ADVANCES IN TECHNOLOGY-BASED EDUCATION: TOWARDS A KNOWLEDGE BASED SOCIETY

Proceedings of the II International Conference on Multimedia and Information & Communication Technologies in Education

m-ICTE2003
www.formatex.org/micte2003/micte2003.htm

Badajoz, Spain, December 3-6th 2003

Edited by

A.Méndez-Vilas
J.A.Mesa González
J.Mesa González

ISBN **Volume I**, (Pages 1-658): 84-96212-10-6
ISBN **Volume II**, (Pages 659-1335): 84-96212-11-4
ISBN **Volume III**, (Pages 1336-2026): 84-96212-12-2

Published by:
JUNTA DE EXTREMADURA,
Consejería de Educación, Ciencia y Tecnología (Badajoz, Spain), 2003
Printed in Spain

TEACHING LIGHT SOFTWARE DEVELOPMENT PROCESSES TO UNDERGRADUATE STUDENTS

R. O'CONNOR

School of Computing, Dublin City University, Dublin, IRELAND
E-mail: roconnor@computing.dcu.ie

Software engineering education is a challenging task. It must provide students with a knowledge of software development processes and the issues facing software developers in a commercial context However, computing students tend to focus on the programming aspects of developing software and frequently take a 'hacking approach' to developing program code, rather than considering the software development process. In order for students to appreciate the importance of the software development process, it first has to be taught. Such teaching should enable the definition of a software process that supports sound software engineering principles and which facilitates process learning. This paper presents the issues of SPI education, presents a series of SPI education experiments and discusses the results of these under the heading of: time distribution, time estimation, size estimation and defect analysis, with a view to Universities and colleges producing high quality computing graduates who have a sound knowledge of the importance of software process and SPI techniques.

1 Introduction

Software process improvement (SPI) is an essential topic in any software engineering curriculum. Some degree programs have a specific SPI course, while others intersperse it throughout the curriculum. Some SPI courses focus on a particular method (such as CMM, PSP or ISO 9000), while others discuss several methods. SPI also plays a part in other courses. For example, in project work the process the student follows should be one that undergoes continuous improvement, as it would in an industrial setting.

A key issue in SPI is to have a firm basis in terms of measurement. We have to collect measurement data and use the data to identify areas for improvement. After a process change, we must continue to measure to assess if the change made us achieve our objectives. SPI is inherently difficult to teach in a course as improvement mostly is concerned with long-term objectives. Thus to teach improvement, we must provide measurements to the students and allow they to use the data and collect new data, and thus evaluate if they have improved. SPI frameworks aimed at the individual such as PSP (Personal Software Process) [1] and PIPSI (Process for Improving Programming Skills in Industry) [2] are designed to help students and practitioners organise and plan their work, track their performance, manage software quality, and analyse and improve their individual process.

SPI education should be geared toward the perspective of the individuals students being trained for it to have the most effect. Establishing, a software engineering process, and improvement upon that process, depends on the individuals. If the individuals are disciplined, and adequately trained in the principles of software engineering, the chances are good that the software process will be successfully defined, implemented, and improved. This, in-turn, increases the probability of achieving the aim of an on time, within budget, error-free software product.

SPI frameworks aimed at the individual such as PSP (Personal Software Process) and PIPSI (Process for Improving Programming Skills in Industry) are designed to help students and practitioners organise and plan their work, track their performance, manage software quality, and analyse and improve their individual process. While many Universities in the USA teach SPI and SPI-related courses at the individual level using the PSP, the take up in Europe has been substantially smaller. This paper discusses the issues associated with teaching SPI at the individual level and presents the results of a series of teaching experiments, using a European developed PSP derivative, known as PIPSI.

2 Personal Software Process

The PSP [1] was developed to address the deficiencies of the CMM at the individual level. It is based on a set of key concepts:

- The process: each developer should follow a defined process
- The measures: each developer should monitor his or her performance by using a set of measures. The same developer collects the data and analyses it.
- Estimation and planning. Each developer should use adapted techniques to estimate the duration of his project, plan it, monitor its advancement and compare with the plan.

- Quality. Each developer should use specific techniques to improve the quality of her project. The goal is zero defect software. Defects should be eliminated as soon as possible, possibly before the first compile.

The PSP consists of ten programming assignments and four reports. Students start with a simple process that includes estimating and recording their effort for different process phases. As students progress through the assignments, they learn about, and use, more detailed and mature processes. The PSP relies heavily on data collection, mathematical estimation techniques and formal reviews (design and code), and the use of analysis to improve process effectiveness.

The results from applying the PSP are contradictory. In an academic setting a major study has shown that developers reduce the number of defects they leave in their programs while not changing their productivity [3]. In an industrial setting other studies have shown that estimation accuracy, and the quality of the product both improve [4]. So in principle, the PSP can be used to teach good software engineering practices and instil a disciplined approach. Nevertheless, many educators have experiences problems implementing PSP in the curriculum [5,6]:

- Students do not like it. They perceive the PSP as tedious to use and complain that it takes away from their 'real work'. Especially for small exercises (< 100 LOC) it is very difficult to convince students to use it.
- Data collection becomes very difficult in frequent edit-compile-debug cycles.
- Manual data collection becomes infeasible when class sizes exceed 50.
- If a PSP course has to be included into the curriculum, it typically has to replace existing courses.
- PSP requires basic programming knowledge, i.e. it has to be aimed at students who have a 'flawed' process. Wouldn't it be better to teach them a disciplined process from the very beginning?

3 Process for Improving Programming Skills in Industry (PIPSI)

The PIPSI (Process for Improving Programming Skills in Industry) project [2] is an ESSI funded project which aims to provide a process improvement framework for use by individual software engineers working in European SME's. The focus of the project is on improving individual software engineering skills thus generating bottom-up improvement. The PIPSI approach, whose aim is to present the techniques in a way that makes them more attractive and more easily used in small and medium-sized organisations and development teams. The focus of the PIPSI is on bottom-up process improvement by: 1) defining a personal process, 2) personal project management and 3) personal quality management.

The entire model is buttressed and controlled through the use of measurement. By collecting data on their own performance, students learn about how they develop software. The measures help them understand the fundamental relationship between size and effort and, through this understanding, enable them to improve their estimating abilities. Furthermore, by gathering data on their defect rates they witness how employing practices such as personal code reviews and the use of checklists will allow them to produce higher-quality program code. The measures provide information on performance, information can then lead to process improvement and process improvement can lead to the production of better quality software on time. Finally collecting performance data on an ongoing basis moves students from defining their own development process, through managing it to optimising it.

Through PIPSI training, students complete programming tasks on which they collect increasing quantities of data. Early exercises capture effort measures. Subsequent exercises gather size data whilst the concluding exercises capture defect and quality measures. Students can now develop more accurate and predictable delivery estimates. The final element of PIPSI is that of personal quality management. As students complete PIPSI program exercises, they collect data on the defects injected into those programs.

This process illustrates in which development phases they inject and remove defects. Furthermore, the defects are categorised by type, thus allowing a causal analysis to be performed which can then lead to defect prevention. PIPSI focuses on proven quality control mechanisms such as design and code reviews which enable developers to remove defects earlier in the development process. This achieves the twin objectives of removing defects at the front end of the development cycle where they are cheaper and easier to fix and, as a corollary, means testing time is more focused as fewer defects are escaping into test.

4 Teaching Experiment

To assist with validation of the PIPSI approach and its subsequent development a series of validation exercises [7] were undertaken in four European countries in an industrial context and also in an academic environment [8]. This paper focuses on the trial subsequent application of PIPSI practices to a group of final year computing students in Dublin City University.

The trial group consisted of 42 final year computing students, with PIPSI classes being held over a 3 week period in the academic year 2002-03. Students undertook 5 programming exercises and were required to collect successively more detailed data by following the PIPSI approach previously outlined. The students initially gathered effort measures and then progressed to personal project management by relating effort to task size and finally focusing on quality management through understanding defects. In section 5 we present and analyse the students data collected during the experiment. This is analysed under the heading of: time estimation, size estimation and defect analysis.

From the first exercise, students were required to keep track of the time they have taken to complete a particular programming task. A simple development process of "Pre-Build", "Build" and "Post-Build" is followed in all exercises, and the study group recorded their time in each of these areas for programs 1 to 5. One of the teaching objectives was to convince the group of the need to spend more time in the earlier phases of development such as understanding requirements, detailed design and the use of code reviews. Previous studies have shown how spending a greater proportion of time in the earlier life-cycle phases significantly reduces the amount of time required for testing as the product can be built 'right first time' and less rework and repair is required in test.

5 Data Analysis

The exercises used for the trials were deliberately short in order to allow for the maximum amount of data to be collected during the training period. Because the exercises were quite small many of the estimating errors are quite large. Other researchers have also found that small tasks can generate significant percentage errors. However, over time, when the disciplines have been applied to much larger tasks and sufficient historical data has been gathered, then the error percentages can be reduced. Figure 1 shows the time estimation error for the study group for programs 1 to 5.

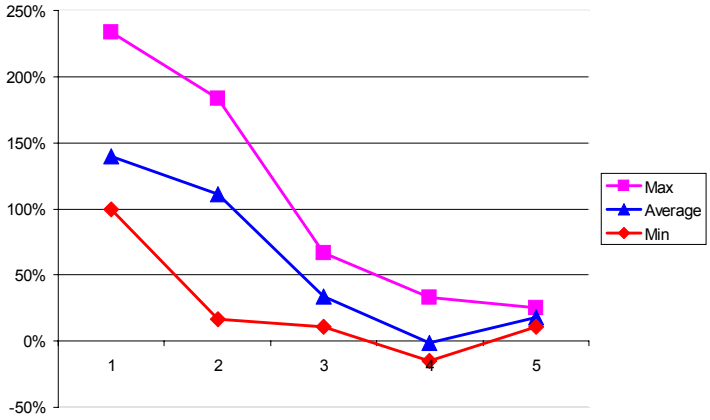

Figure 1. Group Time Estimation Accuracy

Generally students found the same level of difficulty with each of the programs. However, their estimates varied considerably in the early stages of PIPSI training, with the largest underestimate being 233% and the smallest underestimate being 100% for programming exercise 1. However, with some basic understanding of their over estimation data and more careful use of estimation techniques, the estimation error steadily fell over programming exercises 2 to 5. In the case of program 4, the students were generally becoming both more confident in their estimation and less conservative. For programming exercise 4, the best student overestimated the time required by 15%, the worst student underestimated by 33%, whilst the average estimate was out by just 1%. However, some element of the sudden change to overestimation maybe due to the description of exercise 4, which student subsequentially commented on as being verbose and "sounding harder than it was". It should also be noted that in programs 3, 4 and 5 size and productivity measures are used to derive time estimates so if the student collects subsequent figures this would indicate whether the improvement in time estimating is due to this approach.

Size estimation data was collected for programming exercises 3 to 5 (see figure 2). Initially the students were underestimating the size (LOC) of the programs (3) by 30%, however, this fell to an average underestimate of 11% by program 5. There were however some outlying figures, with the worst underestimate being 50% and the best overestimate just 3%. However, none of the participants overestimated the size of the programs. Obviously no firm conclusions can be drawn in this regard, but this should be monitored in future programs. It is also worth comparing actual program size with actual development time, to see the relationship between the size of a program and the time taken to develop it.

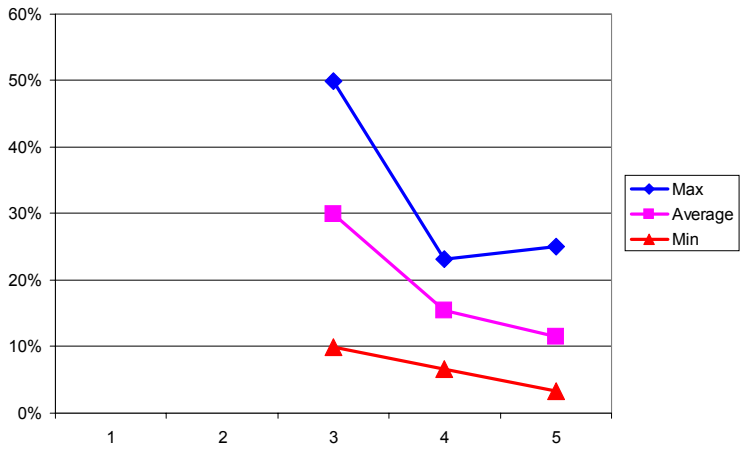

Figure 2. Group Size Estimation Accuracy

Defect data was collected for programming exercises 4 and 5, with students conducting a full code review prior to compilation of their programs, thus illustrating the benefits of the code review process, as opposed to the traditional student usage of the compiler to catch syntax and other basic errors. In hindsight it would have been more useful to either collect defect data on earlier programming exercises, or only conduct the code review process for exercise 5, as this would have allowed for a comparison of defect data before and after introducing the code review process.

As the programming exercises were small, it is not valid to directly analyse defects in the traditional manner, such as; defects density rates, defect injection rates, etc. However, student perception the code review process is very interesting. When initially introduced, students were unwilling to follow a pre-compile review, as they considered a primary use of the compiler was to highlight errors to the programmer. Further, may students found it counter-intuitive to follow a code review process, without using the compiler.

Whilst students did agree that the code review process highlighted many defects, in particular syntax errors, many claimed to be aware of regularly making similar syntax errors in the past. Thus it is not clear that if students were aware of defects they commonly injected into their programs, that they took any action to avoid such defect injection. A primary aim of the defect measurement stage of PIPSI is to make programmers aware of the defects they inject, so they may take steps to avoid injecting such defects, by modifying programming style / habits.

6 Conclusions

Whilst the data we have presented does not provide conclusive proof of the benefits of using PIPSI there are many encouraging signs which are inline with previous PIPSI studies using a smaller number of participants [7,8]. Time estimation accuracy has shown steady improvement over the duration of the study, as has size estimation. However, it is worth noting that short exercises (such as those used in this study) can generate quite large estimating errors, as participants tend to overcompensate for previous errors. Also, small tasks will always produce significant percentage errors, e.g. a 2-minute underestimate in a 20-minute task is a 10% error. It is expected that when applied to larger tasks and when size and productivity data are introduced that time estimating error will be reduced.

The data gathered by individual should act as a momentum to them to continue to use and monitor the PIPSI approaches to determine how they work for them in the longer-term. There were also some qualitative benefits from the study. Many of the students commented on how they had not thought about programming in this way prior to embarking on the study. A number expressed how previously, they were unaware of the proportions of time they were spending in the various development phases and furthermore had never taken product size into account.

There exists many different approaches for working with processes and process improvement. We are currently exploring the idea of introducing Extreme Programming (XP) [9] into a software engineering teaching module. There has been early experiments in comparing XP-like pair programming with individual programming based on PSP practices [10]. Although these results to date are not conclusive, they do indicate the merit of further study.

In the future we intend to carry out some formal academic studies of pair programming using the same exercises as those used in the PIPSI trials. This will enable me to make some comparisons between the

effectiveness of the two approaches. Furthermore, a longitudinal study is intended to ascertain if students still use pair programming and subsequently XP on a regular basis after teaching.

References

1. Humphrey, W. A, Discipline for Software Engineering, Addison Wesley, 1995.
2. IPSSI Project home page, available at http://www.computing.dcu.ie/research/ipssi
3. Hayes, W. and Over, J. W. Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers, Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, 1997.
4. Ferguson, P. et al, Results of Applying the Personal Software Process, IEEE Computer, May 1997.
5. Borstler, J., Should we teach the PSP for its own sake?, Position paper at Workshop on Teaching PSP and TSP in Universities, 14th Int. Conference on Software Engineering Education, USA, 2001.
6. Olofsson, S., Evaluation of the PSP in Undergraduate Education, Technical Report UMNAD 272.99, Umea University, 1999.
7. O'Connor, R., Duncan, H., Coleman, G., Morisio, M., McGowan, C., Mercier, C. and Wang, Y., Improving Professional Software Skills in Industry - A Training Experiment, Technical Report CA-0201, Dublin City University, 2001.
8. O'Connor, Coleman, G. and Morisio, M., Software Process Improvement Education - A European Experiment, Proceedings of 6th Annual Conference on Innovation and Technology in Computer Science Education, England, 2001.
9. Beck, K., Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.
10. Nawrocki, J and Wojciechowski, A., Experimental Evaluation of Pair Programming, Proceedings of 12th European Software Control and Metrics conference, London, April 200