

Human-Oriented Improvement in the Software Process^{*}

Khalid Sherdil
Nazim H. Madhavji

School of Computer Science, McGill University
3480 University Street, Montreal
Quebec, Canada H3A 2A7

Phone: (514) 398-3740
Emails: madhavji@opus.cs.mcgill.ca
sher@lums.edu.pk

Abstract

By doing any task repeatedly, individuals can usually improve continuously due to the experience gained (called autonomous first-order learning). In addition, they can improve due to the injection of software development technology by the organization (called second-order learning). Organizations have studied such learning curves to make decisions regarding cost estimation and budgeting, production and labor scheduling, product pricing, etc. Such progress behavior was studied in a laboratory setting in an experiment involving a sample of 12 student software developers, who completed one small-sized project every week for ten weeks. A within-subject, repeated-measure, time-series quasi-experimental design was used as the research method. This also included the Goal/Question/Metric (GQM) paradigm with some additional validation techniques from Social Sciences/MIS/Software Engineering. Statistical tests showed that on average, progress takes place at a rate of about 20%, with technology injection (i.e., second-order learning) amounting to 13% improvement over autonomous learning alone. Such a distinction is useful for making personal decisions in software development and managerial decisions regarding training programs and making engineering technology changes. The study was replicated, twice, with samples of size 30 and 12. The average progress rate for the 54 subjects (in the three studies) was 18.51%.

1 Introduction

The role of human agents in the software process is widely recognized to be a critical one [1]. An important aspect of human involvement in the process is to find out by how much an individual improves in the process over a period of time. For example, by how much one improves in the defect quality of software, rapidity with which software is developed, estimation of the size and defect quality of the system to be built or enhanced, amount of reuse carried out while developing software, etc. By knowing the improvement capabilities of humans, it adds to our knowledge of software process performance or enactment; it enables us to improve our predictions about the quality, cost and deliverability of the software that would

^{*} This research is, in part, supported by NSERC, Canada.

be built; and it enables us to search for ways to provide improved tool and technology support in the software process.

It has been known in other disciplines, through studies, that an individual can expect continuous improvement in productivity as a consequence of a growing stock of knowledge and experience gained by repeatedly doing the same task [5]. Organizations have used this type of progress behavior in making concrete decisions regarding cost estimating and budgeting, production and labor scheduling, product pricing, etc. [4][16]. While considerable research on this topic has been carried out in industrial and manufacturing sectors [33], we found little such emphasis in the business of software development. Consequently, the lack of such knowledge in the field of software process may be hampering us in making software processes effective.

Now, the improvement in performance, typically for some production activity, is represented in a mathematical form, called a progress function. In simple terms, a progress function represents the percentage decline in cost or labor requirement as the cumulative output increases by one unit [16][18], as shown in Figure 1.

Progress functions differ from the widely used term *learning curves* because the former also incorporate a *second-order* learning mechanism [12]. Whereas first-order learning (also referred to as *autonomous* learning) is the improvement due to the experience which a person gains by repeatedly doing the same task, second-order learning (also known as *induced* learning) is due to the technologies injected by the organization [2][23]. Although the distinction between these two is often blurred [23], our objective is to analyze them separately in our work. Such a distinction is useful for making decisions regarding initiating formal training programs and making engineering technology changes [2]. In the software process, for example, improvements due to technology changes can focus on those aspects where personal excellence is saturated or is at a plateau.

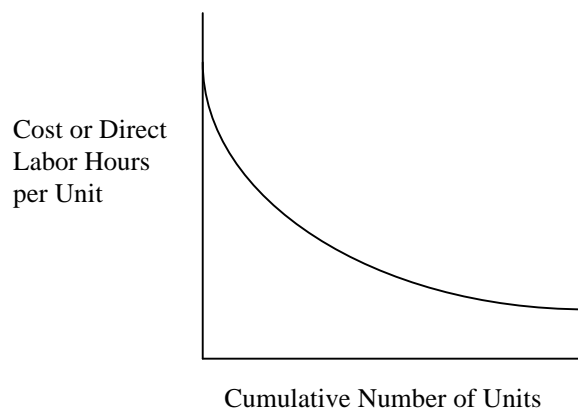


Fig. 3. A Progress Function. As the cumulative number of units increases (X-axis), the cost of production per unit decreases (Y-axis). Or equivalently, as time progresses, the labor time needed to produce one unit decreases.

It is important to note that second-order learning should not be confused with the term *Organizational Learning* [24]. The latter refers to aggregate learning by all the employees of a firm due to team effort. In this paper, we are not analyzing organizational learning. Rather, our objective is only to analyze individual learning, which is composed of first-order and second-order components. Obviously, such personal improvement eventually translates into organizational improvement.

In order to study individual progress functions, we have carried out a laboratory experimental study. An integral part of this experiment is the Personal Software Process, PSP, designed by Humphrey [17].

In this experiment, we deployed 12 student software developers working on 10 small programming projects, at a rate of one project per week (January to April, 1994). For each project, they kept track of measurements, such as the details of every defect found, lines of code (new, reused, modified), time spent on each development phase, etc. For consistency, all software developers used the same standards throughout the experiment, such as the same programming language, the same coding standards, the same personal process, the same data collection techniques, the same defect type standards, etc. The students had voluntarily enrolled in the PSP course taught at McGill University. They were all full-time students, and were enrolled in other courses offered during the semester as well. They spent 3 hours per week on attending two PSP lectures, and on average, four and a half hours per week working on the software projects. The motivation for the students to collect comprehensive and accurate data lay in the fact that they were graded on the quality of their data. They underwent an intensive program through which they were taught various software development technologies, as second-order learning. This training program introduced one new technology every week, such as size estimation methods, design reviews, code inspection, data collection and analysis, etc.

For statistical validation and reliability, the study was replicated twice. Samples of size 30 (September to December, 1994) and 12 (September to December, 1995) were used. Hence the total number of subjects in the three studies was 54.

We had two objectives in our study. The *first objective* was to measure the rate of progress, through five specific variables: (i) programmer-productivity, (ii) defect-rate, (iii) error in estimating size of project, (iv) error in estimating time for implementing the project, and (v) error in estimating self-productivity. The *second objective* was to determine the extent to which the injection of technologies helped in personal improvement. Since injection of technology corresponds to second-order learning, the second objective can be written in the form of a *hypothesis* as:

Does second-order learning contribute significantly to progress, compared to the first-order learning alone?

That is, is it worth while to invest in injecting new technology, or would the same amount of progress automatically occur by first-order learning alone (without technology injection)? The two types of technologies which we chose for injection in this experiment were: (a) methods for estimating size, and (b) code reviews.

It is well-known in the software engineering practice that the five variables dealing with productivity, defects and estimation are important. Similarly, the two technology types dealing with size estimation and code reviews are also useful. Thus, the study undertaken to investigate progress functions has wide significance in the field of software process.

Our experiment design is the hybrid of the time-series quasi-experimental design with repeated measure within-subject design [20][21]. The measurement technique used followed the guidelines of Basili's GQM [6], which is widely used as a metaprocess to conduct measurement experiments.

Our investigation has several original aspects to it which have not been dealt with before. As mentioned earlier, while considerable research on progress functions has been carried out in industrial and manufacturing sectors, little emphasis has been paid in the software process field. Furthermore, besides measuring the progress in productivity, our work involves measuring the progress in product quality and personal skills, which were not considered in other studies. Finally, as far as we know in the software process field, this is the first attempt to study the distinguishing effects of first-order and second-order learning.

Section 2 discusses related work on the subject of learning curves. Section 3 gives the details of PSP. Section 4 describes our experiment model. A description of the research method is given in section 5, while data analysis is done in section 6. This is followed by some observations in section 7 and conclusion in section 8.

2 Related Work

In the late nineteenth century, industrial expansion in the USA was accompanied by growing efforts to control management processes using empirical methods [12]. At that time, there were two prevailing theories: economic and managerial. The economic theory focused on the equipment and other capital goods for achieving a greater firm productivity. The managerial theory focused on static cost functions that were insensitive to time and experience. The progress function was thus a major discovery because it suggested that the efficiency of industrial process was dynamic and changing. [12]

Between 1900 and 1930, the use of progress functions was applied to the domain of aircraft manufacturing while during the second world war, it was applied to the domain of ship-building [25][12]. Contractors searched for ways to predict cost and time requirements for the construction of ships and aircraft [33]. Hence, the importance of progress functions grew and by now they have been applied to economic, engineering and managerial fields, and there exists over 60-

years of literature on this topic. These fields include electronics, machine tools, paper-making, steel, apparel, automobiles and others [12].

Wright, an engineer and administrator, was the first to report the phenomenon of learning curves, in 1936 [12][33]. He observed that as the quantity of units manufactured doubles, the direct labor time taken to produce each unit decreases at a uniform rate depending on the manufacturing process being observed. Subsequent studies often substituted direct cost or cumulative output for labor time [16].

Wright used the log-linear model, which has been followed in most studies. This model can be represented by

$$y = Kx^n$$

where K is the input cost for the first unit.

The progress ratio, p , is defined by

$$p = 1 - 2^{-n}$$

In this paper, the term *progress ratio* will be used interchangeably with the *percentage progress*, which is given by

$$p = (1 - 2^{-n}) \times 100$$

In order to describe mathematically the above equations, we need the values of the constant parameters, such as p and n (given above). Several studies have been carried out to estimate these parameters. The most common value of the progress ratio, p , reported in the literature, is approximately 0.20, or $p = 20\%$ [16][33]. (In Literature, such a learning curve is sometimes termed as 80% learning curve, but we will not use this definition).

In 1965, Levy [23] proposed that to improve the planning process, the learning behavior of the firm (besides that of the individuals) also needs to be understood. Earlier, in 1952, Hirsch [16] had found that about 87% of the changes in direct labor requirements were associated with the changes in technical knowledge, which is a form of organizational learning. More recently, in 1991, Adler and Clark [2] formed a Learning Process Model, in which the organizational learning is further attributed to (i) *engineering/technology* changes and (ii) to the labor *training*.

Compared to the other disciplines, it is clear that in software engineering we have not yet explored progress functions. If we are to get grips with process improvement (and in turn, product improvement, cost and cycle-time), we ought to investigate the role of humans in the software process. This paper examines only the progress functions aspects of human agents in the process. Nevertheless, it is an important thread, among other related human issues, in the software process field.

3 Learning through the Personal Software Process

An integral part of our experiment is the personal software process, PSP, designed explicitly by Humphrey [17] although this has existed implicitly to some degree in many organizations who perform product and process measurements, analysis and improvement. PSP facilitates learning through the

injection of various software practices. The principles of the PSP are to help the individuals to [17]:

- know their own performance: to measure their work, to recognize what works best, and to learn how to repeat it and improve upon it
- understand variation: what is repeatable and what they can learn from the occasional extremes
- incorporate these lessons in a growing body of documented personal practices

In this way, software developers would gain experience in empirical work, something in which they lag behind the disciples of other subjects, for example, the Physicists. Such experience, along with the specific features of PSP could lead to individual improvement. In turn this personal improvement will translate into team performance improvement, and finally into organizational improvement. Hence PSP is only one of the many steps towards higher organizational performance.

In PSP, as designed by Humphrey, software developers work on 10 small programming projects at a rate of one project per week. For each project, they kept track of measurements, such as:

- Logical Lines of Code, LOC, (new/reused/modified)
- Defects found (from 190 different defect types)
- Phase of Injection and Removal of Defects (from 8 phases)
- Time spent on fixing each defect
- Time spent on each activity and phase of the project
- Estimated and actual values of project size and time

The above measures are ample for studies on individual progress rate. However, for making comparisons amongst the subjects, we needed consistency in their measures. Therefore, all the software developers had to use the same environment, programming language (C/C++), physical and logical coding standards, defect type standards, data collection techniques, etc. throughout the course of the experiment.

Details of the background of each subject were taken from them. For this purpose, our measurement instrument included a 6 page assessment questionnaire [30], which the subjects filled. Later each subject had to appear in a 30 minute interview to verify and validate their personal data. This data included their education excellence level, job experience, programming experience, etc. Section 5.2 gives details of the subjects.

The subjects were given an intensive training program through which they were taught various software practices, hence providing them a second-order training. They attended three hours of interactive lectures every week with an opportunity to discuss, amongst other things, the measurement techniques and goals. This training program introduced one new concept every week, which helped them in improving their software development process. These concepts included:

- Measuring and Tracking the project
- Software Project Planning
- Statistical Methods for Estimating Size and Time
- Schedule and Resource Planning

- Code Reviews & Defect Prevention Strategies
- Structured Design Methods
- Cyclic Personal Process

The practices introduced to the subjects through PSP form an integral part of our model of learning, which is discussed below.

4 Model of Learning

Figure 2 represents our model of learning, which is derived from models developed by Levy in 1965 [23] and Adler and Clark in 1991 [2]. This figure shows that concurrent to any development activity are the two types of learning: first-order and second-order. These types of learning eventually lead to improvement. Note that this model is built around the first and second objectives of our study (see Section 1.0). In particular, this model uses our *hypothesis* as a base, to test if second-order learning contributes significantly to progress, compared to the first-order learning alone.

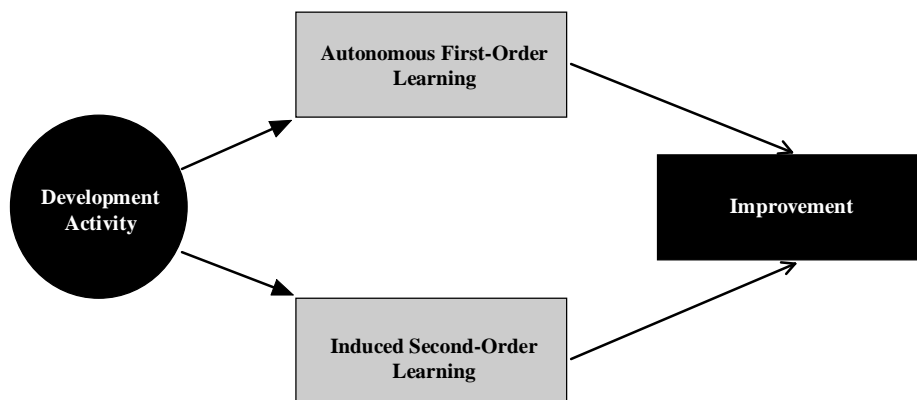


Fig. 2. A model for learning. Development activities are supported by first-order and second-order learning, which can lead to improvement.

Engineering measurements can be generally characterized as either process or product related [29]. We have emphasized both these types of measurements through five different progress functions: (i) progress in productivity, (ii) progress in size estimation abilities, (iii) progress in time estimation abilities, (iv) progress in productivity estimation abilities, and (v) improvement in product-quality. In each case, the progress ratio, p , is found (for the *first objective*) from the equations given for the learning curve.

It is important to realize that we are not measuring the constructs *product-quality*, *programmer productivity* or *personal skills* from all possible measurement angles. To do so would broaden the study beyond the scope intended and at the expense of the focus of this paper. The respective metrics used: LOC/hour, Defects/KLOC and Estimation Error, are specific angles through which we examine the described constructs. We are interested in measuring only the relative increase or decrease in the values for these metrics (and not their absolute values), which is sufficient for the purpose of studying progress functions from the specific angles chosen.

Two key aspects of the PSP are the Size Estimation method and Code Reviews. These practices are injected after the 3rd and the 6th projects

respectively, out of the total of nine programming projects and one non-programming one. A comparison of data from before and after the injection of these technologies can give an idea of their benefits (for the *second objective*). Below we describe the two technologies:

4.1 Size Estimation

Various methods exist in the literature for estimating the cost, size and time of a programming task, e.g., Boehm's COCOMO [10]. However, such models seem to work in certain environments, but not in others, and it is difficult to tailor them to the characteristics of different environments of the individuals.

At the PSP level, we need an estimation procedure which can utilize the conceptual design at the very early (planning) stage of the project to produce an estimate. One such method is Albrecht's Function Points [3]. However, such methods are known to have reliability problems [19] and are principally used for estimates in commercial data processing [17]. Therefore, we used a PROxy-Based Estimating (PROBE) method [17] in which an individual uses his or her own past data to produce a size estimate for each Object in the conceptual design. These Object sizes are then mapped to obtain the total program size, using the database of the individual's previous programs. Hence this method is customized to the needs of each individual subject.

4.2 Code Reviews

According to Fagan [15], there is evidence that early experience gained from code inspections causes programmers to reduce the defects in the later phases. Reports from industry [28] suggest that code inspections can be far more efficient than testing. In an experiment involving professional programmers, code reading detected more software faults than did functional or structural testing [9].

As a part of PSP, Humphrey has also emphasized code-reviews or inspections. Special emphasis is paid to those defects which have occurred frequently. For this purpose, the subjects analyze their past data and prepare Pareto charts of the most frequently occurring defects. A checklist of these defects is then made, which aids in the code review process.

In order to prepare Pareto charts, detailed past data of defect types is a prerequisite. Therefore, besides measuring the phases of injection and removal of defects and the time taken to fix each defect, the subjects were also required to categorize the defects using a defect types standard, which contained over 190 different defect types [30]. Subjects were also required to give an explanation for each defect. The list of defect types was prepared by the authors based on experience, as well as from the various defect categories given in [17] and [26].

5 Research Method

This section describes the *measurement instrument* (Section 5.1), followed by details on subjects, projects and environment (Section 5.2), the *experiment design* (Section 5.3), and the details of the *data collection process* (Section 5.4).

5.1 Measurement Instrument used for Gathering Data

We have used techniques from Social Sciences and MIS to add to Basili's Goal/Question/Metric measurement paradigm, GQM [8], especially in the area of validation tests [13]. The goals of our investigation and constructs were published in a 3-page workshop position paper [31] at the start of our experiment. Below we describe the steps which we followed in our GQM-derivative method:

1. Identify a set of goals based upon your needs
2. Define the Constructs which quantify these goals
3. Develop metrics which provide the data for the constructs
4. Validate the goals, constructs and metrics
5. Execute a mechanism for collecting and validating data
6. Analyze the data collected to study the goals, and check for statistical conclusion validity [32]

One of the difficult steps is defining the constructs, since it often requires the interpretation of fuzzy terms such as quality or productivity within the context of the development environment [6]. In order to formally validate our goals, constructs and metrics, we contacted seven experts in the field of software measurements, and asked them to fill out a fifteen page validation form [30]. This form gave details of the metrics, and asked the experts to check for, amongst other things, *content* validity. This survey was followed by detailed interviews with these experts wherever a divergence in views was found. Such a method of validating metrics is sometimes also referred to as *Face Validity* [21].

Table 1 describes the goals (G1-2), constructs (C1, C2a-b) and metrics (M1a-2, M2a-b), along with their units. We selected those constructs and metrics which have been used widely in previous studies, e.g., LOC/hr, Defects/KLOC. Through the process of validation described above, we are confident that the variables have high *effectiveness*, and the *construct* and *content validity* are strong. Our original model was to study the constructs/variables against cumulative output alone. However, when we began considering unhypothesized variables, we had to take into account the *motivation* and *problem complexity* (see [30] for details), which led to stronger *internal validity*. *External validity* has always been a critical issue for experiments. This experiment alone cannot be used to generalize the results for application in different software environments. No single experiment does. However, a well-defined and executed experiment adds to a body of knowledge, which can subsequently be used to make generalizations.

Another important issue in empirical studies is instrument *reliability*. However, the nature of our experiment was, in essence, to take the same measurements week after week. Further, the study was replicated twice. The fact that all these measurements gave consistent results reflects on the high reliability of our instrument. The initial questionnaire regarding the subject's background [30] was the only instrument that was used just once (at the beginning of each experiment). Hence for its reliability check, during the third and fourth projects, all the subjects were interviewed individually. During these 30 minute interviews, they

were re-asked all those questions where ambiguous, illogical or doubtful answers were initially given.

Table 1: Goals, Constructs and Metrics

Goals	Constructs	Metrics
G1: To measure the rate of improvement	C1: Progress Rate, p	M1(a): p in Programmer-Productivity (LOC/hr)
		M1(b): p in Defect Quality (Defects/KLOC)
		M1(c): p in Size Estimation Error
		M1(d): p in Time Estimation Error
		M1(e): p in Productivity Estimation Error
G2: To study the effects of technology injection	C2a: Improvement in Size Estimation Abilities	M2(a): Percentage Decrease in Size Estimation Error
	C2b: Improvement in Quality	M2(b): Percentage Decrease in Defect-Rate

5.2 Details on Subjects, Projects and Environment

All the three studies were conducted involving students enrolled in the PSP course at McGill University. The course is optional. The students were not paid for the work, but were graded. They were clearly told that they would be graded only on the quality of the data which they collect, and not on their productivity, defect-rate, estimation ability, or learning rate. This helped in ensuring that the students deliver comprehensive and accurate data.

Since the students voluntarily enrolled in the PSP course, the experimenters had no influence in the selection process. Sampling bias does occur, however, since there is a possibility that only those students who wanted to make their software development processes more disciplined, or learn more about PSP, enrolled in the course. Paid or forced subjects can also lead to bias and side effects.

The students were studying towards a Major or a Minor in Computer Science. This course led to Credit towards their degree. They were not paid. They were all full-time students, and were enrolled in other courses offered during the semester as well. During the first three weeks, they were allowed to drop out of the course freely. After this period, they could do so only at the expense of a Withdrawal (W) grade. They spent three hours per week on attending two PSP lectures and approximately four and a half hours per week working on the software projects. The motivation for them to collect reliable data lay in the fact that they were graded on the quality of data.

All the subjects had extensive programming experience. Particularly in C, they had at least 4 to 5 semesters of experience. Half of them had some full-time

job experience in the software development industry, while the other half had no such full-time or part-time job experience. In our initial study, we used only graduate students as subjects. For greater external validity, in the replication studies we used a mixture of graduate and undergraduate students. There were about 50 different subjective and objective attributes on subject and environment data, of which selected ones for the first study are tabulated in Appendix A.

Over 90% of the students used the School's computer laboratory. Hence their environment did not cause any variation in the results. The remaining ones used machines in their homes, with slightly different programming and physical environment. According to the results obtained in a study on programming environments by Curtis [11] (in which every subject had a different environment), our data could have at most 4% variation in the results due to non-uniform environment, which can be ignored.

The projects were small, on average 201 LOC and requiring four and a half hours for implementation. Although from such projects one should not make direct inferences about industrial scale projects, it is important to note that large projects are developed a bit at a time, by individuals. In so far as this is concerned, all the software development issues (such as size estimation of small components, coding of small objects, review code, etc.) in large systems are present in PSP projects. Thus for generalizability (for large-scale systems) from this PSP experiment, one needs to do a thorough contextual analysis of a large system. This analysis should examine the large system in terms of decomposed, small, components. At this point in our research, we feel that our PSP results could only be considered in light of such small components (possibly loosely coupled) in a large system.

5.3 Experiment Design

If the O's represent the observations and the X's represent the treatments, then a time-series design has the form:

O O ... O X O O ... O

Or, with a control group, it has the form:

Group 1 O O ... O X O O O

Group 2 O O ... O ~X O O O

where ~X means that the treatment X was not applied. Unfortunately, our context did not permit us to have a control group. Because of this contextual difficulty, we used a within-subject design, where a control group is needed only to vary the order of the treatments and not the types of treatments. A repeated-measure design has the pattern:

O X1 O X2 O

Or with a control group, it has the pattern:

Group 1 O X1 O X2 O

Group 2 O X2 O X1 O

In our design the two second-order treatments which we chose to study were:

X1 = Size Estimation Training

X2 = Code Reviews for Defect Removal Training

The above two treatments are mutually exclusive and are not known to affect each other. Hence we do not need two separate groups, and our design can be reduced to:

O X1 O X2 O

A positive aspect of our design is that there is not one but three observations before and after each treatment, increasing the reliability of the experiment. Thus, our design is:

O O O X1 O O O X2 O O O

where each observation is taken at a one week interval. Since the two treatments are independent, we can break up our design into:

O O O X1 O O O O O O

O O O O O O X2 O O O

These are in fact two separate Time-Series Quasi-experimental designs. If used carefully, Quasi-experimental designs can be used in valid scientific studies. In our case, we not only have such a design but also have several added features attributed to a true within-subject design. These include the pre and post test observations, repeated within-subject measures over a period of time, and random selection. In fact, our design is equal, at least, to a Time-Series Quasi-experimental design, and at most to a Repeated-Measure Within-Subject True experimental design.

Experiment designs have various threats to validity [21]. *Maturation* is any naturally occurring process within the subjects that can cause a change in their performance, e.g., fatigue, boredom, stress, etc. In our case, this was important because the subjects could not voluntarily withdraw from the study after the first three projects. Sometimes the pre-test observations sensitize the subjects, for example, by making them relaxed. Such a threat is known as *Testing*. Similarly, *Hawthorne Effect* is the tendency for subjects to show increased productivity when their performance is being monitored. An advantage of using Time-Series design with a large number of observations is that these three threats are considerably reduced. Further, since we are measuring relative values of metrics instead of absolute ones, the effects of those threats which apply to all the observations are neutralized.

However, due to the long period involved, several other threats may be perceived. *Mortality* refers to any dropout of subjects while *Instrumentation* refers to any changes in the measurement procedures, during the middle of the experiment. In our experiment, neither of these two scenarios occurred. *History* refers to any event that coincides with the treatment and could have a similar effect as the treatment. Since none of the subjects were learning process improvement techniques or object-oriented programming elsewhere, we can safely rule out this threat.

We did not reveal our goals to the subjects, hence avoiding *Demand Characteristics*. The subjects were repeatedly assured about *Confidentiality*. However, the threat due to *Evaluation Apprehension* cannot be totally ruled out since the subjects knew that one purpose of the PSP course was for them to improve, and hence they might have acted in such a way as to give desirable data. We counteracted this threat with repeated discussions on the importance of PSP for oneself, and that honesty was the best policy as no one was penalized for defects, low productivity or poor estimation skills.

5.4 Data Collection

We employed a goal-directed method for data collection [7]. One of the most important aspects of this method is to validate the data. Most of the data collection forms which we used in this experiment were initially designed and validated by Humphrey and his colleagues, and later by the software engineering group at McGill University, which made minor changes during the validation process. In addition, from our research perspective, we had to develop some extra measurement instruments.

There are various different data collection techniques, of which we used the Logs, Forms, Templates, Spreadsheets, Databases, Summary Reports, Automatic LOC Counters and Automatic Complexity Analyzers. Following are the steps we took for our data collection process:

1. We devised an initial six page questionnaire for subjects background, of which two pages were prepared by Humphrey. The other four pages were prepared and validated by the software engineering group at McGill University.

2. Every week the subjects were given a project, along with detailed instructions on how to complete it. These projects were such that they helped the subjects in following PSP. e.g., developing LOC counters for measuring their program sizes, developing tools to help them in size estimation, etc. A description of the requirements was given in briefing sessions with further clarifications via Email or individual help.

3. The subjects collected detailed data in three stages: (a) Planning, where they made estimates of program size, time, etc.; (b) Execution, where they worked on the project and simultaneously recorded the time and defect data for each phase; and (c) Post-Mortem, where they completed the summary reports. They were told explicitly that the main criteria for evaluating their work was the quality of the data they collected.

4. Subjects data was then checked for consistency, accuracy and logical validity. Automatic tools were also used to verify the program size values. We also checked if any two subjects were illegally exchanging code, but never found such an occurrence.

5. A weekly feedback report was then given to each subject, informing them of any errors and giving advice for future. It was in these feed-back reports that the subjects were assigned grades as well. These reports ensured that the data is not deficient in correctness, consistency or completeness.

6. Between the third and fourth project, all the subjects were interviewed. Each interview was at least 30 minutes long and detailed minutes were recorded. Here the subjects were asked details of any assumptions they had made in their data collection process.

7. It is beneficial to include the data-suppliers in the data-collection process so that they can provide reliable data [7]. During the second half of the experiment, the subjects had enough data points from the first half (projects 1 through 5). They were then asked to use GQM to identify their own goals and then to analyze them using their data.

From the above descriptions, we note that under the constraints of time and budget, we have attempted to ensure that our data is as consistent, complete and correct as it can get.

6 Data Evaluation and Analysis

This section presents an analysis of the collected data, with particular emphasis on statistical validity. As mentioned in Section 1, we had two objectives. Analysis for both these objectives is given below.

6.1 Objective 1: Rate of Progress

As described earlier, the following five learning indices were used to measure the progress:

M1(a) p in Programmer Productivity

M1(b) p in Defect Quality

M1(c) p in Size Estimation Abilities

M1(d) p in Time Estimation Abilities

M1(e) p in Productivity Estimation Abilities

In order to calculate p, we carried out regression analysis to find the best-fit equation of the learning curve for each subject. For example, Table 2 gives Size Estimation Error calculation results of a sample subject.

Table 2: Sample Size Estimation Results

Study No.	2			
Subject No.	22			
Project No.	Estimated Size (LOC)	Actual Size (LOC)	Size Estimation Error (%)	Cum. Output (LOC)
1	50	75	33.33%	75
2	143	171	16.37%	246
3	205	253	18.97%	499
4	238	111	114.41%	610
5	92	86	6.98%	696
6	177	264	32.95%	960
7	233	257	9.34%	1217
8	96	102	4.90%	1319
9	263	277	5.05%	1596

The Percentage Size Estimation Error is then plotted against the Cumulative Output, as shown in Figure 3.

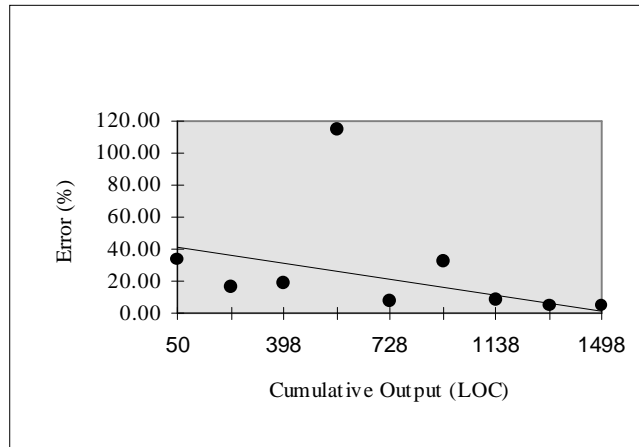


Fig. 3. An example of regression. Percentage Size Estimation Error is plotted against Cumulative Output, in a scatter diagram. The best-fit line is sketched and its linear and logarithmic equations are determined.

Linear and Logarithmic Regression is used to find the equation of the best-fit through the scatter graph of Figure 3. Quadratic Regression was also carried out, but the decrease in regression error due to it was not statistically significant compared to Linear Regression. Hence the Quadratic Equations were not used. Linear Regression gives an equation of the form $y = ax + b$, which in this case is $y = -0.02x + 42.05$. Similarly, Logarithmic Regression gives an equation of the form $y = Kx^n$, which in this case is $y = 214.33x^{-0.415}$. From this value of n (-0.415), p can be calculated using $p = (1 - 2^n) \times 100$, which gives $p = 24.99$. This is the Progress Rate in Size Estimation.

Similarly, the Progress Rate for the other four metrics is also calculated. The average value of p for all five metrics is then calculated for each subject. These values are shown in Table 3.

The average learning index is 18.8% with a standard deviation of 14.0. However, observe that there is an outlier with $p = -18.0$, which represents negative learning (or *forgetting*). After discarding this outlier, the average learning index is 22.1% with a standard deviation of 8.9.

Similar results were obtained from the two replication studies. These are listed in Appendix B in detail, and are summarized in Table 4 below.

Table 3: Progress Percentages for the First Study

Subject No.	Progress %, p
1	30.4 %
2	37.6 %
3	14.4 %
4	17.9 %
5	31.4 %
6	30.5 %
7	16.0 %
8	15.4 %
9	26.8 %
10	10.8 %
11	-18.0 %
12	12.2 %
Mean	18.8 %
Std. Dev.	14.0
Mean (without Outlier)	22.1%
Std. Dev. (without Outlier)	8.9

Table 4: Progress results for the three studies

Study No.	Subjects	Progress %, p	Std. Dev.
1	11	22.1 %	8.9
2	30	13.2 %	9.0
3	12	28.5 %	14.8
	Total = 53	Weighted Mean	12.1
		= 18.5 %	

The average progress percentage for the 53 subjects was 18.5%. Note that this is close to the value of 20% reported in various past research results [16][33] from other disciplines. Past studies also show that the variance in results is expected to be high [12], as is true in our case.

6.2 Objective 2: Second-Order Learning

The second objective was to determine the extent to which the injection of technologies helped in personal improvement, over and above first-order learning. Consider the case of Code Reviews (X2). Our design can be represented as:

O1 O2 O3 O4 O5 O6 X2 O7 O8 O9

If we plot only the points O1 through O6, and find the best-fit through them, we would obtain the equation of the learning curve representing first-order learning alone (see Figure 4), since the second-order technology has not yet been injected. This equation would give the defect-rate in terms of the cumulative

output. Based on this equation, we can extrapolate and predict the defect-rate for the next three observations (O7, O8 and O9).

These predicted values can then be compared with the actual measurements of the defect rates, following injection of X2. Whereas the predicted values represent the expectations from first-order learning alone, the actual points would represent the extra decrease in the defect-rate attributable solely to X2.

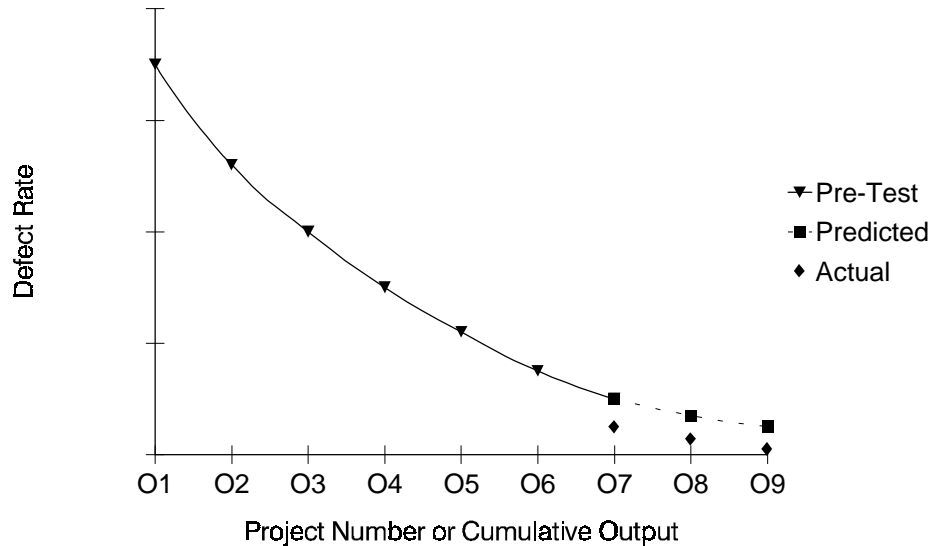


Fig. 4. Extrapolation of Pre-test Data Points. Defect Rate (in Defects/KLOC) is plotted against the Cumulative Output, or more simply, against the project number.

The first six data points (termed as pre-test) represent the observations/measurements before the treatment was applied. Regression was carried out on these six points to obtain equation of the best-fit. This equation was used to predict the values of defect-rate for the next three observations, using extrapolation. Finally, these Predicted values were compared with the Actual values which were measured in the study. Whereas the predicted values represent the defect rate expected from the first-order learning alone, the actual values highlight the decrease due to second-order learning.

In our case, the mean defect-rate for the actual values was 12.6% less than the mean defect-rate of the predicted values. Hence the injection of technology helped the subjects improve by approximately 13% over and above the improvement due to first-order learning.

A similar analysis showed that size estimation training (X1) reduced the estimation error by 6.5%. These results are summarized in Table 5.

Table 5: Effects of Second-Order Learning

Software Development Technology	Improvement due to Technology injection
X1 = Size Estimation Techniques	6.5 %
X2 = Code Reviews	12.6 %

7 Observations

We observe from the described investigation that technology insertion in the software process helps improve human performance in the process. This is not new and is intuitive. However, the investigation suggests that total improvement is approximately 20% (see Table 4), which is comparable to that found in other disciplines (see Section 2.0 on Related Work).

An organization can immediately use such knowledge to its advantage. For example, CAE Electronics, a developer of real-time flight simulator line of products in Canada, recognized the described laboratory results as a basis for justifying investment in the area of human resource development for their numerous software projects. In this respect, a major value of the described laboratory research is that it can act as a trigger for kick-starting initiatives in industry. In addition, however, the CAE study shows that the defect yield had improved from 12% to 28% with the introduction of code reviews and the reduction in testing time was from 37% before code reviews to 17% afterwards [14]. One can thus see the benefits of carrying out laboratory experiments and connecting them to appropriate studies in industry.

Besides the general observation of 20% improvement in the laboratory experiment, our investigation has isolated the improvement effects due to second-order learning. In our study, the improvements due to second-order learning were 6.5% for size estimation and 12.6% for code reviews. This separation of improvement effects in human performance of software processes can be considered as a new and significant insight.

As described by Gestalt, it is probably no longer the case that first-order learning is of prime importance. Organizations can take advantage of the insight gained from our study in specific ways. For example, organizational maturity goals could be aligned to attract new contracts based on second-order learning and improvement-predictions on human performance in software development.

Also, second-order learning (or technology injection) can be directed at those areas of software processes that are likely to yield relatively significant benefits. For example, what are the implications (for a given project) of the fact that benefit due to X1 is 6.5%, and that due to X2 is 12.6%? Similarly, what are such benefits in other areas of software development, such as requirements engineering, design, testing, system integration, etc.? These questions alone suggest the need to carry out further investigations in the domain of human-oriented process improvements.

8 Conclusion

This paper describes a method for setting up and executing an experimental study. This includes a standard paradigm for laying out our goals and constructs, the development of a model (as depicted in Figure 2), building up of a measurement instrument to study the model, validation of the instrument, setting up of a formal experiment design, validation of the data collection process, and execution of the experiment along with replications. These techniques have been selected from the fields of Social Sciences, Experimentation, MIS and Software Engineering.

Our investigation into human-oriented software process improvement suggest an overall gain of 20% in specific areas investigated (size estimation and code reviews). Our results also separated, quantitatively, second-order learning effects from first-order learning effects in the software processes identified.

While these laboratory results are encouraging, they should not be used as-is in other environments without assessing the contexts. In fact, replication of such investigations in other environments is urgently needed, and is encouraged by the authors. Only then will we be able to build general principles and rules of thumb on a solid empirical foundation.

BIBLIOGRAPHY

- [1] Call for Papers, 9th International Software Process Workshop, Airlie, Virginia, October 1994
- [2] Adler, P. and Clark, K., "Behind the Learning Curve: A Sketch of the Learning Process," *Management Science*, vol. 37, no 3 (May 1991), pp 267-281
- [3] Albrecht, A., Gaffney J. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, vol. SE-9, no 6, Nov. 1983, pp 639-648
- [4] Argote, L., Beckman, S. and Epple, D. "The Persistence and Transfer of Learning in Industrial Settings," *Management Science*, vol. 36, no 2, (Feb. 1992), pp 140-154
- [5] Arrow, K. "The Economic Implications of Learning by Doing," *Review of Economic Studies*, vol. 29 (April 1962a), pp 166-170
- [6] Basili, V. "Quantitative Evaluation of Software Methodology," Technical Report TR-1519, Dept. of CS, University of Maryland, July 1985
- [7] Basili, V. and Weiss, D. "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, vol. SE-10, no 6 (Nov. 1984) pp 728-738
- [8] Basili, V., Rombach H., "Goal Question Metric Paradigm," *Encyclopedia of Software Engineering*, vol. 2, 1994, John Wiley & Sons, Inc.
- [9] Basili, V., Selby R. "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, vol. SE-13, no 12, Dec. 1987, pp 1278-1296
- [10] Boehm, B. "Software Engineering Economics," 1981, Prentice-Hall, Englewood Cliffs, NJ
- [11] Curtis, B., et al. "Productivity Factors and Programming Environments," *Proceedings of the Seventh International Conference on Software Engineering*, Washington DC, IEEE Computer Society, pp. 143-152

- [12] Dutton, J., Thomas, A. and Butler, J. "The History of Progress Functions as a Managerial Technology," *Business History Review*, vol. 58 (Summer 1984), pp 204-233
- [13] Emam, K. E., Moukheiber, N. and Madhavji, N. "The Empirical Evaluation of the GQM," *Proceedings CASCON 1993*, vol. 2, pp 265-289
- [14] Emam, K. E., Shostak B., Madhavji N. H., "Implementing the Personal Software Process in an Industrial Setting," *Proc. 4th Int. Conf. on Software Process*, Brighton, U.K., Dec 1996 (To appear)
- [15] Fagan, M. "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, vol. SE-12, no 7, Jul. 1986, pp 744-751
- [16] Hirsch, W. "Manufacturing Progress Functions," *The Review of Economics and Statistics*, vol. 34 (May 1952), pp 143-155
- [17] Humphrey, W. "A Discipline for Software Engineering," Addison Wesley, 1995
- [18] Kemerer, C. F. "How the Learning Curve Affects Case Tool Adoption," *IEEE Software*, May 1992, pp 23-28
- [19] Kemerer, C., Porter B. "Improving the Reliability of Function Point Measurement: An Empirical Study," *IEEE Transactions on Software Engineering*, Vol. 18, No 11, Nov. 92, pp 1011-1024
- [20] Kerlinger, F. "Foundations of Behavioral Research," 3rd ed., 1986, Holt, Rinehart and Winston, New York, NY
- [21] Kidder, L. "Research Methods in Social Relations," 5th ed., 1986, Hold, Rinehart and Winston, NY.
- [22] Kleinbaum, D. "Applied Regression Analysis and other Multivariable Methods," 2nd ed., 1988, PWS-Kent Pub. Co., Boston, MA
- [23] Levy, F. K. "Adaptation in the Production Process," *Management Science*, vol. 11, no 6, (April 1965), pp B136-B154
- [24] Miles, R. E., Snow, C. C., "Organizational strategy, structure and process," McGraw-Hill, NY, 1978
- [25] Rapping, L. "Learning and World War II Production Functions," *Review of Economics and Statistics*, vol. 47, 1965, pp 81-86
- [26] Ripley, D., Druseikis F. "A Statistical Analysis of Syntax Errors," *Computer Languages*, Vol. 3, 1978, pp 227-240
- [27] Rosenthal, R. "Essentials of Behavioral Research: Methods and Data Analysis," 1984, McGraw-Hill, New York, NY
- [28] Russell, G. "Experience with Inspection in Ultra-Scale Developments," *IEEE Software*, Jan 91, pp 25-31
- [29] Shepperd, M. "An Evaluation of Software Product Metrics," *Information and Software Technology*, Vol. 30, No 3, April 1988, pp 177-188
- [30] Sherdil, K., "Personal Progress Functions in the Software Process," Masters Thesis, School of Computer Science, McGill University, Montreal, QC, Feb 1995
- [31] Sherdil, K., Madhavji N. "Personal Progress Functions in the Software Process," *Proceedings of 9th International Software Process Workshop*, Airlie, Virginia, IEEE Computer Society, Oct 1994, pp 117-121
- [32] Straub, D. "Validating Instruments in MIS Research," *MIS Quarterly*, June 1989
- [33] Yelle, L. "The Learning Curve: Historical Review and Comprehensive Survey," *Decision Sciences*, vol. 10 (Feb. 1979), pp 302-328

Appendix A: Statistics on the Subjects, Projects and Environment, for the First Study

Subject Attributes	Value
Number of Males	10
Number of Females	2
Mean experience with C Language	28 months
Median experience with C Language	24 months
Mean total programming experience	6.5 years
Median total programming experience	6.0 years
Number with full-time and part-time job experience	6
Number with no job experience	6

Project Attributes	Value
Average Total Size of a program (including Reused Code)	201 LOC
Average Time spent on a project	4 hrs. 27 min.
Average Reused Code per program	81 LOC
Average New and Changed Code per program	120 LOC
Average Defects recorded per program	10.3

Environment Attributes	Value
Subjects using standard machine and (Gnu) compiler	10
Subjects using personal computers and Borland compiler	2
Subjects using the standard laboratory	9
Subjects using different laboratories	3

Appendix B: Progress Percentage, p, for second and third studies

	Second Study	Third Study
Subject No.	Progress Percentage	Progress Percentage
1	-9.1 %	27.4 %
2	7.7 %	24.3 %
3	14.1 %	36.6 %
4	12.4 %	11.4 %
5	-8.4 %	40.0 %
6	10.8 %	51.1 %
7	8.8 %	37.2 %
8	22.5 %	32.0 %
9	14.3 %	2.0 %
10	15.1 %	10.8 %
11	15.4 %	30.0 %
12	22.8 %	29.8 %
13	9.7 %	
14	16.8 %	
15	17.7 %	
16	20.1 %	
17	20.7 %	
18	12.0 %	
19	-5.6 %	
20	4.0 %	
21	8.3 %	
22	19.0 %	
23	15.8 %	
24	21.4 %	
25	17.2 %	
26	26.4 %	
27	22.3 %	
28	23.5 %	
29	9.8 %	
30	10.6 %	
Mean	13.2 %	28.5 %
Std. Dev.	9.0	14.2