK

Early adopters of PSP in the classroom first appeared in 2000 [9, 1]. However, one trial by Lisack et al. (2000) [9] resulted in failure to get learners to understand the value of PSP, as demonstrated by the questionnaire result that more than half of the students strongly disagreed with the question "I can see that the PSP topics we covered would be helpful for doing quality work in future jobs.". Another trial conducted by Abrahamsson et al. (2002) [1] found that learners encountered difficulty in improving the accuracy of estimating effort, and found that students complained about the high overhead of taking logs during programming tasks. Hence, adapting PSP for the classroom has not been particularly easy. A more workable implementation is needed in order for PSP to be adopted in the classroom.

The problem of logging overhead in programming has been discussed not only in terms of adaptation for educational purposes, but also in actual professional programming. Johnson et al. (2003) [8] summarized logging methods into three generations. The first generation is manual logging as introduced by the original introductory text for PSP [6]. PSP Studio and PSP Dashboard are pieces of software that run on the computer, and were classified as second-generation systems. They consist of a user interface for inputting the break-down of tasks, and for measuring the effort expended on each task by specifying the start and end time in a similar fashion to using a stopwatch. However, the act of starting and stopping the timer forces the programmer to context-switch between programming and logging. Third-generation tools employ an automatic recording approach for the work done on the computer. Johnson et al. (2003) proposed a system named Hackystat, and in parallel, Sillitti et al. (2003) proposed a similar system named PROM [15].

In the past decade, several improvements to the third-generation tools have been proposed. Hochstein et al. (2005) proposed an approach combining manual and automated measurement of effort [5]. Akinwale et al. (2006) proposed automatic defect log generation by analyzing compilation errors [2]. Yuan et al. (2005) proposed an environment consisting of an embedded IDE (Eclipse) with integrated second-generation tools [16].

Several tools have been proposed for supporting the use of PSP in the classroom. Liu et al. (2008) proposed PDAT which can collect and display data for teachers [10]. However, the aim of the study was to assist teachers in the management of programming courses, which differs from the purpose of the original PSP. Other tools for data collection have been proposed. For example Norris et al. (2008) [14] proposed ClockIt which is integrated into the BlueJ programming environment. A real-time monitoring system for supporting teachers is also available for practical use [3]. The purpose of the present study differs from that of the aforementioned tools that focus on data collection, with this study instead aiming to assist with the analysis process among students based on the PSP framework.

Murphy et al. (2009) proposed Ratina [13] which has user interfaces for both students and teachers. However, the interface for students is limited to displaying a list of the records and certain particular metrics, and does not support task analysis.

As described above, previous studies have been limited to collecting and displaying low-level data. Information about "What was done" is needed for process analysis, and cannot be generated from low-level records. There have been no proposals of tools for assisting students with the analysis process, and successful use of PSP in the classroom has not been reported.

3. TOOL DESIGN

3.1 Goals

Our goal is to develop a tool that enables learners to analyze their own programming process after performing the task. The tool needs to be designed so that it gains the acceptance of learners and allows analysis to be conducted with reasonable overhead in the classroom.

3.2 Architecture

The overall architecture of the tool, which includes both recording and visualization, is shown in Figure 1. This is a typical third-generation architecture in which loggers embedded in the development environment can record logs. The tool records logs to the local disk, which is different from other tools like Hackystat which record logs and send

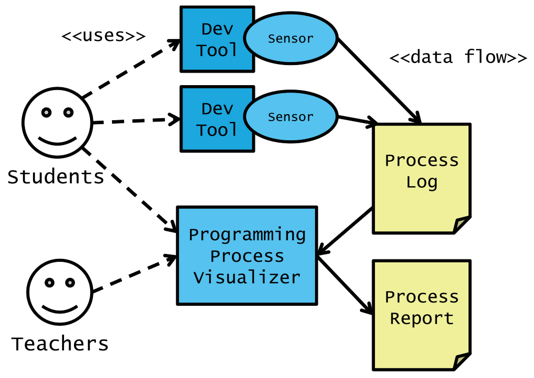


Figure 1: Architecture for logging and visualizing software

them to a server. Viewing the status in real time is not necessary for our goals.

We have currently developed two loggers, one for Eclipse and one for an editor we developed for an introductory programming course. The minimum unit of logging is editing of a single character. The Eclipse logger is available on the update site and is easy for beginners to install. Although the tool was written in Java, logging is supported for any text-based programing language.

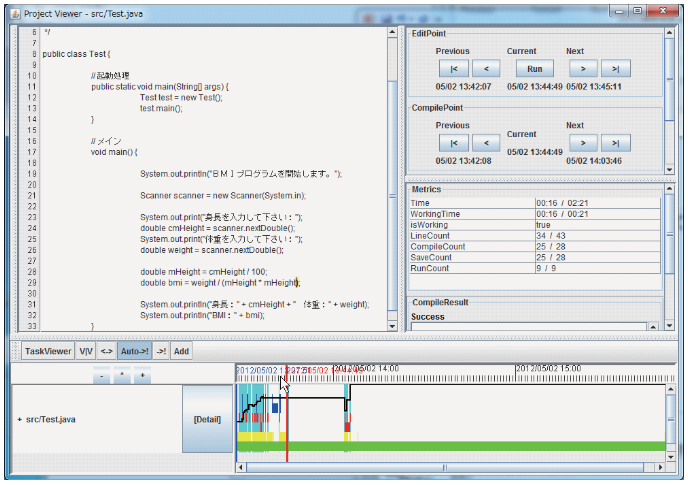
3.3 Programming Process Visualizer

Programming Process Visualizer (PPV) is software for visualizing the recorded programming logs. The software is composed of two components "Replaying Animation View" and "Task Management View".

3.3.1 Replaying Animation View

The main window of this software is named "Project View Window" and is shown in Figure 2. The window consists of three panes, "Source Code Pane (top left)", "Metrics View Pane (top right)", and "Timeline Pane (bottom)". "Source Code Pane" shows a replay of the programming process in an animated fashion. The user can move to or select any point they want to observe by manipulating the bar in the "Timeline Pane". As the point is moved, the differences are highlighted in the source code. "Metrics View Pane" shows automatically calculated metrics for the source code at that point. Currently, the metrics include number of lines, working time, number of compilations, number of runs, and compilation results. The working time is calculated by summing the time excluding periods of no user operation for 5 minutes or more (which corresponds to the "Active Time" in Hackystat). "Timeline Pane" shows user operations, events, and states with the x axis representing time. Events are colored by kind and plotted in time series as shown in Figure 3.

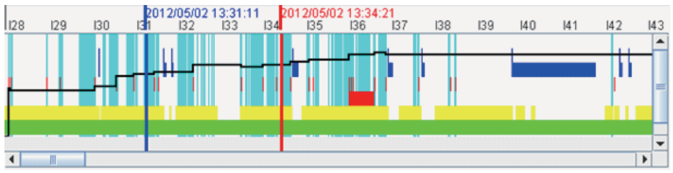
A bar area which indicates the history of the line of code is shown in the upper part of the timeline view. The area displays two bars. The red bar indicates the time for which the viewer is currently showing the code. The blue bar is described in the next section. The timeline pane has functions for zooming in and out, allowing users to analyze their programming process from both a descriptive and perspective view.



```
 */

public class Test {

        //起動処理
        public static void main(String[] args) {
                Test test = new Test();
                test.main();
        }

        // メイン
        void main() {

                System.out.println("ＢＭＩプログラムを開始します。");

                Scanner scanner = new Scanner(System.in);

                System.out.print("身長を入力して下さい：");
                double cmHeight = scanner.nextDouble();
                System.out.print("体重を入力して下さい：");
                double weight = scanner.nextDouble();

                double mHeight = cmHeight / 100;
                double bmi = weight / (mHeight * mHeight);

                System.out.println("身長：" + cmHeight + "　体重：" + weight);
                System.out.println("BMI：" + bmi);
        }
```

Metrics	
Time	00:16 / 02:21
WorkingTime	00:16 / 00:21
isWorking	true
LineCount	34 / 43
CompileCount	25 / 28
SaveCount	25 / 28
RunCount	9 / 9

Figure 2: Project View Window



<Legend>

light blue:	edit
blue:	run
blue line:	running
red:	compilation
red line:	compilation error
yellow:	viewing
green:	editor active

Figure 3: Timeline Pane

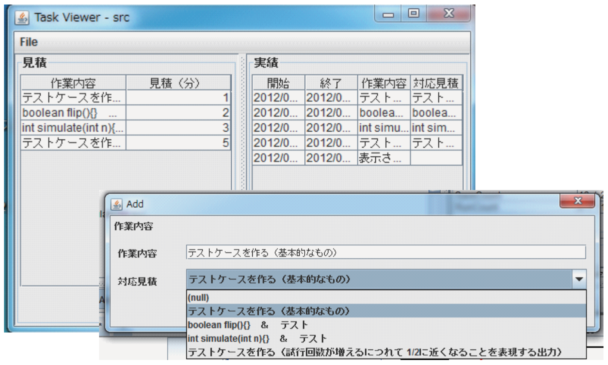
3.3.2 *Task Analysis Support*

Although it is a third-generation tool, the tool can record only low-level user operations. However, a "task level" description is needed in order for learners to analyze their programming process. The "task level" description represents the purpose of the low level user operations. Hence, students should attach task level tags to the timeline by observing the low-level user operations and interpreting what they were attempting to do by those operations.

PPV has functions that support "task level" analysis. The "Task Viewer" which is used for task management and the window for inputting the task level tags are shown in Figure 4. The task viewer is composed of two panes. The left pane is for managing estimates, and the right pane is for actual times.

The user can specify the estimated effort by inputting a task label and time in the estimation input window. The user uses the red and blue bars mentioned in the previous section for specifying the actual time. First, the user moves the red bar to the finishing time of the task and the blue bar to the starting time. The user then adds the actual time for the task by clicking the add button and inputting the task name. The user can then select the task name from the already entered effort estimates.

After the user has finished the task analysis, PPV can generate an analysis report written in HTML. The report contains the ratio of actual time to estimated time, lines of code, and the number of compilations and runs of each task. Users can view the report through a web browser, and teachers can publish reports to the Internet, allowing students to share them.



見積

作業内容	見積（分）
テストケースを作...	1
boolean flip(){}　...	2
int simulate(int n){...	3
テストケースを作...	5

実績

開始	終了	作業内容	対応見積
2012/0...	2012/0...	テスト...	テスト...
2012/0...	2012/0...	boolea...	boolea...
2012/0...	2012/0...	int simu...	int sim...
2012/0...	2012/0...	テスト...	テスト...
2012/0...	2012/0...	表示さ...	

作業内容

作業内容　テストケースを作る（基本的なもの）

対応見積　テストケースを作る（基本的なもの）

(null)
テストケースを作る（基本的なもの）
boolean flip(){}　&　テスト
int simulate(int n){}　&　テスト
テストケースを作る（試行回数が増えるにつれて 1/2に近くなることを表現する出力）

Figure 4: "Task Viewer" task management window

4. EMPIRICAL STUDY METHOD

4.1 Subjects

An experimental evaluation of the tool was conducted in the second introductory programming course at our university. Although the main theme of the course was object-oriented programing using Java, the students had experience in a variety of languages from the first introductory programming course, including Java, JavaScript, C, and Processing. We therefore conducted some review lessons for Java including basic control statements and creating methods with arguments in the first 5 weeks of the course. PSP activities using the proposed tool were conducted in the last two weeks of the review lessons.

4.2 Tasks

Task 1 and Task 2 shown below were used in this study. The aim of Task 1, covered in the fourth week of the course, was to create a method with arguments, and the aim of Task 2, covered in the fifth week, was to create a fractal image using recursion.

Task 1 Create a method to draw a matrix using the turtle graphics library. The interface of the method is given by the following prototype, and the matrix is drawn with the specified number of rows, columns, and size of each cell.

```
void drawMatrix(int row, int col, int size) {
}
```

Task 2 Create a program to draw a fractal image (you can select from Koch, Gasket, Tournament, or Star) using the turtle graphics library.

4.3 Procedure

The empirical study was conducted as follows.

1. Students created their effort estimates using PPV.
2. Students completed the task (creating a program).
3. Students conducted an analysis of their programming process using PPV.

In this study, use of PSP was limited to effort estimation and effectiveness analysis. Other applications, including size analysis and defect analysis, were not covered. A 30 min lecture on the concept and purpose of the PSP framework was given to students before the first task.

4.4 Method of Evaluating the Process Analysis

We used the actual effort time, accuracy of effort estimation, and qualitative and quantitative evaluation of the task description in order to evaluate the process analysis by students. The actual effort time was then provided by PPV and the accuracy was calculated using the method of Hayes [4], given by the following equation.

$$Accuracy = \frac{Estimated - Actual}{Estimated} \qquad (1)$$

4.5 Questionnaire

We conducted a questionnaire consisting of four questions for the students after the two analysis sessions were finished. The questions consisted of the following statements which the students were asked to respond to, based on the Likert scale (1. strongly disagree, 2. disagree, 3. neutral, 4. agree, 5. strongly agree).

Q1 The analysis of your project was useful for developing your programming ability.

Q2 The tool (PPV) was useful for conducting the analysis.

Q3 Your ability to accurately estimate effort improved through the process.

Q4 Seeing the reports of other students was useful for developing
your programming ability.

Additionally, we asked the students how long they took to perform the analysis.

5. RESULTS

5.1 Actual Effort and Accuracy of Estimates

The histogram in Figure 5 shows the distribution of actual effort (amount of time spent in minutes) for the class. The descriptive statistics are n = 30, min = 5, max=179, median = 31.30, mean = 49.43, and sd = 43.25 for Task 1, and n = 24, min = 9, max = 320, median = 57.50, mean = 95.58, and sd = 91.49 for Task 2. Both the mean and sd for Task 2 are higher than for Task1, which we interpreted as resulting from differences in the difficulties of the tasks.

Figure 6 shows a histogram of the distribution of effort estimation accuracy for the class. The descriptive statistics are n = 30, min=-537, max = 76, median = 18.5, mean = -27.73, and sd = 142.00 for Task 1 and n = 24, min = -433, max = 77, median = -60.00, mean = -83.29, and sd = 128.60 for Task 2. The mean accuracy for Task 2 was worse than Task 1, which means that the accuracy did not improve. We attribute this result to the effect of increasing task difficulty being stronger than the effect of learning (improvement by experience). Many students commented on the increasing difficulty in the questionnaire, which supports this interpretation.

5.2 Result of Task Analysis

The distribution of the number of tasks analyzed by students is shown in a boxplot (Figure 7). In the figure, T1 and T2 represent Task 1 and Task 2, Est represents estimated, and Act represents actual. For example, T1.Est represents the number of tasks declared in the estimation for Task 1.

The median is around 6, and there is no significant difference in the median between estimated and actual values. The ranges between the maximum and minimum and between the quartiles were wider for actual than for estimated. Our interpretation is that there are large individual differences in task division. Task division is difficult even for engineers [11], [12]. These results indicate the necessity of guidelines for task division.

Examples of student reports are shown in Figure 8 for evaluating the quality of task descriptions and task division. We chose four examples consisting of examples that were evaluated as excellent and poor for each task. Example 1-A is an example of an excellent report for Task 1 and 1-B is an example of a poor report. We can clearly see the incremental process in 1-A, since the first step was to draw a basic shape, the second step was to create a loop for the rows, and the third step was to create a loop for the columns. In comparison, the descriptions in 1-B were ambiguous or too abstract, with the main task focused on the "create method" task, which indicates failure of task division.

For Task 2, example 2-A shows an example of an excellent report in which the process is clearly described as drawing the motif first and then creating a recursive part. By comparison, in example 2-B the student struggled with the process of "create a program to move the turtle as I intended".

5.3 Student Reactions

The results of the questionnaire are summarized in Table 1. The average of the distribution for Q1 was "agree", and the percentage of "disagree" was 10%. These results can be evaluated as better than the results of other trials [9, 1].

The average of the distribution for Q2 was also "agree". The score was considered to be similar to that of Hackystat

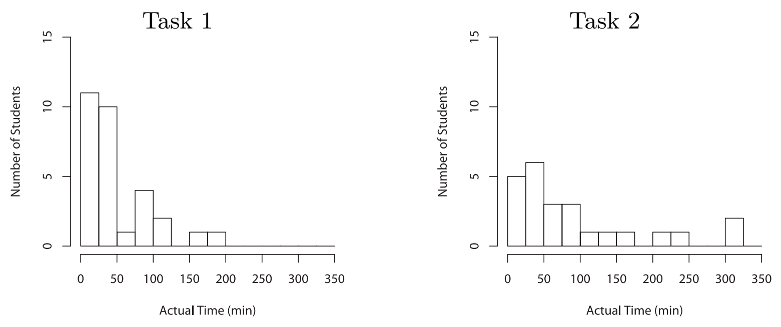


Figure 5: Actual time.

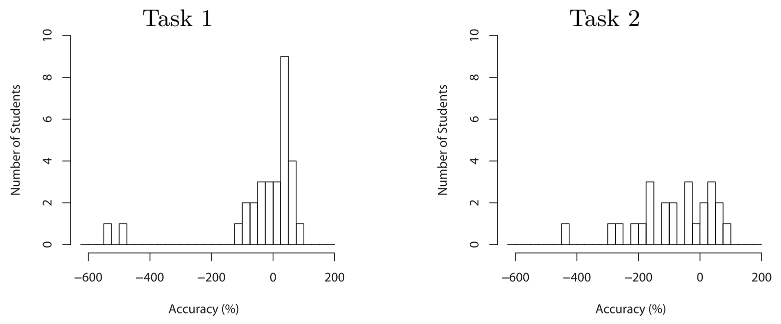


Figure 6: Accuracy for effort estimation.

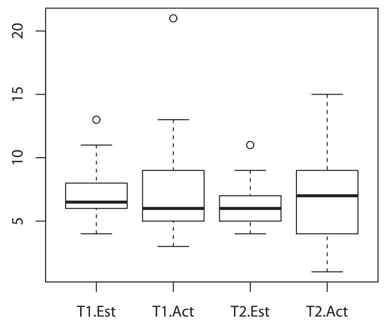


Figure 7: Number of tasks (Est=Estimated, Act=Actual)

[7]. However, the score that Hackystat obtained was only for showing recorded data, whereas the score for this tool included the analysis process. Hence, we consider that the students gave the tool a high score in terms of usefulness for analyzing their programming process. Some students commented that they enjoyed using the tool, and few students complained about the user interface of the tool.

Many students answered Q3 negatively in terms of their improvement at estimating effort. These students mentioned the difficulty of the task as a reason for their answers. Many students also agreed that sharing reports between peer students was effective according to the results of Q4. One of the students commented "It was useful to know the difficulty of the task by comparing the process reports", and another student commented "I could understand the programming process even if it was another student's".

According to the results of the questionnaire, the descriptive statistics for the effort (amount of time in minutes) put into the analysis are n=32, min=5.00, median=15.00, mean=17.03, max=40.00, and sd=8.69 for Task 1. Since the mean value of effort for the Task 1 was 49.43 (min), the ratio (analysis time / programming time) can be calculated to be 35%. Although this overhead for the analysis is not small, it is considered to be within the workable range for the university classroom.

6. CONCLUSIONS

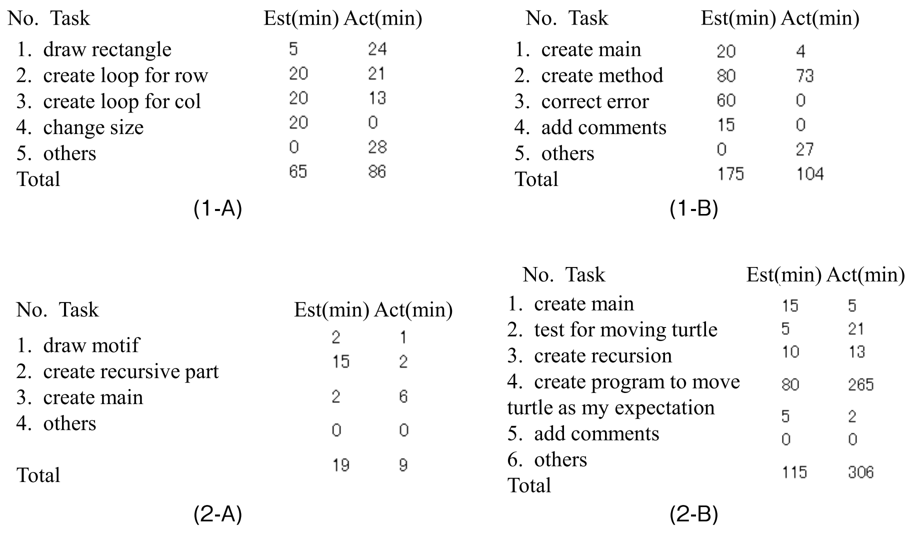
This paper proposed a tool named Programming Process Visualizer based on the PSP framework which enables students to analyze their programming process after completing a programming task. The results of an empirical study when this tool was employed in a university class showed that the students reacted positively to the tool.

7. REFERENCES

[1] P. Abrahamsson and K. Kautz. Personal Software Process: Classroom Experiences from Finland. In *Proceedings of the 7th International Conference on Software Quality (2002)*, pages 175–185, 2002.

[2] O. Akinwale and S. Dascalu. DuoTracker: tool support for software defect data collection and analysis. *International Conference on Software Engineering Advances*, 2006.

[3] A. Alammary, A. Carbone, and J. Sheard. Implementation of a Smart Lab for Teachers of Novice Programmers. *Australasian Computing Education Conference (ACE2012)*, pages 121–130, 2012.



No. Task	Est(min)	Act(min)
1. draw rectangle	5	24
2. create loop for row	20	21
3. create loop for col	20	13
4. change size	20	0
5. others	0	28
Total	65	86

(1-A)

No. Task	Est(min)	Act(min)
1. create main	20	4
2. create method	80	73
3. correct error	60	0
4. add comments	15	0
5. others	0	27
Total	175	104

(1-B)

No. Task	Est(min)	Act(min)
1. draw motif	2	1
2. create recursive part	15	2
3. create main	2	6
4. others	0	0
Total	19	9

(2-A)

No. Task	Est(min)	Act(min)
1. create main	15	5
2. test for moving turtle	5	21
3. create recursion	10	13
4. create program to move turtle as my expectation	80	265
5. add comments	5	2
6. others	0	0
Total	115	306

(2-B)

Figure 8: Examples of task descriptions

Table 1: Results of questionnaire

	1 Strongly Disagree	2 Disagree	3 Neutral	4 Agree	5 Strongly Agree
Q1. The analysis was useful	1 (3.2%)	3 (9.7%)	8 (25.8%)	18 (58.1%)	1 (3.2%)
Q2. The tool was useful	0 (0.0%)	4 (12.9%)	5 (16.1%)	19 (61.3%)	3 (9.7%)
Q3. Your accuracy improved	7 (21.9%)	8 (25.0%)	8 (25.0%)	9 (28.1%)	0 (0.0%)
Q4. Other students' reports were useful	0 (0.0%)	3 (9.4%)	5 (15.6%)	23 (71.9%)	1 (3.1%)

[4] W. Hayes. Process SM (PSP SM): An Empirical Study of the Impact of PSP on Individual Engineers. *Technical Report SEI, CMU*, 1997.

[5] L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, J. Carver, and G. R. Bergersen. Combining self-reported and automatic data to improve programming effort measurement. In *ACM SIGSOFT Software Engineering Notes*, volume 30, page 356. ACM, 2005.

[6] W. S. Humphrey. *Introduction to the Personal Software Process*. SEI Series in Software Engineering. Addison-Wesley Professional, 1997.

[7] P. Johnson, H. Kou, and J. Agustin. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. *ISESE'04.*, 2004.

[8] P. Johnson, H. Kou, J. Agustin, and C. Chan. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. *ICSE 2003*, pages 641–646, 2003.

[9] S. K. Lisack. The Personal Software Process in the classroom: student reactions (an experience report). *Software Engineering Education Training 2000*, pages 169–175, 2000.

[10] C.-h. Liu and S. Chen. Applying PSP to Support Teaching of Programming Courses. *Journal of Computers*, 19(3):55–66, 2008.

[11] W. Maalej. From work to word: How do software developers describe their work? *Software Repositories, 2009. MSR'09.*, pages 121–130, 2009.

[12] W. Maalej. Can development work describe itself? *Mining Software Repositories (MSR), 2010*, pages 191–200, 2010.

[13] C. Murphy, G. Kaiser, and K. Loveland. Retina: helping students and instructors based on observed programming activities. *SIGCSE '09*, pages 178–182, 2009.

[14] C. Norris, F. Barry, J. F. Jr, and K. Reid. ClockIt: collecting quantitative data on how beginning software developers really work. *SIGCSE '08*, pages 37–41, 2008.

[15] A. Sillitti, A. Janes, and G. Succi. Collecting, integrating and analyzing software metrics and personal software process data. In *Euromicro Conference 2003*, pages 336–342, 2003.

[16] X. Yuan, P. Vega, and H. Yu. A Personal Software Process Tool for Eclipse Environment. *Software Engineering Research and Practice, SERP 2005*, pages 717–723, 2005.