

On the Economic Feasibility of Pair Programming

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
williams@csc.ncsu.edu

Hakan Erdogmus
Institute for Information Technology
National Research Council
Ottawa, Canada
Hakan.Erdogmus@nrc.ca

ABSTRACT

This paper provides a summary analysis of pair programming based on the empirical study conducted at the University of Utah [Williams2000]. According to this study, pairs produce higher quality code but that it might take 15% longer to produce this higher quality code. Once code is written, it is tested and ultimately released to the customer. When the savings of putting higher quality code into test and into the hands of the customer are considered, economic analysis shows that this initial 15% cost is more than made up for. We base this conclusion on a break-even metric that takes into account the efficiency and productivity of a pair programmer relative to a solo programmer. Our analysis is valid under a particular model of software development in which the value of the system under development is realized incrementally.

General Terms

Management, Economics

Keywords

Pair programming, agile software development, Extreme Programming, software engineering economics, economic analysis, valuation, net present value, break-even analysis.

1. INTRODUCTION

The economic feasibility of pair programming [Williams+2000a, Williams+2000b] is a key issue. Many instinctively reject pair programming because they think that code development costs will double. If it is more expensive, managers simply will not permit it. Naturally, the goal of a software firm is to be as profitable as possible while providing their customers with the best, high-quality products quickly and cheaply. Organizations decide whether to adopt process improvements based on the business value of their outcome.

First, we perform a basic comparison of solo and pair programming under similar development processes. This basic comparison indicates that pairs perform better considering efficiency and overall productivity. We further this basic comparison by incorporating additional factors into a more complex Net Present Value (*NPV*) analysis. This economic comparison indicates that pair programming should be considered as an economical alternative over solo programming. *NPV* considers the time-value of money: the premise that a dollar today is worth more than a dollar in the future. However, the parameter that captures time-value of money is eliminated in the comparison metric. Economic models based on *NPV* have previously been suggested to evaluate the return on software quality and infrastructure initiatives. Fore examples, see: [Boehm1981, Erdogmus1999, Erdogmus+1999, Favaro+1998, Levy1987].

2. FACTORS OF THE BASIC COMPARISON MODEL

Sound research design guides us to have only one *experimental* variable, the variable under study. Programmers create software using a variety of software development practices. We ensured all programmers in the study used all the same practices (for design, testing, etc.) except for the experimental variable, the *work unit*: *solo programmer* versus *pair of programmers* working in tandem. All solo programmers followed the Personal Software Process (PSP) [Humphrey2000]. Pair programmer followed an extension of PSP that accommodates pairs programming in tandem, without affecting other practices [Williams2000]. In our analysis, we refer to the work unit by *N*:

- *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).

The values of the several other model variables we use are determined based on past research studies and statistics reported in the literature. The chosen values are primarily for illustration purposes. The actual values could be different, and they would most likely be both project- and skill-dependent. However, we believe the general conclusions we make are still sound within reasonable variability of these values.

- π : *new-code productivity* (LOC/person-hour). The average per-person hourly output (LOC) of new code for the work unit.

According to a study by Hayes and Over [Hayes+1997], 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).

Anecdotal evidence [Wiki1999, Auer+2001] suggests that pairing does not take any additional time over solo programming. However, the University of Utah experiment indicated that pairs might take 15% more time than solo programmers. In our analysis, we are conservative and assume that the observed 15% difference is real (despite the fact that this difference was statistically insignificant). With this assumption, in a single person-hour, each developer of a pair produces an average of $25/(1.15) = 22$ LOC. Thus, for $N = 1$ we use $\pi = 25$ LOC/person-hour, and for $N = 2$, π is taken to be a conservative 22 LOC/person-hour.

- β : *defect rate* (defects/LOC). The average number of defects per unit of output (LOC) associated with the work unit.

According to Jones [Jones1997], code produced in the US has an average of 39 raw defects per 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 defect rate of $(39)(0.15) = 6$ defects/KLOC. This figure represents defects that escape to the customer. 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 [Gross+1999]. We adopt the average 6 defects/KLOC (or .006 defects/LOC) as the value of β for $N = 1$ (soloist).

As was discussed in [Williams2000], the code written by the pairs in the experiment passed an average of 90% of the specified acceptance tests compared to code written by soloists, which passed on average only 75% of the same test suite. This result suggests that pairs have only 40% of the defects that solo programmers have after acceptance tests have been run. If we assume that this ratio is retained in defects that escape to the customer, we can adopt $(6)(0.4) = 2.4$ defects/KLOC as the value of β for $N = 2$ (pair).

- ρ : *rework speed* (defects/person-hour). The speed at which defects are fixed by the work unit following the deployment of a piece of code.

A study of a set of industrial software projects from a large telecommunications company [Russel1991] showed that each defect found by a customer 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 [Humphrey1995]. Based on this observation, the rework speed ρ for $N = 1$ (soloist) is taken to be $1/33 = 0.03$ defects/person-hour.

No data is available regarding the effect of pair development on rework activities. We will assume pairs will perform rework with the same 15% cost relative to soloists as was found in the experiment. Under this assumption, the

estimated rework speed ρ for $N = 2$ will be $0.03/1.15 = 0.026$ defects/person-hour.

3. THE BASIC COMPARISON MODEL

We begin by a basic comparison model, which defines two derived metrics: efficiency and overall productivity.

3.1 Efficiency

Efficiency is defined as the percentage effort spent on developing new code relative to the total lifecycle effort (which includes the effort expended on rework).

Given a new-code productivity rate of π , the effort required in person-hours to deploy ω lines of code of output is given by:

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

This quantity specifies the *initial development* (or *pre-deployment*) *effort*, and is followed by *rework* (or *post-deployment*) *effort* once the code has been delivered to the client.

Rework effort, E_{post} , is the maintenance effort expended due to runaway defects after a piece of new code has been deployed.

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

Here $\omega\beta$ is the 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 size N .

Total effort, E_{tot} , is the sum of the initial development and rework efforts. Efficiency, ϵ , is defined the ratio of the initial development effort E_{pre} to the total effort E_{tot} . It is given by:

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

3.2 Overall Productivity

Overall productivity, π_0 , is the average hourly output of defect-free code per programmer hour (assuming that the code is defect-free after rework). It equals the total output ω in LOC divided by the total effort E_{tot} in person-hours. This measure shows just how much the *realized productivity* can be dominated by the extensive cost of rework late in the lifecycle. Expressed in terms of efficiency and new-code productivity, overall productivity is given by:

$$\pi_0 := \pi \epsilon$$

3.3 Results for the Basic Comparison Model

Table 1 compares a soloist to a pair with respect to the metrics *efficiency*, *new-code productivity*, and *overall productivity*. In each row, the cell in bold typeface indicates the more favorable alternative with respect to the corresponding metric. Pairs fair considerably better in efficiency and overall productivity, indicating that initial investments in quality during pair programming pays for itself over the product lifecycle.

Table 1. Comparison of a soloist to a pair.

	Soloist	Pair
Efficiency (ϵ) (decimal %)	.17	.34
New-code productivity (π) (LOC/person-hour)	25	22
Overall productivity (π_0) (LOC/person-hour)	4.3	7.4

4. FACTORS OF THE ECONOMIC COMPARISON MODEL

The economic comparison model is more complex than the basic comparison model and considers additional important factors that determine economic feasibility.

A software project *incurs costs* as it accumulates labor hours and *realizes value* as it delivers functionality. 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 (1) the project’s labor cost, (2) the time-value of money, and (3) the value that the project earns proportionate with the output it produces. The following sections discuss the factors and the underlying parameters that will be used in the advanced analysis.

4.1 Labor Cost

Programmer labor (C) is often the most important cost driver in a software development project. We will assume that initial development and rework are performed by the same work unit, resulting in the same constant value for both variables. Since C is invariant, we will later be able to eliminate it later through a ratio metric.

4.2 Time Value of Money and Present Value

When costs and benefits of a project are spread over time, the time at which the costs are incurred and benefits are realized must be taken into account. A benefit that is expected to be realized in the future is worth less in today’s dollars than a benefit that is realized now. Time value of money captures this basic economic notion. The current worth of a future benefit expressed is referred to as its *present value* (PV).

In economic terms, cost and benefits of a project are represented as negative and positive cash flows, respectively, that occur at specific points in time. *The fundamental implication of time value of money is that a project should earn as fast as possible and spend as slow as possible to generate maximum economic benefit.*

4.3 Earned Value and Unit Value

Earned value (EV) expresses the output produced by a work unit in currency terms based on a linear relationship between development effort and value created. Each unit of new code produced earns a fixed amount of value. We assume that rework effort does earn any value. Only projects that are 100% efficient earn extra value for each labor hour, creating a disincentive to produce defective code, and conversely, an incentive to produce quality code.

Let *unit value*, V , refer to the average value earned by one unit of output, or new code, produced. In our case, V is the average value of a single line of new code. Then, earned value corresponding to an output level of ω lines of code is given by:

$$EV := V \omega$$

4.4 Value Realization

Realized value is value that is realized by the client when a functional, defect-free product has been *delivered* to a customer. Total realized value does not necessarily equal total earned value at a given point in time. For example, earned value may never be realized if a project fails to deliver a functional product to the client.

4.5 The Software Factory Model

For our economic analysis, we consider a value realization model called the *Software Factory*. New code is developed, deployed, and reworked in very small increments. Initial development of new code and rework of deployed code are intertwined in a never-ending cycle. Value is realized in very small increments as micro-chunks of new functionality are gradually delivered. Consequently, earned and realized value coincide in this model. The Software Factory model is illustrated in Figure 2. The ticks represent micro-deployment points where small chunks of new code are delivered incrementally. In the perfect (idealized) version of the Software Factory model, the distance between two deployment points approaches to zero, resulting to a truly continuous process. Agile software development methodologies [Fowler2000] such as XP [Beck+2001] and Scrum [Rising2000] support frequent delivery of working code to customers. Thus agile methodologies are best treaded under the Software Factory model. We will use the idealized model as an approximation to reality in our analysis.

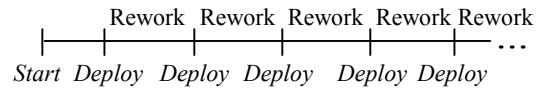


Figure 1: The Software Factory model of value realization.

4.6 Defect Recovery Efficiency

Defect recovery 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 [Jones1997] states that in large industrial projects, more than half of the defects in customer code have a latency of one year, while total coverage four years after deployment hovers around 97%.

In agile methodologies, continuous testing and frequent client feedback are believed to lead to a defect recovery with low latency and high coverage. In our analysis, we assume a perfectly efficient defect recovery process: one with full coverage and zero

latency. These idealized conditions are opposing in terms of their impact on net value generated: while increased coverage tends to decrease net value, increased latency tends to increase it. When time value of money 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 differ from one development process to another.

5. THE ECONOMIC COMPARISON MODEL

5.1 Net Present Value (NPV)

Capital investment decisions are often made based on the concept of *Net Present Value* [Ross1996]. Economic models based on *NPV* have previously been used to evaluate the return on software quality and infrastructure initiatives [Levy1987, Boehm1981, Erdogmus+1999, Favaro+1998].

NPV is the difference between the present value of benefits and the present value of costs:

$$NPV = PV(\text{benefits}) - PV(\text{costs}),$$

where *PV* denotes present value.

A project is thought to have business value when its *Net Present Value*, *NPV*, is positive, and to be unprofitable when its *NPV* is negative. Among a set of possible projects, the one with the highest *NPV* generates the most value, and should be favored over the others.

For the Software Factory model, *NPV* reduces to:

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

Here *IRV* denotes *Incremental Realized Value* and *TDC* denotes *Total Discounted Cost*. We do not discuss these two components of the *NPV* equation for the Software Factory model further in this paper. For the derivation of the quantities *IRV* and *TDC*, refer to Appendix B in [Williams2002].

NPV is very sensitive to changes in *V*, the average value earned by one unit of output produced, measured in \$/LOC. Figure 3 shows how *NPV* varies as *V* and the time horizon of the project increases for a pair operating under the Software Factory model, where the labor cost *C* is fixed at \$50/hour. Note that the slope of the *NPV* curve changes drastically along the output axis as *V* varies.

Due to this sensitivity, our interest is not on *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 and of the constant labor cost *C*. *Breakeven Unit Value* meets the first need.

5.2 Breakeven Unit Value

Breakeven Unit Value (*BUV*) is the threshold value of *V* above which the *NPV* is positive. *BUV* is determined by solving the equation $NPV = 0$ for *V*. Recall that *V* is measured in \$/LOC, based on the assumption that each unit of output produced increases the value earned by a constant amount.

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 turn a profit. In the Software Factory model, *BUV* also makes the economic comparison independent of the particular choice of the discount rate.

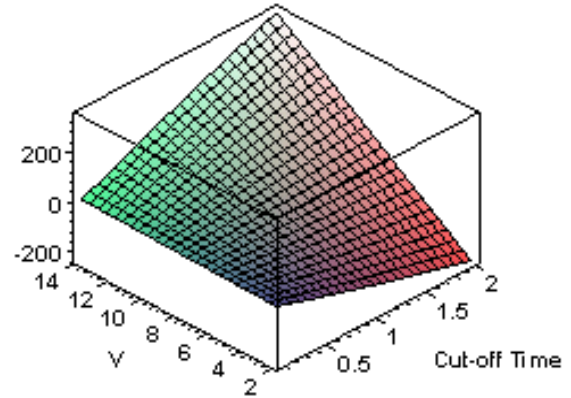


Figure 2: NPV for a pair under the Software Factory model as a function of unit value *V* and cut-off time τ for a fixed annual discount rate of 10%. Output is plotted in KLOCs. Cut-off time, measured in years, represents the time horizon of the project. Labor cost is fixed at $C = \$50/\text{hour}$.

5.3 BUV in the Software Factory

When both value realization and cost accumulation are continuous and incremental, *BUV*'s dependence on both output and discount rate is broken. However, *BUV* remains dependent on *C*, the hourly labor rate.

$$BUV_{\infty} := \frac{C}{\pi \epsilon}$$

For example, for a fixed labor cost of $C = \$50/\text{person-hour}$, a soloist achieves a *BUV* of \$12/LOC while a pair a *BUV* of \$7/LOC, indicating an advantage for the pair. These figures represent the minimum marginal benefit required in the Software Factory for a project to break even.

5.4 Breakeven Unit Value Ratio (BUVR)

A comparison can be made between the return on investment offered by the two alternatives through an examination of the ratio of their *BUVs*.

Define *BUV Ratio* (*BUVR*) as the ratio of the *BUV* of a soloist to the *BUV* a pair.

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

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.

5.5 BUV Ratio in the Software Factory

The *BUV Ratio*, *BUVR*, under the Software Factory model is given by:

$$BUVR_{\infty} = \frac{\pi_{\text{pair}} \epsilon_{\text{pair}}}{\pi_{\text{solo}} \epsilon_{\text{solo}}}$$

Taking the ratio effectively eliminates the invariant *C* that appears in the *BUV* equation. Furthermore, the metric *BUVR* makes the comparison between the two work units of the unit value *V*.

The value of *BUVR* is thus constant at 1.7 representing roughly a 40% advantage for the pairs over the soloist. Note that this advantage is independent of time-value of money, output produced, cut-off time (time horizon of the project), unit value, and labor cost. Effectively, we obtain a metric that provides an all-round comparison.

The *BUVR* metric suggests a clear economic advantage for pairs provided that, everything else being equal, pairs in reality can achieve the suggested relative productivity and efficiency rates.

6. SUMMARY

The results of our analyses demonstrate the potential of pair programming as an economically viable alternative to individual programming. Using the empirical results that demonstrated that pairs produce higher quality code in 15% more calendar time than individuals, we showed that pairs have a higher efficiency and overall productivity rate. Additionally, considering a more complex economic model that departed from Net Present Value, we demonstrated that pairs can increase the business value of a project by significantly reducing the minimum marginal benefit required for a project to break even.

We have not discuss the mathematical derivation of the *BUV* metric in this paper. The interested reader can find it in Appendix B of [Williams2002].

7. REFERENCES

- [Auer+2001] Auer, K. and Miller, R. *XP Applied*. Reading, Massachusetts: Addison Wesley, 2001.
- [Beck2000] Beck, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.
- [Beck+2000] Beck, K. and Fowler, M. *Planning Extreme Programming*. Reading, Massachusetts: Addison Wesley, 2001.
- [Beck+2001] Beck, K., M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The Agile Manifesto," 2001. <http://www.agileAlliance.org>
- [Boehm1981] Boehm, B. W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
- [Erdogmus1999] 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, Los Angeles, California, 1999.
- [Erdogmus+1999] Erdogmus, H. and J. Vandergraaf, "Quantitative Approaches for Assessing the Value of COTS-centric Development," presented at Sixth International Symposium on Software Metrics, Boca Raton, FL, 1999.
- [Favaro+1998] Favaro, J. M., K. R. Favaro, and P. F. Favaro, "Value-Based Software Reuse Investment," *Annals of Software Engineering*, vol. 5, pp. 5-52, 1998.
- [Fowler2000] Fowler, M. "Put Your Process on a Diet," in *Software Development*, vol. 8, 2000, pp. 32-36, 2000.
- [Gross+1999] Gross, N., M. Stepanek, O. Port, and J. Carey "Software Hell," in *Business Week*, 1999, pp. 104-118, 1999.
- [Hayes+1997] Hayes, W. and J. W. Over. "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.
- [Jones1997] Jones, C. *Software Quality: Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.
- [Humphrey1995] Humphrey, W. S. *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.
- [Humphrey2000] Humphrey, W. S. The Personal Software Process (PSP). Technical Report CMU/SEI-2000-TR-022, ESC-TR-2000-022, December 2000.
- [Levy1987] Leby, L. S. *Taming the Tiger: Software Engineering and Software Economics*. New York: Springer-Verlag, 1987.
- [Rising2000] Rising, L. and N. S. Janoff. "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, 2000.
- [Ross1996] Ross, S. A. *Fundamentals of Corporate Finance*. Irwin/McGraw-Hill, 1996.
- [Wiki1999] Wiki, "Programming In Pairs," in *Portland Pattern Repository*, accessed June 29, 1999, <http://c2.com/cgi/wiki?ProgrammingInPairs>.
- [Williams2000] Williams, L. A. *The Collaborative Software Process*. PhD Dissertation in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.
- [Williams+2000a] Williams, L., R. Kessler, W. Cunningham, and R. Jeffries. "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, pp. 19-25, 2000.
- [Williams+2000b] Williams, L. A. and R. Kessler. "All I Ever Needed to Know About Pair Programming I Learned in Kindergarten," in *Communications of the ACM*, vol. 43, 2000.
- [Williams2002] Williams, L. A. and R. Kessler. *Pair Programming Illuminated*. To be published by Addison-Wesley, 2002.