SUBMITTED DRAFT

THE ECONOMICS OF SOFTWARE DEVELOPMENT BY PAIR PROGRAMMERS

HAKAN ERDOGMUS
National Research Council, CANADA

LAURIE WILLIAMS
North Carolina State University, USA

ABSTRACT

Evidence suggests that pair programmers – two programmers working collaboratively on the same design, algorithm, code, or test – perform substantially better than the two would working alone. Improved quality, teamwork, communication, knowledge management, and morale have been among the reported benefits of pair programming. This paper presents a comparative economic evaluation that strengthens the case for pair programming. The evaluation builds on the quantitative results of an empirical study conducted at the University of Utah. The evaluation is performed by interpreting these findings in the context of two different, idealized models of value realization. In the first model, consistent with the traditional waterfall process of software development, code produced by a development team is deployed in a single increment; its value is not realized until the full project completion. In the second model, consistent with agile software development processes such as Extreme Programming, code is produced and delivered in small increments; thus its value is realized in an equally incremental fashion. Under both models, our analysis demonstrates a distinct economic advantage of pair programmers over solo programmers. Based on these preliminary results, we recommend that organizations engaged in software development consider adopting pair programming as a practice that could improve their bottom line. To be able to perform quantitative analyses, several simplifying assumptions had to be made regarding alternative models of software development, the costs and benefits associated with such models, and how these costs and benefits are recognized. The implications of these assumptions are addressed in the paper.

INTRODUCTION

Both anecdotal and statistical evidence [10, 24, 32, 34] indicate that pair programming, the practice whereby two programmers work side-by-side at one computer collaborating on the same design, algorithm, code or test, is highly productive. One of the programmers, the driver, has control of the keyboard/mouse and actively implements the design, program, or test. The other programmer, the navigator, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects and also thinks strategically about the direction of the work. On demand, the two programmers brainstorm any challenging problem. Because the two programmers periodically switch roles, they work together as equals to develop software. Many have used the pair programming technique for decades, and several publications in the mid-late 1990s extolled its benefits [11, 12, 24]. More recently, many impressive anecdotes among those practicing the Extreme Programming (XP) software development methodology [1, 3, 6, 30, 31] greatly aroused awareness of pair programming as a technique to improve quality, productivity, knowledge management, and employee satisfaction [10, 25, 34].

In 1999, a formal experiment was run to investigate the effectiveness of the pair programming practice. The experiment was run with advanced undergraduates at the University of Utah. Sometimes issues of external validity are raised when empirical software engineering studies are conducted with students. These issues arise because projects undertaken within a semester in artificial settings need not deal with matters of scope and scale that often complicate real, industrial projects. However, such settings are still valuable as test-beds. They have the potential to provide sufficient realism at low cost while allowing for controlled observation of important project parameters [13]. The University of Utah empirical study focused on the interactions between and the overall effectiveness of two programmers working collaboratively relative to programmers working alone. Issues of complexity and scale are not significant inhibitors in such a study.

Software development methodologies, or *processes*, are prescribed, documented collections of software practices (specific methods for software design, test, requirements documentation, maintenance, and other activities) required to develop or maintain software. Williams developed the Collaborative Software Process^{SM} (CSP^{SM}) methodology as her dissertation research [35]. CSP is based on Watts Humphrey's well known Personal Software Process^{SM} (PSP^{SM}) [21], but is specifically designed to leverage the power of two programmers working together. The University of Utah experiment assessed the

effectiveness of solo programming using the PSP vs. the effectiveness of pair programmers using the CSP. These two processes were specifically chosen to best isolate the effects of pair programming; essentially all the other practices followed by the programmers were identical. The experiment yielded statistically significant differences between the performance of pair programmers and of individual programmers [9, 10, 34, 35]. In this paper, these experimental results are used to perform a quantitative analysis of the economic feasibility of pair programming. The findings complement and strengthen the qualitative benefits of pair programming that have been reported previously.

The economic feasibility of pair programming is a key issue. Many instinctively reject pair programming because they believe code development costs will double: why should two programmers work on each task while a single programmer can do the job? If the practice is not economically feasible, managers simply will not permit its use. Organizations decide whether to adopt process improvements based on the bottom-line implications of the outcomes. Naturally, the goal of software firms is to be as profitable as possibly while providing their customers with the best, high-quality products quickly and cheaply.

The economic feasibility analysis of the pair programming practice centers on how it fairs relative to solo programming under a given value realization model. We assume a product realizes value when clients or end users are delivered a working product. Even a partial, but working, product can provide benefits. We will compare pair programming with solo programming first based on simple performance metrics, and then considering these metrics under two different value realization models. In the latter case, the analysis utilizes Net Present Value (NPV) [27] as the basis for comparison. This approach per se is not novel. Economic models based on NPV have previously been suggested to evaluate the return on software quality and infrastructure initiatives; for examples, see [7, 14-16, 23]. Our analysis differs in that it relies on a breakeven analysis instead of a pure NPV analysis.

ASSUMPTIONS

The economic feasibility of pair programming is assessed by focusing on the performance of a single pair of programmers with respect to the performance of a solo programmer, under the assumption that both the pair and the solo programmer are undertaking the same programming task. Whether the solo or pair programmers work in isolation or are part of a larger project team is thus immaterial.

The assessment makes a number of other simplifying assumptions. Some of these assumptions abstract away from extraneous factors over which the programmers or developers normally have no control, while others reduce the number or limit the behavior of the underlying variables in order to make a quantitative comparison possible:

- Cost accumulation. Labor cost is the only kind of cost considered. All costs are recognized instantly as they are accrued. One-time overhead costs, such as the pair jelling time [35], are disregarded. Since we compare a single programming pair to a single programmer working alone, the pattern of expenditures for labor costs is linear in each case: costs are accrued continuously and at a constant rate.
- Measures of time and effort. All variables that measure time and effort use *compressed time*. Compressed time excludes idle time, interruptions, and time spent on non-development or extra-task activities.
- Defect recovery process. The post-deployment defect discovery process is assumed to be perfectly efficient. This implies that after a piece of code has been deployed, all defects are found instantly. We assume that post-deployment defects are found by the clients of the deployed code, be them end users or a quality assurance team. Hence the defect discovery time is not included in the development effort. (This is also implied by the compressed-time assumption: from the programmers' perspective, the time it takes for post-deployment defects to be discovered is idle time, and as such, it is disregarded in the analysis.) Defects are fixed at a fixed rate, which depends on the development process.
- Value realization. A linear relationship is assumed between the amount of code deployed by the programmers and the value generated through the development activity. Deployed code instantly realizes value when it is defect-free. Code may be deployed in arbitrarily small increments.
- Ranges and baseline values of model parameters. Whenever necessary and reasonable, statistics previously reported in the literature are used to determine the ranges and baseline values of model parameters.

ABSTRACTION OF THE DEVELOPMENT PROCESS

The two development processes that underlie the comparison are the Personal Software Process (PSP), which is designed for individual programmers, and the Collaborative Software Process (CSP), which is designed for pair programmers. The CSP practices are intentionally based on PSP practices, with the exception of pair programming. As a result, we consider that our comparison of PSP and CSP is essentially a comparison between solo and pair programming. The shared practices of the two processes, therefore, are not discussed here.

In what follows, we refer to a single developer or a team of developers simultaneously working on a piece of code as a *work unit*. We represent the development process in terms of two descriptive and three empirical parameters. The size of the work unit uniquely differentiates the CSP from the PSP.

DESCRIPTIVE PARAMETERS:

The two descriptive parameters of the development process are:

- N: *size of the work unit* (persons). The number of developers in a work unit. N equals 1 for a solo programmer (hereby, a *soloist*), and 2 for a pair of programmers (hereby, a *pair*). Thus $N = 1$ if the work unit consists of a soloist following the PSP. $N = 2$ if the work unit consists of a pair of developers working in tandem on the same task following the CSP.
- υ: *value realization model.* The pattern in which a work unit delivers a finished or partial product, and accordingly generates value. This parameter will be discussed later in the paper.

The work unit (N) and the value realization model (υ) are the only independent parameters in the process model. When we vary N, we always keep the value of υ constant. The values of the empirical parameters all depend on N.

EMPIRICAL PARAMETERS:

Before we introduce the empirical parameters, we need to define how we measure the *output* (denoted by ω) of a work unit. A work unit, depending on how efficient it is, is able to produce only a certain amount of output within a given time. Conversely, a work unit, again depending on its efficiency, requires a certain amount of time to produce a given amount of output. In the latter case,

the output targeted by the work unit can be thought of as the *size* of the project or task undertaken.

In software development, output is an elusive concept to represent and measure. It's by and large a subjective notion whose interpretation it the cause of much controversy. To be meaningful, the measure of output should correlate with how much technical functionality is provided by the software artifact produced. Yet there is no universally accepted way of counting technical functionality. We use the most widely adopted and easy to compute measure, *lines of code* (LOC). However, LOC is just a proxy. Some argue that it is not an appropriate measure of output in that LOC may not always correlate well with the amount of functionality delivered. More abstract measures, such as function points, have been suggested as alternatives, but these are not suitable for use in our analysis because of their coarse and non-uniform granularity.

The unit of two empirical parameters, productivity and defect rate, depend on the adopted unit of ω. If LOC is substituted by another output measure, the units of productivity and defect rate will change accordingly.

Having defined output, the three empirical parameters of the development process are:

- π: *productivity* (LOC/hour). The average hourly output of the work unit.
- β: *defect rate* (defects/LOC). The average number of defects per unit of output (per LOC) associated with the work unit.
- ρ: *rework speed* (defects/hour). The speed at which the work unit fixes defects in a piece of previously deployed code, after the defects have been discovered.

The values of these three parameters are determined empirically based on past research studies and statistics reported in the literature. The chosen values are primarily for illustration purposes, and represent information available at the time of writing. The actual values could be different, and they would most likely be both project- and skill-dependent. The specific results reported here are sensitive to the empirical parameters to varying extents, however we believe that the general conclusions are much less so under the assumptions of the analysis. A sensitivity analysis is performed at the end of the paper.

PRODUCTIVITY :

According to a study by Hayes and Over [19], the average productivity rate of 196 developers who took PSP training was 25 LOC/hour. This figure will be the chosen value of π for $N = 1$ (soloist). Note that the developers in the PSP training course were essentially free from normal business interruptions. As a

result, this figure may seem high when compared with productivity figures based on monthly rates in which programmers' total output is compared with their total time (including meetings, absences, vacation, etc.). However, the Hayes and Over productivity figure is appropriate for our analysis as we use compressed time in all measures. We use the term *compressed time* to refer to pure programming time (including rework time that is associated with fixing defects). Compressed time excludes interruptions, vacation, and idle time.

The University of Utah study [34, 35] reported that a pair spends on average only 15% more effort (in total person-hours) than a soloist to complete the same programming task. This result however was not statistically significant, with approximately a 40% probability of the observed difference in the mean being due to chance. Although further analysis was not performed on the data set to verify whether the two-tailed t-test employed was powerful enough to detect the difference at the specified alpha level in the first place, anecdotal evidence [1, 32, 33] is supportive of no significant total effort penalty for pair programmers after *pair jelling* has occurred [2, 34, 36]. Pair jelling is the time period in which programmers learn to work effectively in a pair, to give and to accept objective suggestions, and to communicate during development. In our firsthand observations, there is a one-time jelling cost of between 1to 40 hours the first time a programmer pairs. Subsequently, there is another short 30-60 minute jelling period when a programmer pairs with a different programmer for the first time; during this time the programmers learn each other's strengths and weaknesses relative to their own.

We err on the conservative side by assuming that the observed 15% difference is real. With this assumption, in a single person-hour, each programmer of a pair produces an average of 25/(1.15) = 21.74 LOC, and together they produce twice this volume, or 43.48 LOC. Thus, benchmarked relative to the baseline PSP productivity level of 25 LOC/hour, the value of π for N =2 (pair) is taken to be a conservative 43.48 LOC/hour.

These pair productivity rates are within 20-30% of those recently reported by a technology company in India that used both pair and solo programming in a Voice-over-IP project. This project reported a pair-to-soloist productivity ratio of 2.8 (3.3 KLOC/month for solo programmers versus 9.6 KLOC/month for pair programmers based on a 60-hour work week) [33]. Note that this ratio is much higher compared to the more conservative ratio of 1.74 adopted here.

DEFECT RATE :

According to Jones [22], code produced in the US has an average of 39 raw defects per thousand LOC (KLOC). This statistic is based on data collected from such companies as AT&T, Hewlett Packard, IBM, Microsoft, Motorola, and

Raytheon, with formal defect tracking and measurement capabilities. According to the same reference, on average, 85% of all raw defects are removed via the development process, and 15% escape to the client.

Together the two pieces of statistics suggest an average post-deployment defect rate of (0.039)(0.85) = 0.00585 defects/LOC. The number is consistent, though on the low side, with data from the Pentagon and the Software Engineering Institute, which indicate that typical software applications contain 5-15 defects per KLOC [18]. We adopt the average 0.00585 defects/LOC as the baseline soloist value of β, for $N = 1$.

During the empirical study of pair programming vs. solo programming, Williams [9, 10, 34, 35] observed that at the end of the project, code written by pairs on average passed 90% of the specified acceptance tests compared to code written by soloists, which passed on average only 75% of the same test suite. The results were statistically significant at an alpha level of less than .01. Assuming that the test suite provided full coverage, this result suggests a pair-to-soloist post-deployment defect rate ratio of .6 (corresponding to an improvement rate of 1 – .6 = 40%). Thus benchmarked relative to the soloist ($N = 1$) baseline value of 0.00585 defects/LOC, the adopted value of β for a pair ($N = 2$) is (0.00585)(0.6) = 0.003510 defects/LOC.

The adopted soloist value of β is close to the average defect rate of 0.00534 defects/LOC reported by the Indian company mentioned previously [33]. However, for pairs, the company reported defect rates that are an order of magnitude lower than the adopted β value of 0.003510 defects/LOC, both during unit testing (at 0.0002 defects/LOC) and during acceptance testing (at 0.0004 defects/LOC), corresponding to an improvement of over 90% over soloists. In the initial analysis, we will err on the conservative side again by adopting the figures yielded by the University of Utah study. Later in the paper, sensitivity analysis will show how an improvement as dramatic as the one reported by the Indian company affects the results.

REWORK SPEED :

A study of a set of industrial software projects from a large telecommunications company [29] reported that each discovered (post-deployment) defect required an average of 4.5 person-days, or 33 person-hours of subsequent maintenance effort or rework (based on a 7.5-hour workday). This statistic is consistent with data reported by Humphrey [21]. Based on this observation, the value of rework speed ρ for a soloist ($N = 1$) is taken to be 1/33 = 0.0303 defects/hour. Again this servers as our baseline value for computing the pair defect rate.

No data is available regarding the effect of pair development on rework activities. We will assume pairs can achieve rework productivity gains comparable to those reported for the initial development activities. Under this assumption, the estimated rework speed ρ for a pair (N = 2) will be (2)(0.0303)/1.15 = 0.0527 defects/hour. This assumption would especially be valid for agile development processes such as Extreme Programming, where no clear separation exists between rework and development activities.

INITIAL ABSTRACT MODELS:

For now we leave υ, the value realization model, unspecified since it will not be needed for the initial comparison. Thus the initial abstract models that represent the two development processes are:

$$\text{Solo} = \{N = 1, \pi = 25.0, \beta = 0.00585, \rho = 0.0303\},$$

$$\text{Pair} = \{N = 2, \pi = 43.478, \beta = 0.003510, \rho = 0.0527\}.$$

Note that pair jelling costs [2, 34, 36] have been excluded in this model. At this point, we have no viable empirical data beyond the anecdotes discussed above regarding jelling costs. Exclusion of jelling costs injects a bias into the analysis in favor of pair programming. If jelling cost is a one-time cost, this bias should not be significant. However if it is recurring due to pair rotation or turnover, it should be factored into the productivity parameter to eliminate the bias. Fortunately, productivity is the least sensitive of the three empirical parameters, as discussed below. This helps reduce the bias in the analysis.

THE BASIC COMPARISON MODEL

The basic comparison model consists of three metrics: efficiency, unit effort, and unit time.

EFFICIENCY:

Efficiency, ε, is defined as the percentage effort spent on developing new code, exclusive of the effort expended on rework. Given a productivity rate of π, the effort required to produce ω lines of code of output is given by:

$$E_{pre} := \frac{\omega\, N}{\pi}$$

This quantity specifies the *initial development* (or *pre-deployment*) *effort*. Initial development is followed by *rework* (or *post-deployment*) *effort* once the

code has been deployed (fielded, or delivered to the client). Rework effort, E_{post}, refers to the maintenance effort expended to fix runaway defects after a piece of new code has been deployed and all such defects have been found.

$$E_{post} := \frac{\omega\ \beta\ N}{\rho}$$

Here ωβ is the total number of defects and ρ is the speed of rework. Effort is always adjusted to the work unit by multiplying it by the work unit's size N.

Total effort, E_{tot}, is the sum of the initial development and rework efforts:

$$E_{tot} := \frac{\omega\ N\ (\rho + \beta\ \pi)}{\pi\ \rho}$$

Efficiency, ε, is then the ratio of the initial development effort E_{pre} to the total effort E_{tot}. It is thus given by:

$$\varepsilon = \frac{\rho}{\rho + \beta\ \pi}$$

The percentage effort spent on rework then equals $1 - \varepsilon$, or:

$$\frac{\beta\ \pi}{\rho + \beta\ \pi}$$

It may seem counterintuitive at first that efficiency and productivity are inversely related. Why should increasing productivity reduce efficiency? It is because under a constant defect rate, the number of post-deployment defects increases with output. Therefore, all other parameters remaining same, an increase in productivity results in a higher number of total post-deployment defects, increasing the rework effort, and ultimately decreasing the percentage effort spent on initial development.

Some development techniques allegedly increase productivity while reducing the defect rate at the same time. For example, agile development processes claim to achieve this [3]. If such is the case, simultaneously, a reduction in βπ and an increase in ρ result, and consequently efficiency increases.

UNIT EFFORT :

Unit effort, *UE*, is the total effort, in *compressed* person-hours, required to produce one unit (LOC) of defect-free output. *Compressed* time refers to time excluding interruptions, delays, other overhead, and idle time.

It is calculated by dividing total effort E_{tot} by total output ω corresponding to that output. Expressed in terms of productivity and efficiency, unit effort is given by:

$$UE := \frac{N}{\pi \varepsilon}$$

UNIT TIME:

Unit time, *UT*, is the *compressed elapsed* time, in hours, required to produce one unit (LOC) of defect-free output. *Elapsed* time is measured as the delta between the times of occurrence of two events.

Unit time is calculated by dividing unit effort *UE* by the size of the work unit *N*. Expressed in terms of productivity and efficiency, unit time equals:

$$UT := \frac{1}{\pi \varepsilon}$$

RESULTS OF BASIC COMPARISON MODEL:

Table 1 compares the two abstract models Solo and Pair with respect to the metrics *efficiency, unit effort*, and *unit time*. In each row, the cell in bold typeface indicates the more favorable alternative with respect to the corresponding metric. The model Pair fairs considerably better in all of the three metrics, amounting to nearly 100% improvement in efficiency, over 40% reduction in unit effort, and over 70% reduction in unit time.

Table 1. Comparison of the models Solo and Pair using the base comparison model metrics of efficiency, unit effort, and unit time.

	Solo	Pair
Efficiency (ε) (decimal %)	.172	**.340**
Unit Effort (UE) (person-hours/LOC)	.233	**.135**
Unit Time (UT) (hours/LOC)	.233 (= UE)	**.068**

THE ECONOMIC COMPARISON MODEL

A software project *incurs costs* as it accumulates labor hours and *realizes value* as it delivers new technical functionality for the end users. A project is economically feasible when the total value it creates exceeds the total cost it incurs. We assume that the net value generated depends on four factors: (1) the project's labor cost; (2) the value that the project *earns* commensurate with the output it produces; and (3) the way in which this earned value is recognized, or *realized*, through a specific pattern of deploying the output produced; and (4) the discount rate r used to bring the underlying cash flows to the present time. The economic comparison model takes into account the effect of each of these factors.

LABOR COST:

Programmer labor is often the most important cost driver in a software development project. Let C_{pre}, and C_{post} denote the hourly average labor cost of *initial development* and *rework*, respectively, per person per hour, including salary and benefits. We will assume that initial development and rework are performed by the same work unit, resulting in the same constant value for both variables. Thus:

$$C_{pre} = C_{post} = C$$

We account for labor costs as such costs are incurred, in a similar fashion a business using accrual-based accounting would recognize expenses when they are transacted. However, to avoid choosing an arbitrary period for transacting labor costs, we assume instead that these costs are accrued in a continuous manner as a serious of infinitesimally small transactions.

DISCOUNT RATE:

When the costs and benefits of a project are spread over a long period of time, the economic analysis must take into account, in addition to their magnitude, the specific times at which these costs and benefits are recognized in terms of concrete cash flows. To maximize net economic value, a software project should realize benefits as early as possible and incur costs as late as possible.

We assume that the resulting cash flows are discounted at a fixed continuously compounded rate r from the time of their occurrence relative to the project's start time. The various interpretations of the discount rate – in terms of opportunity cost, time value of money, project risk, minimum required rate of return, or combinations thereof -- is beyond the scope of this paper. We refer the

reader unfamiliar to the standard capital budgeting literature; for example, see Ross [2].

EARNED VALUE:

Earned value (EV) is a well known quantitative project tracking method [7, 8, 20, 21]. With EV tracking, a project's expected outputs or resources are estimated and scheduled for delivery or consumption, respectively. As the project progresses, it earns value relative to this delivery/consumption schedule, so that at completion, the project's earned value equals the total estimated output or the total estimated consumption. For example, for a project with a target to produce 100 units of a product, after having produced 20 units, the project has a current EV of 20. With a target of 200 units, after having produced the same amount of units, it has an EV of 10. In both cases, every unit produced increases the accumulated EV by a fixed amount: by one unit in the former case, and by half a unit in the latter. Let this constant incremental value be denoted by V. Then earned value corresponding to a total output of ω is given by:

$$EV := V\,\omega$$

We refer to V, the value earned by one unit of output, as the *unit value*. In our case, V corresponds to the average currency value of a single line of code, expressed in $/LOC.

According to this model, not every labor hour expended earns value. Effort, such as rework, that does not increase output or result in new technical functionality does not earn any value. Therefore earned value considers rework effort as wasted effort. Consequently, only projects that are 100% efficient earn extra value for each labor hour expended.

VALUE REALIZATION:

In software projects, earned value is not necessarily the same as *realized value*. The distinction between the two is important. Earned value can be seen more as an expression of *potential* value commensurate with effort spent given the productivity level of the development team. That value however may never be realized, for example if the project fails to deliver a useable artifact. Potential value is *realized* when an artifact leaves production and is delivered to its client. This can be accomplished in small increments or in large chunks over the course of a software development project. The rate at which realized value accumulates depends on the frequency with which working code fragments are deployed to the client. Hence although value can be *earned* on a continuous basis, it may not be *realized* until much later.

The concept of realized value may also be explained in reference to the two alternative methods of income recognition of the Generally Accepted Accounting Principles [37]. In cash-based accounting, income from services rendered is recognized when services are paid for, while in accrual-based accounting, income from services are recognized when services are delivered. Thus the concept of realized value admits an accrual-based view of value recognition rather than a cash-based view.

New code is developed, deployed, and reworked in increments of different size, and as such, realized value is accumulated at the same pace as obligations regarding the different size increments are fulfilled. In the economic analysis, we consider two alternative value realization models: *single-point delivery* (value realized at the end) and *incremental delivery* (value realized incrementally on a continuous basis). These two models are located at the opposite extremes of the value realization spectrum. The contrast helps demonstrate the impact of the underlying pattern of value realization on the economic feasibility of a process. Most real projects fall somewhere in between these two theoretical extremes. In contract-based development, the terms of the contract dictate the actual value realization pattern. New contracting models are being put forward with different compensation structures; for example, see Beck and Cleal [5]. The model we use in our analysis can easily be adapted to a particular compensation model.

The Single-Point Delivery Model

With traditional, waterfall-like [28] models of software development, code delivery to the client often occurs in one large chunk. The scope of the project is fixed and finite. Hence, value is realized in a single step at the very end. We will refer to this value realization model as *single-point delivery,* or *deferred realization*. The single-point delivery model is illustrated in Figure 1. The horizontal dimension denotes compressed elapsed time with respect to a single project. Time τ marks the end of the project (completion). It also coincides with the time of the realization of value accumulated over the course of the project. Note that during rework, from τ_{pre} to τ, the project does not earn any extra value.

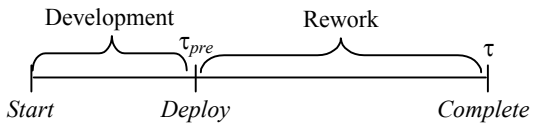

Figure 1: Single-point delivery model of value realization.

Incremental Delivery Model

At the opposite end of the spectrum is the *incremental delivery* model. The scope of the project may not be predetermined, and the responsibility of the work unit may extend to perpetuity. New code is continuously developed, deployed, and reworked in small increments. Development of new code and rework of deployed code are intertwined in a never-ending cycle. Consequently, value is realized in very small increments as micro-obligations involving small chunks of new code are gradually fulfilled.

The generic incremental delivery model is illustrated in Figure 2. Again, the horizontal dimension represents compressed elapsed time. The ticks correspond to deployment points at which the work unit delivers new functional code. In the idealized version of the incremental delivery model, the distance between two subsequent deployment points approaches to zero, resulting to a truly continuous process. We will consider this idealized version only, which we refer to as the *continuous delivery* model.

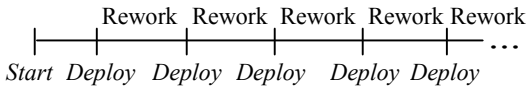

Figure 2: Incremental delivery model of value realization.

EFFICIENCY OF DEFECT DISCOVERY:

A factor that affects value realization is the efficiency of defect discovery. We define a defect as a fault that was *not* discovered *or* removed before deployment, but subsequently is discovered by the client. Alternatively, in an environment where the client is integrated in the development team, defects may be discovered in collaboration with the work unit during the acceptance testing of new code. Defect discovery efficiency involves two components: *latency* and *coverage*.

Latency is the elapsed time between the deployment of a software artifact and the discovery of a fault by the client. Coverage is the number of defects reported or discovered in relation to the total number of defects (including those that have not been discovered).

In practice, the discovery of defects by the client can neither be instantaneous nor complete. For example, Jones [22] states that in large industrial projects, more than half of the runaway defects (post acceptance testing) have a latency of one year, while total coverage four years after deployment hovers around 97%. Thus empirical evidence suggests an

exponential latency model with a half-life of roughly one year with traditional development. In contrast, agile software development processes [4, 17], such as Extreme Programming and SCRUM [26], that rely on short cycles, continuous testing, and frequent client feedback will tend to have a low latency and high coverage.

The economic analysis assumes a perfectly efficient defect discovery process: one with full coverage and zero latency. These idealized conditions are opposing in terms of their impact on net value: while increased coverage tends to decrease net value, increasing latency tends to increase it. When the discount rate is taken into account, these assumptions lead to a conservative overall bias, with a mild tendency to underestimate net value. However the level of underestimation may be different for different processes.

ECONOMIC COMPARISON MODEL

NET PRESENT VALUE:

The *Net Present Value* of a software project can be written as the difference between the *present value* (PV) of the project's benefits and the present value of its costs. This definition is adapted to the current context by representing the benefits in terms of earned value and the costs in terms of labor costs. With this adaptation, NPV becomes very sensitive to changes in the unit value V.

Figure 3 shows how NPV varies as V varies in the neighborhood of 5% to 30% of the unit labor cost C for a pair under single-point delivery. NPV is represented by the vertical axis. The NPV = 0 plane splits the V-Output space into feasible (NPV > 0) and infeasible (NPV < 0) regions. The range of V is chosen to emphasize the behavior of NPV in the neighborhood of this feasibility plane. Note that the slope of the NPV curve changes drastically along the Output axis as V varies. Because of this sensitivity, our interest is not in NPV per se. We need a derived metric whose value can be used to rank two alternatives independent of a particular choice of unit value. *Breakeven Unit Value* meets this need.

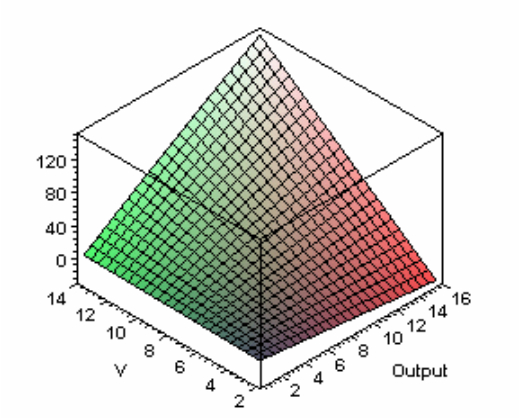

Figure 3: NPV as a function of unit value *V* and output ω for a fixed discount rate *r* = 0.1. Output is plotted in KLOCs. The labor cost *C* is set to 50. *V* varies from 5% to 30% of the labor cost *C*.

BREAKEVEN UNIT VALUE – A RELATIVE RETURN-ON-INVESTMENT METRIC:

Breakeven Unit Value is the threshold value of *V* above which the NPV is positive:

$$\text{BUV} = \min\{\, V \mid NPV \geq 0 \,\}$$

BUV is determined by setting solving the equation NPV = 0 for *V*. Recall that *V* is measured in $/LOC, and represents the fixed increase in earned value per each additional unit of output produced.

A small BUV is better than a large BUV. As BUV increases, a project becomes less and less worthwhile because higher and higher margins are required to move NPV into the feasible region. Thus, we can think of BUV as a relative measure of return on investment.

BREAKEVEN UNIT VALUE RATIO (BUVR):

Using BUV ratios, we can make a one-step comparison between two processes to gauge their relative feasibility. Define BUV Ratio (BUVR) as the ratio of the BUV of model Solo_υ to the BUV of Pair_υ, where υ denotes one of the two value realization models.

$$BUVR = \frac{BUV_{solo}}{BUV_{pair}}$$

Values of BUVR greater than unity indicate an advantage for pairs; values smaller than unity indicate an advantage for soloists. As this ratio increases, the advantage of the pair over the soloist also increases.

The metric BUVR makes the comparison between the two paradigms not only independent of *V*, but also of the hourly labor cost *C*. BUVR depends on:

- the internal parameters of the models Solo and Pair,
- the value realization model υ , and
- the discount rate and the output (applicable only under the single-point delivery model).

SUMMARY OF RESULTS:

Table 2 summarizes the results of the economic comparison. The process models $Solo_{\upsilon}$ and $Pair_{\upsilon}$ are compared under two value realization models with respect to the BUVR metric. The two value realization models considered are the *single-point delivery* model (υ = 1) and the idealized version of the incremental delivery model (or the *continuous delivery* model, $\upsilon = \infty$).

In the table, *r* denotes the discount rate. Projects of higher risk usually require the use a proportionately higher discount rate. Note that we apply the same discount rate for both negative cash flows (costs) and positive cash flows (benefits). In practice, costs and benefits may be subject to different levels and types of risk, possibly warranting the use of different discount rates. A detailed discussion of the relationship between risk, return, and discount rate is beyond the scope of this paper, but can be found in any introductory corporate finance text [27].

Comparison 1 (single-point delivery) depend both on the discount rate and the amount of output produced by the development unit. In general, as the discount rate and output increase, BUVR, hence the advantage of pairs over soloists, increases (with a slightly positive second partial derivative). In comparison 2 (continuous delivery model), the BUVR is constant and greater than unity, representing a steady advantage for pairs.

The limit behaviors are described by the rows "As ω *or r* approaches to infinity" (development continues to perpetuity *or* discount rate is very high) and "As δ approaches to 0". The subsequent row is the range of BUVR for each comparison when both the discount rate and output range from zero to infinity. The final row specifies which model fares better in each case.

Overall, a pair operating under the continuous delivery model (the model $Pair_{\infty}$) yields the lowest (best) BUV since this model combines the improved efficiency and productivity of the pair with the advantage of incremental value

realization. These results highlight the impact of the realization model on the economic analysis.

The findings are sensitive to the three empirical parameters π (productivity), β (defect rate), and ρ (rework speed) to varying degrees. The next to last row of Table 2 summarizes the results of sensitivity analyses. Sensitivity is discussed in more detail later in the paper.

Table 2. Comparison of the models Solo and Pair under different value realization models using BUVR

	Models Compared	
BUVR Behavior (BUVR = BUV_{solo}/BUV_{pair})	**1.** $Solo_1$ to $Pair_1$ (Single-Point Delivery)	**2.** $Solo_\infty$ to $Pair_\infty$ (Continuous Delivery)
As discount rate (*r*) increases:	BUVR increases at an increasing rate	BUVR is constant
As output (ω) increases:	BUVR increases	BUVR is constant
As ω or *r* approaches to infinity:	BUVR approaches to infinity	BUVR remains constant at 1.3
As δ approaches to 0	BUVR approaches to its min. value of 1.5	BUV remains constant at 1.3
Range of BUVR when the empirical parameters are fixed:	[1.5, ∞)	Constant at 1.3
Sensitivity of BUVR to changes in empirical parameters:		
Defect Rate (β):	High	High
Productivity (π):	Low	Low
Rework Speed (ρ):	Medium	Medium
Overall better model	$Pair_1$	$Pair_\infty$

BENEFITS AND COSTS IN SINGLE-POINT DELIVERY:

We now explain the elements of the economic analysis for the single-point delivery model in more detail. When value is realized only at project completion, NPV can be written as:

$$NPV_1 := DRV - TDC_1$$

Here DRV denotes *Deferred Realized Value*, IRV denotes *Incremental Realized Value*, and TDC *denotes Total Discounted Cost*. Each of these parameters is discussed in detail below.

Deferred Realized Value

Deferred Realized Value (DRV) is the accumulated earned value at project completion expressed in *present value* terms. DRV is given by:

$$DRV := V\,\omega\,\mathbf{e}^{(-r\,\tau)}$$

where EV is the earned value, τ is time to completion expressed in compressed elapsed time, and r is the fixed, continuously compounded discount rate. The factor $e^{(-r\tau)}$ brings the deferred EV to the present time.

Expressed in terms of unit value V and output ω, DRV equals:

$$DRV := V\,\omega\,\mathbf{e}^{\left(-\frac{r\,\omega}{\pi\,\varepsilon\,h_y}\right)}$$

where π is the process (work unit) productivity (LOC/hour), ε is the process (work unit) efficiency (unitless), and h_y is the total number of labor hours in a calendar year.

An optimal level of output exists that maximizes the deferred realized value. This level of output is defined by the root of the partial derivative of DRV with respect to ω:

$$\frac{\partial}{\partial\omega}DRV = 0$$

Then maximum DRV is given by:

$$DRV_{max} = \frac{V\,\pi\,\varepsilon\,h_y\,\mathbf{e}^{(-1)}}{r}$$

Note that maximum DRV increases with efficiency, but decreases with discount rate. Since V, h_y, and, r are constant, the maximum DRV ratio of a soloist to a pair is simply given by $(\pi_{solo}\,\varepsilon_{solo})/(\pi_{pair}\varepsilon_{pair}) = UT_{pair}/UT_{solo}$, yielding a constant value of 0.39. This implies that the maximum value realizable under the single-point delivery model by a soloist is less than half the maximum value realizable by a pair. This limit is independent of unit value and discount rate.

Marginal Cost

Marginal cost is the additional cost accumulated by a work unit per unit time of work. For the single-point delivery model, marginal cost before and after deployment will be different if hourly labor cost for initial development and

rework are different. We will calculate a total discounted cost based on this general case, and then use the assumption $C_{pre} = C_{post}$ to simplify the result.

For a project with τ years to completion and a process efficiency of ε, the *Marginal Initial Development Cost* in dollars per year is given by:

$$mC_{pre} := \frac{E\ \varepsilon\ C_{pre}}{\tau} = h_y\ N\ \varepsilon\ C_{pre}$$

Similarly, the *Marginal Rework Cost* in dollars per year is:

$$mC_{post} := \frac{E\ (1 - \varepsilon)\ C_{post}}{\tau} = -h_y\ N\ (-1 + \varepsilon)\ C_{post}$$

Total Discounted Cost can now be calculated from these two components.

Total Discounted Cost

We assume that labor costs are incurred on an ongoing basis as a project progresses. This is a reasonable assumption since corporations incur payroll cash flows in regular discrete installments, for example, on a weekly, bi-weekly, or monthly basis. Labor costs are discounted as they are incurred. For projects with a sufficiently long time horizon, a continuous model is a reasonable frequency-independent approximation to the discrete model in which labor costs are incurred in a periodic manner.

With these assumptions in mind, the *Total Discounted Cost* for the model Solo_1 is obtained by summing marginal costs accumulated over infinitesimally small intervals, both before and after deployment:

$$TDC_1 := \int_0^{\tau_{pre}} mC_{pre}\ e^{(-r\,t)}\,dt + \int_{\tau_{pre}}^{\tau} mC_{post}\ e^{(-r\,t)}\,dt$$

Here $mC_{pre}\,dt$ and $mC_{post}\,dt$ represent initial development and rework costs, respectively, accumulated over a small interval dt in the neighborhood of elapsed time t. The factor $e^{(-rt)}$ brings the small cash flow that occurs over dt to the present time by discounting it over the period t. Here the variable of integration, t, is measured in units of compressed elapsed time.

When $C_{pre} = C_{post} = C$, the sum of the two integrals reduces to:

$$\frac{h_y N C (-2 \varepsilon e^{(-r \tau_{pre})} + \varepsilon - e^{(-r \tau)} + e^{(-r \tau_{pre})} + \varepsilon e^{(-r \tau)})}{r}$$

By substituting

$$\tau = \frac{\omega}{\pi \varepsilon h_y} \qquad \tau_{pre} = \frac{\omega}{\pi h_y}$$

in the above equation based on the relationship between compressed elapsed time τ and output ω, it is possible to express Total Discounted Cost in terms of total output ω and efficiency ε.

BUV in Single-Point Delivery

Under the single-point delivery model, BUV depends on both output (ω) and discount rate (*r*). It increases as either of these variables increases. Figure 4 shows BUV for the model $Pair_1$ (pair under single-point delivery), for a fixed labor cost of *C* = $50/hour. BUV increases with output as well as with discount rate because of deferred value realization under the single-point delivery model, where higher and higher profit margins are required as total time to completion increases.

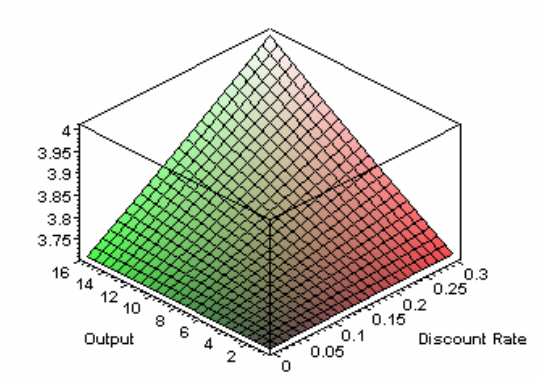

Figure 4: Breakeven Unit Value for the model $Pair_1$ for a fixed hourly labor cost of $50. Output is in KLOCS.

When the discount rate is zero, BUV in the single-point delivery model is given by:

$$\lim_{r \to 0} BUV_1 = \frac{(1 - 2 \varepsilon + 2 \varepsilon^2) N C}{\pi \varepsilon}$$

The limit yields a constant minimum BUV both for a pair and for a soloist.

BUVR in Single-Point Delivery

The economic advantage of pairs over soloists is evident in the single-point delivery model. BUVR is at least 2.24 when the discount rate is zero. In other words, the BUV for a soloist is at least 124% higher than the BUV for a pair. The pair's advantage increases as output or discount rate increases. The effect is illustrated in Figure 5, which plots $Solo_1$ to $Pair_1$ BUVR as a function of total output and discount rate. Pairs accumulate costs faster, but more than compensate for this by realizing value earlier. The larger the project or the higher the discount rate, the more pronounced is the advantage of pairs over soloists.

As can be seen in Figure 5, BUVR is not very sensitive to changes in the discount rate although BUV itself is (Figure 4). Taking the ratio smoothes the impact of discount rate out to a certain degree. For example, even at high values of output (for large projects), a six-fold increase in the discount rate increases the BUVR by less than 19%. Below an output of 5 KLOC (for small projects), BUVR increases by less than 6%.

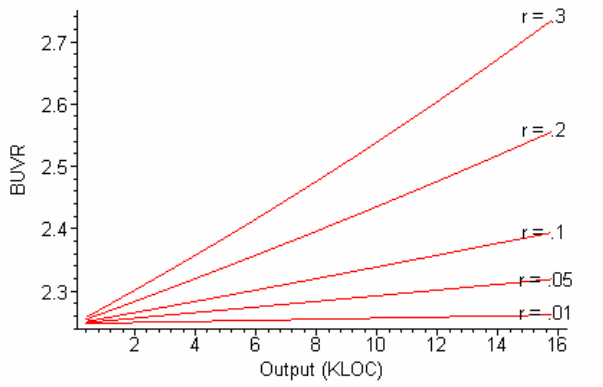

Figure 5: BUVR of the model $Solo_1$ to the model $Pair_1$ as a function of output for different discount rates.

BENEFITS AND COSTS IN CONTINUOUS DELIVERY:

We now explain the elements of the economic analysis model for the continuous delivery model in more detail. For the continuous delivery model, NPV can be expressed as:

$$NPV_{\infty} := IRV - TDC_{\infty}$$

Here IRV denotes *Incremental Realized Value*, and TDC again *denotes Total Discounted Cost*.

Marginal Value Earned

Marginal Value Earned (MVE) is the average value earned per additional unit of elapsed time (measured in $/year, elapsed time is in terms of compressed time). Given a completion or cut-off time of τ, measured in compressed elapsed time, MVE equals:

$$MVE := \frac{EV}{\tau} = \frac{V\,\omega}{\tau}$$

Representing output ω in terms of elapsed time eliminates the variable τ, allowing MVE to be expressed as a function of productivity π and efficiency ε:

$$MVE := V\,\pi\,\varepsilon\,h_y$$

Incrementally Realized Value

Incrementally Realized Value (IRV) is the total value earned over a given time period. Since value realized as earned, it is also discounted as earned. If τ is the time to project completion or the cut-off time, then IRV is given by:

$$IRV := \int_0^{\tau} MVE\ \mathbf{e}^{(-r\,t)}\,dt$$

As usual, the variable of integration, *t*, is measured in compressed elapsed time. Expressed in terms of efficiency ε and productivity π, IRV equals:

$$IRV := -\frac{V\,\pi\,\varepsilon\,h_y\,(\mathbf{e}^{(-r\,\tau)} - 1)}{r}$$

As the cut-off date approaches infinity, IRV asymptotically approaches its maximum value. This limit represents the value of operating a single work unit to perpetuity under constant discount rate. Maximum IRV is given by:

$$IRV_{max} = \frac{V\,\pi\,\varepsilon\,h_y}{r}$$

As with Deferred Realized Value, maximum IRV increases with efficiency and decreases with discount rate.

Pairs achieve a 53% higher maximum IRV than soloists. Since the discount rate (r), the unit value (V), and the number of labor hours in a calendar year (h_y) are the same for both soloists and pairs, the pair-to-soloist ratio of maximum IRV is given directly by the pair-to-soloist ratio of efficiency.

Marginal Cost

Marginal cost was defined as the expected incremental cost of development and rework per additional unit of elapsed time. The same definition remains in effect here, but its computation is slightly different than the one for the single-point delivery model. Since in the continuous delivery model, initial development and rework are intertwined, marginal cost, mC_∞, can be written as:

$$mC_\infty := \frac{E\ \varepsilon\ C_{pre} + E\ (1 - \varepsilon)\ C_{post}}{\tau}$$

where E is the total effort. When $C_{post} = C_{pre} = C$, marginal cost reduces to:

$$mC_\infty = h_y\ N\ C$$

Total Discounted Cost

As is the case in the single-point delivery model, under the continuous delivery model, labor costs are accrued and discounted as they are incurred to calculate the Total Discounted Cost (TDC). If the variable t represents compressed elapsed time, TDC can be written by the following integral:

$$TDC_\infty := \int_0^\tau mC_\infty\ \mathbf{e}^{(-r\,t)}\ dt$$

After substituting the marginal cost with the corresponding term, the above definite integral reduces to:

$$TDC_\infty := -\frac{h_y\ N\ C\ (\mathbf{e}^{(-r\,\tau)} - 1)}{r}$$

Maximum Discounted Cost

The *Maximum Discounted Cost* that a work unit under the continuous delivery model can earn at a constant discount rate is the asymptotic value of TCD incurred by the work unit to perpetuity. This limit is given by:

$$TDC_{\infty,\,max} = \frac{h_y\, N\, C}{r}$$

A pair consistently incurs twice the maximum discounted cost incurred by a soloist. This is because the ratio of maximum TDC is determined solely by N, as the labor cost (C) and the discount rate (r) are assumed to be the same for both models.

BUV in Continuous Delivery

When both value realization and cost accumulation are continuous and incremental, BUV's dependence on output and discount rate is broken in the continuous delivery model. Generically, BUV under the continuous delivery model is given by:

$$BUV_{\infty} := \frac{N\, C}{\pi\, \varepsilon}$$

This yields, for a fixed labor cost of C = \$50/hour, a Breakeven Unit Value of 11.652 for the model Solo_{∞} and 8.96 for the model Pair_{∞}.

BUVR in Continuous Delivery

The BUVR under the continuous delivery model is given by:

$$BUVR_{\infty} = \frac{N_{solo}\, \pi_{pair}\, \varepsilon_{pair}}{N_{pair}\, \pi_{solo}\, \varepsilon_{solo}}$$

The value of BUVR is thus constant at 1.73 under this model of value realization, representing a steady 42% (1 – 1/1.73) advantage for pairs (model Pair_{∞}) over soloists (model Solo_{∞}). Note that this advantage is present regardless of the discount rate and the level of output produced.

SENSITIVITY ANALYSIS

Figures 5 to 7 already illustrated the sensitivity of the BUVR to the two exogenous parameters—namely discount rate (r) and output (ω). The results are summarized in Table 2. In the single-point delivery model, BUVR is mildly sensitive to both discount rate and output. In the continuous delivery model, BUVR is invariant so it is not sensitive to either of these parameters.

What about the three endogenous empirical parameters, productivity (π), rework speed (ρ), and defect rate (β). Since these parameters are descriptive of the development process and the work unit, they warrant further investigation. As summarized in Table 2, BUVR is most sensitive to changes in defect rate, but less so to changes in rework speed, and even less to changes in productivity.

Recall the two pairs of model being compared from Table 2:

- Comparison 1: Solo_1 & Pair_1
- Comparison 2: Solo_∞ & Pair_∞

In the following graphics, the number next to each curve denotes the comparison being made according to the above scheme. We analyze each empirical parameter in order from least to most sensitive. For the production of the graphics, we maintained the discount rate and the output at the arbitrary values of 0.1 (10%) and 7.5 KLOC, respectively. The particular choice of these exogenous parameters only marginally displaces the curves within reasonable ranges, and do not affect the sensitivity results with respect to the three empirical parameters.

We will characterize the sensitivity of BUVR to an empirical parameter over a given range as *insignificant* if the partial derivative of BUVR with respect to the percentage increase in that variable over the range in question is hovering around zero; *mild* if the absolute value of the partial derivative is consistently less than unity, but non-zero; *moderate* if the absolute value is hovering around unity; and *significant* if it is consistently greater than unity. Table 3 summarizes the sensitivity analysis results.

Table 3. BUVR Sensitivity of findings to empirical parameters.

BUVR sensitivity (range specified as percent improvement)	Models Compared: **1.** $Solo_1$ & $Pair_1$	**2.** $Solo_\infty$ & $Pair_\infty$
Productivity (π) over range 0% to 80%	Mild	Mild
Productivity (π) over range 80% to 400%	Insignificant	Mild to Insignificant
Rework speed (ρ) over range 50% to 200%	Moderate	Mild
Rework speed (ρ) over range 200 to 400%	Mild	Insignificant
Defect rate (β) over range 5 to 20%	Moderate	Moderate
Defect rate (β) over range 20 to 85%	Significant	Significant
Defect rate (β) in neighborhood of 85%	Insignificant	Significant

SENSITIVITY TO IMPROVEMENT IN PRODUCTIVITY:

Figure 8 illustrates the sensitivity of the results to the level of improvement in productivity (π) achieved by pairs over soloist. The percentage improvement is expressed relative to the previously adopted productivity value of 25 KLOC/hour for the model Solo. The dotted vertical line marks the benchmark level of improvement achieved by the model Pair according to the University of Utah study.

Overall, BUVR is mildly sensitive to productivity improvements. Both comparisons are initially mildly sensitive to improvements in productivity. The sensitivity decreases as the productivity improvement increases. Around the benchmark level of 74%, the effect is insignificant in comparison 1 and mildly significant in comparison 2.

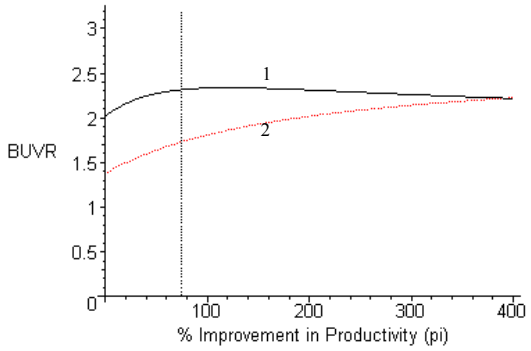

Figure 8: Sensitivity of BUVR to improvements in productivity.

SENSITIVITY TO IMPROVEMENT IN REWORK SPEED:

Figure 9 illustrates the sensitivity of the results to the level of improvement in rework speed (ρ) achieved by pairs over soloist. The percentage improvement is expressed relative to the previously adopted rework speed value of 0.0303 defects/hour for the model Solo. The dotted vertical line again marks the benchmark level of improvement corresponding to the model Pair according to the University of Utah study.

Overall, BUVR is mild to moderately sensitive to rework speed. Both comparisons exhibit a diminishing sensitivity to improvements in rework speed. For comparison 1, a marginal increase in the rework speed of pairs over soloists provide a matching benefit up to an improvement level of around 200%, which we characterize as moderate sensitivity. At the benchmark level of 74%, the effect is moderately sensitive in comparison 1 and mildly sensitive in comparison 2.

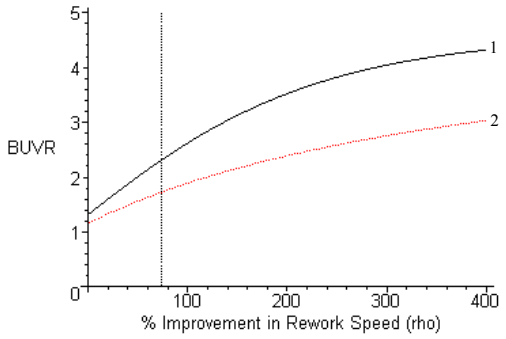

Figure 9: Sensitivity of BUVR to improvements in rework speed.

SENSITIVITY TO IMPROVEMENT IN DEFECT RATE:

Figure 10 illustrates the sensitivity of the results to the level of improvement in defect rate (β) achieved by pairs over soloist. The percentage improvement is expressed relative to the adopted rework speed value of 0.00585 defects/LOC for the model Solo. Unlike in rework speed and productivity, an improvement in defect rate corresponds to smaller, not larger, values of β. A maximum improvement of 100% corresponds to a defect rate of zero. As before, the dotted vertical line marks the benchmark level of improvement corresponding to the original model Pair.

Overall, BUVR is initially moderately sensitive to changes in defect rate, and then becomes increasingly significantly sensitive to it. Around the benchmark level of 60%, the effect is significant. A deviation from this behavior occurs around the 85% improvement neighborhood in comparison 1. There the BUVR peaks and then starts to decline. The peaking effect is attributed to the increasing double labor cost of pairs finally overtaking the diminishing savings from reduced rework effort due to low defect rates. The peaking effect is absent in comparison 2 because of the effect of incremental value realization.

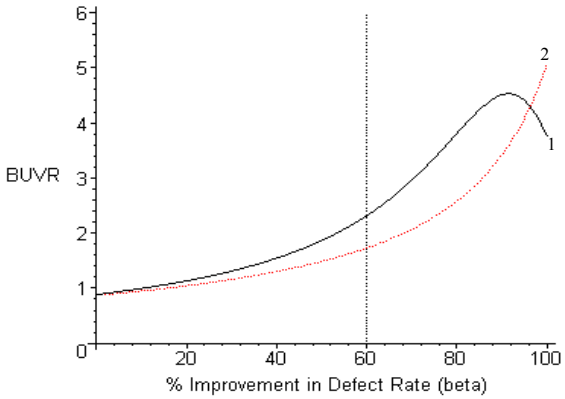

Figure 10: Sensitivity of BUVR to improvements in defect rate.

CONCLUSION AND FUTURE WORK

Quantitative analyses demonstrate the potential of pair programming as an economically viable alternative to individual programming. We compared the two practices under two different value realization models. In each case, we found that pair programming generally creates superior economic value based on data from a previous empirical study and other statistics reported in the general software engineering literature. Although the techniques and concepts employed in the analysis are standard, their use in this particular type of assessment, especially the incorporation of value realization considerations, is novel.

In both practices, net value is maximized when the project realizes value incrementally, for example, through frequent releases. This observation is fully consistent with the general engineering economics intuition that emphasizes early and speedy value realization [7]. The more interesting question that was

addressed by the analysis is how fast a project can afford to spend before the rate of spending overtakes the benefits of early value realization.

Although the findings are not very sensitive to the two exogenous parameters, output and discount rate, they are particularly sensitive to an endogenous parameter, the defect rate. Discount rate was a determining factor between the two practices only in one comparison.

The sensitivity to defect rate is not particularly surprising, since the case for pair programming largely hinges on a significant improvement in defect rate. Our observation confirms that further studies of pair programming should focus on defect rate.

The analyses performed relied on several assumptions. The main ones were the exclusion of pair jelling and other relevant overhead costs, the process efficiency bias due to the particular definition of time to completion, and the zero-latency and full-coverage assumptions regarding defect discovery. These assumptions can be relaxed at the expense of additional model complexity.

The zero-latency defect discovery is not a problematic assumption because post-deployment defects are found by the clients of the deployed code, and not by the developers (the work unit). As a result, defect discovery time is not included in the development effort neither for solo programmers nor for pair programmers. Therefore, even if finding defects becomes more difficult in the field as the number of remaining defects decreases, no bias is introduced by the difference in the post-deployment defect rate between the two practices.

Other considerations include following:

- The disconnect between realized value and business value. We used earned value (expressed in terms of output and unit value) rather than business value, as a proxy for benefits. The inclusion of business value would be meaningful only in concrete contexts because business value strongly depends on project- and market-related factors. Business value can be tackled by redefining realized value independently of earned value, in terms of a separate exogenous parameter. An earlier version of the economic model adopted this view, however, we didn't find it suitable for a generic analysis.

- Treatment of more realistic value realization models that fall between the two extremes discussed. Intermediary models can be tackled by breaking up development along an orthogonal dimension with two components: initial release and subsequent releases. Frequency of subsequent releases can be used as a sensitivity parameter to determine an optimal release cycle. Again, this kind of treatment is most useful if different alternatives are evaluated in a specific, concrete context, rather than for a general comparison.

- More sophisticated characterizations of the empirical models. Sensitivity analyses provide much insight when the empirical models are sufficiently well described by the mean values of the parameters involved. When these parameters are subject to high-levels of variability from one organization or project to another, or even within a single project, and exhibit mutual dependencies, their joint distributional properties become important. If empirical data can be used to infer these properties, probabilistic analyses can be performed at the lowest level. Sensitivity analyses can then be performed at the next level to gauge the impact of errors in the descriptive parameters of the hypothesized distributions.
- Second-order interactions among individual practices. It is possible for the substitution of one practice for another (in this case, pair programming for solo programming) to have unintended, but systematic effects in the rest of the practices shared by the two processes (CSP and PSP). Such second-order interactions could amplify or dampen the observations. Although the University of Utah study did not report effects of this kind, it is still possible that they existed, but were not detected by the study. Second-order interactions are typically complex, subtle, and difficult to detect. For example, pair programmer may be more effective when practiced in conjunction with test-driven development. Future experiments can be designed specifically to reveal these hidden effects.

The potential of pair programming as a viable alternative to traditional solo programming cannot be dismissed on economic grounds. However, in interpreting our findings, the reader should focus on the general behavioral properties of the comparison metrics defined, and not on their specific values. It should be kept in mind that the models used made several assumptions to keep the complexity at a minimum while allowing for meaningful analysis. In addition, the analyses performed relied on existing data of mixed origin, with no independent verification of consistency among the different sources. We remain cautious of portability of these figures since we have no information on the software development methods of the companies involved in those statistics. It would be beneficial to revise the models and repeat the analyses following further experimentation and assessment of pair programming with professional software engineers.

REFERENCES

1. AUER, K. AND MILLER, R., *XP Applied*. Reading, Massachusetts: Addison Wesley, 2001.

2. BASILI, V. R., SHULL, F., AND LANUBILE, F., "Building Knowledge Through Families of Experiments," *IEEE Transactions on Software Engineering*, vol. 25, pp. 456 - 473, 1999.
3. BECK, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.
4. BECK, K., BEEDLE, M., BENNEKUM, A. V., COCKBURN, A., CUNNINGHAM, W., FOWLER, M., GRENNING, J., HIGHSMITH, J., HUNT, A., JEFFRIES, R., KERN, J., MARICK, B., MARTIN, R. C., MELLOR, S., SCHWABER, K., SUTHERLAND, J., AND THOMAS, D., "The Agile Manifesto," pp. http://www.agileAlliance.org, 2001.
5. BECK, K. AND CLEAL, D., "Optional Scope Contracts," *http://www.xprogramming.com/ftp/Optional+scope+contracts.pdf*, 1999.
6. BECK, K. AND FOWLER, M., *Planning Extreme Programming*. Reading, Massachusetts: Addison Wesley, 2001.
7. BOEHM, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
8. CHRISTENSEN, D. S., "The Costs and Benefits of the Earned Value Management Process," *Acquisition Review Quarterly*, vol. Fall, pp. 373-386, 1998.
9. COCKBURN, A. AND WILLIAMS, L., "The Costs and Benefits of Pair Programming," presented at eXtreme Programming and Flexible Processes in Software Engineering – XP2000, Cagliari, Sardinia, Italy, 2000.
10. COCKBURN, A. AND WILLIAMS, L., "The Costs and Benefits of Pair Programming," in *Extreme Programming Examined*, G. Succi and M. Marchesi, Eds. Boston, MA: Addison Wesley, 2001, pp. 223-248.
11. CONSTANTINE, L. L., *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, 1995.
12. COPLIEN, J. O., "A Development Process Generative Pattern Language," in *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt, Ed. Reading, MA: Addison-Wesley, 1995, pp. 183-237.
13. DUTOIT, A. H., BRUEGGE, BERND, "Communication Metrics for Software Development," *IEEE Transactions on Software Engineering*, pp. 615-628, 1998.
14. ERDOGMUS, H., "Comparative evaluation of software development strategies based on Net Present Value," presented at International Conference on Software Engineering Workshop on Economics-Driven Software Engineering, California, 1999.
15. ERDOGMUS, H. AND VANDERGRAAF, J., "Quantitative Approaches for Assessing the Value of COTS-centric Development," presented at Sixth International Symposium on Software Metrics, Boca Raton, FL, 1999.
16. FAVARO, J. M., FAVARO, K. R., AND FAVARO, P. F., "Value-Based Software Reuse Investment," *Annals of Software Engineering*, vol. 5, pp. 5-52, 1998.
17. FOWLER, M., "Put Your Process on a Diet," in *Software Development*, vol. 8, 2000, pp. 32-36.
18. GROSS, N., STEPANEK, M., PORT, O., AND CAREY, J., "Software Hell," in *Business Week*, 1999, pp. 104-118.

19. HAYES, W. AND OVER, J. W., "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," Software Engineering Institute, Pittsburgh, PA CMU/SEI-97-TR-001, December 1997 1997.
20. HUMPHREY, W. S., *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.
21. HUMPHREY, W. S., *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.
22. JONES, C., *Software Quality: Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.
23. LEVY, L. S., *Taming the Tiger: Software Engineering and Software Economics*. New York: Springer-Verlag, 1987.
24. NOSEK, J. T., "The Case for Collaborative Programming," in *Communications of the ACM*, vol. March 1998, 1998, pp. 105-108.
25. PALMIERI, D., "Knowledge Management through Pair Programming Masters Thesis," in *Computer Science*. Raleigh, NC: North Carolina State University, 2002.
26. RISING, L. AND JANOFF, N. S., "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, 2000.
27. ROSS, S. A., *Fundamentals of Corporate Finance*: Irwin/McGraw-Hill, 1996.
28. ROYCE, W. W., "Managing the development of large software systems: concepts and techniques," presented at IEEE WESTCON, Los Angeles, CA, 1970.
29. RUSSELL, G. W., "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, vol. January 1991, pp. 25-31, 1991.
30. SUCCI, G. AND MARCHESI, M., *Extreme Programming Examined*. Boston: Addison Wesley, 2001.
31. WAKE, W. C., *Extreme Programming Explored*. Boston: Addison Wesley, 2001.
32. WIKI, "Programming In Pairs," in *Portland Pattern Repository*, vol. June 29, 1999, 1999, pp. http://c2.com/cgi/wiki?ProgrammingInPairs.
33. WILLIAMS, L. AND KESSLER, R., *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.
34. WILLIAMS, L., KESSLER, R., CUNNINGHAM, W., AND JEFFRIES, R., "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, 2000, pp. 19-25.
35. WILLIAMS, L. A., "The Collaborative Software Process PhD Dissertation," in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.
36. WILLIAMS, L. A. AND KESSLER, R. R., "All I Ever Needed to Know About Pair Programming I Learned in Kindergarten," in *Communications of the ACM*, vol. 43, 2000.
37. DELANEY, P. R., BARRY, J. E., AND NACH, R. *Wiley GAAP 2003: Interpretation and Application of Generally Accepted Accounting Principles*. John Wiley & Sons, 2002.

BIOGRAPHICAL SKETCHES

HAKAN ERDOGMUS (Hakan.Erdogmus@nrc.ca) is a senior research officer with the Institute for Information Technology, National Research Council of Canada. He holds a Master's degree in Computer Science from McGill University, Montreal, and a Ph.D. in Telecommunications from Université du Québec. His current research is in software economics and agile software development, focusing on the evaluation of underlying processes and practices. He delivered several lectures on the economics of agile software development. Dr. Erdogmus is co-editor of *Advances in Software Engineering*, published by Springer.

LAURIE WILLIAMS (williams@csc.ncsu.edu) is an assistant professor of Computer Science at North Carolina State University. She received her undergraduate degree in Industrial Engineering from Lehigh University. She also received an MBA from Duke University and a Ph.D. in Computer Science from the University of Utah. Prior to returning to academia to obtain her Ph.D., she worked in industry, for IBM, for nine years in engineering and software development technical and management positions. She was a founder of the first North American conference on agile software development methodologies, XP Universe/Agile Universe. She is also the author of *Pair Programming Illuminated* and an editor of *Extreme Programming Perspectives*.