When is Pair Programming Better?

Tim Menzies, Jefferey Smith
Lane Department of Computer Science
West Virginia University
PO Box 6109, Morgantown
WV, 26506-6109, USA;
tim@menzies.us,jefferey@jeffereysmith.com

David Raffo
School of Business Administration
Portland State University
P.O. Box 751, Portland,
OR; 97207-0751, USA
davidr@sba.pdx.edu

Abstract

A growing number of researchers are advocating the use of agile processes (AP) in software development. But to date, there have been few arguments, if any, that would convince the doubters of AP's benefit over conventional development methods. Ideally, one would present detailed empirical observations, but the field is too young. Because of this, we must instead turn to model-based results. In this paper we explore the economic benefits of one of AP's core practices, pair programming. We examine several claims as to why pair programming is beneficial; and specifically, in what situations pair programming proves more beneficial than conventional methods.

1 Introduction

In recent times, there have been many arguments made in favor of agile processes (AP). Noted researchers such as Kent Beck [1], Barry Boehm [2], Alistair Cockburn and Laurie Williams [3], Martin Fowler [5], and many others have shown their support for AP methods. Despite this abundance of arguments however, there has been little empirical proof to convince those in doubt. This is primarily due to the fact that the concept is still relatively new. As a result, one must turn to model-based testing to show AP's advantage over conventional methods.

One such model-based study has been conducted previously. In their paper, "Extreme Programming from an Engineering Economics Viewpoint," Müller and Padberg compare the economics of pair programming (PP) with those of conventional methods [13] (pair programming is one of the mostly commonly seen features in agile processes). This paper reviews and extends that analysis. After running more simulations than Müller and Padberg, and after analyzing the results with a novel sensitivity analysis tool, we can refine the Müller and Padberg conclusions. A precise definition will be offered of exactly where pair programming is advantageous, compared to conventional methods.

We choose to study PP because it is a core part of an agile process called "extreme programming", which is the most widely cited agile process in the literature (§2 offers a brief introduction to agile processes). To do so, we will take a set of equations defining some what is known about the business of PP. The core of our analysis is economic (see §3). This economic view results in our modifications to the Müller and Padberg equations (see §4.1). These equations will be explored using a new sensitivity analysis method called *treatment learning* (see §5.2).

The result of this analysis is a detailed analysis of various claims that have been made as to why PP is superior to conventional methods. It will be shown that situations do exist where PP is clearly the better choice (see §4.4). We will also find a set of threshold points that critically control the benefits of PP.

2 Agile Processes / Extreme Programming

AP is an approach that has gained popularity in recent times. It is "iterative, incremental, self-organizing, and emergent" [15]. According to the Agile Manifesto, AP is based on the idea of "uncovering better ways of developing software by doing it and helping others do it [4]." In his article, "Get Ready for Agile Methods, With Care," Barry Boehm states that the primary difference between AP and conventional methods is that "[AP] methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans [2]." He goes on to discuss that while AP methods risk making mistakes that could have been found by planning practices used by conventional methods, conventional methods risk "that rapid change will make the plans obsolete or very expensive to keep up to date [2]." While AP may not be appropriate in every situation, there appears to

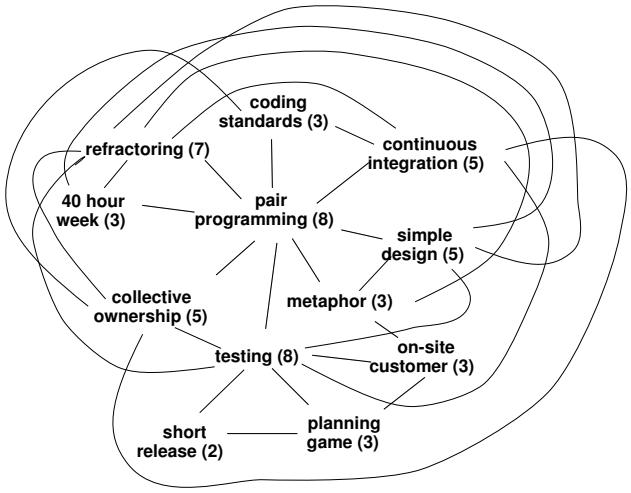

Figure 1. The connection between extreme programming practices. From [1]. Pair programming is shown central in the diagram.

exist cases in which AP may be advantageous to conventional methods.

AP is a collection of software design practices and techniques that diverge from the heavily structured methodologies in favor of a less structured, more adaptive approach. According to its manifesto [4], AP values...

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation;
- Responding to change over following a plan.

There are many positive aspects of AP which might be attractive to software development organizations:

- Continual interaction with the customer helps the project to become more tailored to the customer's needs
- Frequent releases allow the customer to see progress and working code more often
- Pair programming and continual testing leads to fewer errors

A critique of AP is complicated by the broad and diverse nature of the AP movement. However, one of the most popular AP approaches, Kent Beck's *extreme programming*, has been thoroughly specified; and therefore, it is easier to study [1]. According to Beck, extreme programming has the twelve key practices shown in Figure 1. The precise definition of all these practices are beyond scope of this paper. However, note the key role of one of the practices: *pair programming* (PP). PP is the concept of two developers working together at a single machine, designing and writing code cooperatively. Figure 1 shows the connection between XP programming practices. The numbers after each factor in that diagram show how many factors connect to it. Note that pair programming is one the practices with the most number of connections.

AP proponents claim that by working in pairs, developers produce code at a faster rate, and with fewer errors. But the idea of developer pairs leads to two scenarios, each with their own issues:

- **Pool:** If additional developers are added to a project from a pool of developers to create the developer pairs, then does the time/error advantage outweigh the additional developer costs?
- **NoPool:** If additional developers are not added, and instead, the current developers are divided into groups of two, then does the time/error advantage outweigh the additional time needed to write code that results from the fewer number of tasks that can be worked on at one time?

To answer these questions, the economic effects of PP on a software project need to be examined, and compared to the outcome of conventional methods.

3 Measuring benefits using financial measures of performance

Software project managers are increasingly using economic considerations as a guide when making decisions. The financial value of a process decision can be found in many ways including simulation, cost estimation, building analytic models, etc (e.g. see [14]). For each cost or benefit (a.k.a. cash flows), there are four main characteristics to consider: the *magnitude* of the cost or benefit, it's *timing* (i.e. when does it occur), it's *nature* (e.g. one time cash flow, recurring payment, etc.) and the *degree of certainty* of the cash flow. In a basic analysis, costs and benefits are estimated to occur at a fixed point time and with a fixed magnitude. In a more sophisticated analysis, considerations for uncertain magnitudes and timings of cash flows are also dealt with. Some techniques that are often used to assess the financial value of decisions include:

Present value techniques: The present value (PV) for any cash flow simply means discounting the cash flow back to the present time at a specific interest rate (often called *discount rate*).

Net present value (NPV) is the difference between the present values of the benefits less the costs. In a sense, NPV represents the amount of profit that is received from an investment, after the investor is paid back for putting up the money.

Return on Investment (ROI): ROI measures the amount of benefit that is received for every dollar that is invested up front (over and above the original investment). Intuitively, we believe that a ROI of 20% will be preferred to a ROI of 15%. The difficulty with the ROI measure is that this intuition can be deceiving. For example, would we prefer an investment that has a ROI of 20% but only lasts for one year or an investment that has a ROI of 15% that lasts for three years? Another problem with ROI is that for certain complex cash flow scenarios, multiple answers may be obtained. As a result, the ROI measure is used with caution and typically provided in conjunction with other financial measures such as NPV.

Payback Period: Payback period and *discounted payback period* measure the amount of time it takes to return the initial investment. For instance, a payback period of two years means that an investor's initial outlay would be returned in two years and any returns after that would be considered to be pure profit. The discounted payback period may also be considered a present value method in that all cash flows are first discounted back to the present time using a specified interest rate. Again, the discounted payback period would be determined using the discounted cash flows. The problem with the payback period measures is also one of providing incomplete information. For instance, after the initial investment is paid back, how long will the returns continue? What is the overall profitability of this alternative? The technique provides no visibility after the payback period. So, as with the ROI, payback period techniques are usually used in conjunction with Present Value measures such as NPV.

4 The Müller/Padberg Study

4.1 Equations

In their paper, Müller and Padberg present an economic model of software projects for both PP and conventional methods [13]. Their model is based on the Present Value (PV) of the software project, which is calculated and compared for both programming methods.

The PV of the projects is found using Equation 1.

$$PV = \frac{AssetValue}{(1 + DiscountRate)^{\frac{DevTime}{12}}} \quad (1)$$

The $AssetValue$ and $DiscountRate$ are the same for both projects. The $DiscountRate$ is used to simulate time to market: a lower discount rate represents a longer time to market, and a higher discount rate represents a faster time to market.

The equation for the development time, $DevTime$, differs for the two development methods. Since the PP method results in fewer errors, an allowance must be made in the development time for the conventional method to allow for the elimination of an equivalent number of errors. This additional time for quality assurance is the $QATime$. The $DevTime$ equation for the conventional method is:

$$DevTime = QATime + \frac{ProductSize}{DeveloperProductivity * NumberOfDevelopers} \quad (2)$$

And the $DevTime$ equation for the PP method is:

$$DevTime = \frac{1}{DeveloperProductivity} * \frac{1}{NumberOfPairs} * \frac{ProductSize}{(100\% + PairSpeedAdvantage)} \quad (3)$$

where

$$NumberOfPairs = \frac{NumberOfDevelopers}{2} \quad (4)$$

The $QATime$ requires three calculations. First, the number of defects left in a typical project must be calculated using Equation 5.

$$DefectsLeft = ProductSize * \frac{DPKLOC}{1000} * DNE \quad (5)$$

$DPKLOC$ is the number of defects per thousand lines of code, and DNE is the percent of defects not eliminated by conventional techniques (1 - Defects Eliminated by Conventional Processes). Once $DefectsLeft$ has been calculated, the next step is to find the $DefectDifference$. This is simply the $DefectsLeft$ multiplied by the $PairDefectAdvantage$, which is the percent of defects PP eliminates that conventional programming does not.

$$DefectDifference = DefectsLeft * PairDefectAdvantage \quad (6)$$

The final step is to calculate the time required for conventional developers to remove these extra defects. This is known as the $QATime$, and is found using Equation 7.

$$QATime = DefectDifference * \frac{DefectRemovalTime}{DeveloperMonthlyHours} * \frac{1}{NumberOfDevelopers} \quad (7)$$

The final calculation that is required for the project model, which is the same for both the conventional and the PP method, is the development cost, $DevCost$. $DevCost$ is found using Equation 8.

$$DevCost = \frac{DevTime}{12} * (NumberOfDevelopers * DeveloperSalary + ProjectLeadSalary) \quad (8)$$

Parameter	
Developer Productivity	350 LOC/month
Developer Monthly Hours	135 hours/month
Defects per KLOC	100
Defects Eliminated by Conventional Processes	70%
Product Size	16,800 LOC
Developer Salary	$50,000
Project Lead Salary	$60,000
Asset Value	$1,000,000

Table 1.A: **Fixed parameters**

Parameter						
PairSpeedAdvantage	10%	20%	30%	40%	50%	
PairDefectAdvantage	5%	10%	15%	20%	25%	30%
DefectRemovalTime	5h	10h	15h			
DiscountRate	0%	25%	50%	75%	100%	

Table 1.B: **Varied Parameters**

Table 1. Müller and Padberg parameters

4.2 Inputs (Müller/Padberg)

4.3 Method (Müller/Padberg)

For their study, Müller and Padberg used a set of fixed parameters, see Table 1.A, and four parameters, that they considered to be key features, for which they systematically varied their values (Table 1.B).

Note the term *discount rate* in Table 1.B. This rate reflects the time-value of money. A large discount rate means that the passage of time is really valuable either to the potential market value of the product being developed or in terms of the amount of money that must be returned to investors. For example, a 100% discount rate means that the asset looses half of its potential market value every year. Such a discount rate might occur in a very dynamic market with a great deal of competition. On the other hand, a lower discount rate means that a longer time to market has a relatively low cost. For example, a discount rate of 0% would reflect a very stable market with little or no competition and little or no pressure to use the funds for another project. With rapid technology changes and budget constraints within government agencies and private organizations, a realistic discount rate should be greater than 0%.

Müller and Padberg used the values of Tables 1.A and 1.B, plus the equations of Appendix 4.1 to calculate the PV for the conventional method and the PP method. Moreover, they made these calculations for both the Pool and No Pool cases:

- **NoPool:** The number of developers was fixed. In this situation, when the conventional method is using n developers, the PP method is using $\frac{n}{2}$ pairs.
- **Pool:** The number of developers is not fixed, as if there existed pool of developers to create pairs with. In this situation, when the conventional method is using n developers, the PP method is using n pairs, or $2n$ developers.

4.4 Results (Müller/Padberg)

First, Müller and Padberg found that PP is advantageous when the number of pairs is equivalent to the number of developers, as described in the Pool situation. In other words, n developers are more efficient than n/2 pairs. Therefore, as long as there exists additional tasks that can be assigned to individual developers, PP is not the best solution.

However, if the project cannot be subdivided beyond n tasks, and you have access to 2n developers, then PP has the potential to be beneficial in certain cases. Müller and Padberg found that PP is advantageous in this situation when three criteria are met:

1. the project is of small to medium size ($ProductSize$ is not large)
2. the project is of high quality ($AssetValue$ is high)
3. the need for a rapid time to market is present ($DiscountRate$ is high)

5 Our Study

After examining the work done by Müller and Padberg, we felt that a wider range of attributes could be explored. By only varying four parameters, a large number of the model's attributes were not being examined as possible factors in determining the advantages of PP.

5.1 Method (our study)

To correct this, the Müller and Padberg model was reimplemented, and random values within appropriate ranges (see Table 2) were generated for a larger set of attributes (which included the attributes and ranges from the previous study). These attributes were then input into the model and, using Equation 9, the total cost was found for both AP and conventional development methods.

$$TotalCost = \left((1 - \frac{1}{(1 + DiscountRate)^{\frac{DevTime}{12}}}) * AssetValue \right) + DevCost \quad (9)$$

Parameter	Min	Max
PairSpeedAdvantage	-0.5 to 0	0.5 to 1.0
PairDefectAdvantage	5%	30%
DefectRemovalTime (hours)	5	15
DiscountRate	0%	100%
ProductSize (LOC)	10000	250000
DeveloperSalary ($)	45000	65000
ProjectLeaderSalary ($)	60000	90000
AssetValue ($)	200000	2000000
DeveloperProductivity (LOC/month)	100	500
NumberofDevelopers	4	20
DefectsPerKLOC	4	100
DefectsNotEliminated	10%	80%
CommunicationsFactor	1	1.5
Utilization	0.05	1.0

Table 2. Parameters and ranges used by our study

Equation 9 is a modified version of the Müller/Padberg presented value calculation (see Equation 1 in §4.1). This modified equation combines the development cost with the loss in asset value due to the length of time of the development process. The resulting values from Equation 9 represent the total cost for each development method and will always be positive. This then allowed us to use the following ratio without worrying about the effects of negative numbers:

$$\frac{TotalCostofPP}{TotalCostofConventional} \tag{10}$$

This ratio was our means of comparison between the two methods.

The random generation and calculations were repeated for 10,000 cases, the results of each being recorded. These 10,000 cases were then used with a treatment learner to find the attributes that led to an advantage for PP.

Having generated such a large data file, the next task was to reduce it to just the key factors that most influence pair programming. This study was conducted using the **TAR2 treatment learner** to mine the data.

5.2 Treatment Learning

In summary, TAR2 is a tool for performing automatic sensitivity analysis of large data files looking for constraints to parameter ranges that can most *improve* or *degrade* the performance of a system [11, 16]. The tool performs a nearly-exhaustive search over the data, and is capable of discovering treatments not easily found by hand. In this study, *improvement* or *degradation* was measured according to Equation 10; i.e. the impact on the ratio of PP's cost to the cost of conventional methods.

Another way to characterize TAR2 is to call it a *data mining* tool. For a comparison of TAR2 with standard data miners, see [10]. For more on standard data mining tools, see [18].

Many techniques, other than TAR2 have been used previousy for sensitivity analysis. Tradtionally, such an analysis involves an initial *screening* phase tries to reduce the number of variables being studied by some quick early studies. One method used in screening is a divide and conquer technique called *sequential bifurcation* [6]. Using this technique, analysts repeatedly sub-divide the known variables till they find that no major effects in the remaining variables. At each sub-division, some limited simulation studies may be conducted. After screening, another analysis studies the effect of extreme changes to a model such as simulations using minimum and maximum values for each variable or changing the number of connections between components. The results of these extreme simulations may be summarized using mathematical regression. The art of this kind of analysis is to design the least number of extreme experiments to most sample the most number of model variables. After experimenting with extremes, analysts might then explore ranges of each input variable. An *uncertainty analysis* treats each variable as a random variable with a mean and a standard deviation. In uncertainty analysis, monte carlo simulations are conducted to determine the probability of various model outputs.

The above process is labor intensive and can require considerable skill on the part of the analyst. The TAR2 treatment learner was created as an experiment in simplifying sensitivity analysis. TAR2 assumes the *small treatment effect*; i.e. that within a model, a very small number of variables control most of the other variables. This small treatment effect has been reported in many domains, albeit under different names. So much so that Menzies and Cukic [8, 9] and Menzies and Singh [12] speculated that small treatments are an emergent property that will appear in most models.

In models with small treatment sizes, a very fast and simple sensitivity analysis can be performed by TAR2 as follows. Firstly, conduct numerous monte carlo simulations and score each run (via some automatic oracle). Secondly, rank each attribute value by comparing their frequency in high scoring runs to their frequency in low scoring runs. In models with small treatments, a small number of attribute values should get outstandingly large rankings in this step. Thirdly, build treatments by combining a small number of attribute ranges with outstandingly large rankings (a "treatment" is a conjunction of restrictions on the input values that are intended to improve the results of subsequent model outputs). Fourthly, test those treatments by applying them as fresh constraints to a new run of a simulator. The treat-

ments "work" if these constraints do improve the output of simulator. TAR2 automates steps one, two, and three.

Note if a model does not support small treatments then an exponential number of combinations will be required in step three to control a simulation and TAR2 will take too long to execute. That is, TAR2 can *only* work when a domain contains small treatments. The empirical evidence is that TAR2 works in many domains [7, 10, 11], suggesting that small treatments are not uncommon. In the case of our study, all our best treatments used only single attribute values so it was tractable to explore bigger treatments. None of these bigger treatments proved more effective that the smaller treatments. Hence, TAR2 is an adequate method of finding controllers for this domain.

5.3 Using Treatment Learning

The treatment learner was applied to the test cases in two separate studies:

- **S1:** no possible treatments were ignored. That is, no attributes were ignored as possible factors that could be changed to influence the performance of PP.
- **S2:** select attributes that the authors considered to be unchangeable were ignored as possible treatments. These attributes were $PairSpeedAdvantage$, $PairDefectAdvantage$, and $DefectRemovalTime$.

Also, each of these studies were examined in the two situations, **NoPool** and **Pool**, described in section 4.3, for a total set of four studies: **S1/Pool**, **S1/NoPool**, **S2/Pool**, and **S2/NoPool**

In addition to these tests, a set of additional tests were conducted to examine the effects of some specific attributes of PP, and to explore some claims made about PP in the literature. All of these additional tests except **T6** were conducted under the conditions of both **S1/Pool** and **S2/Pool** described above. These additional tests were as follows:

- **T1:** $DeveloperProductivity$ was set to the maximum value, 500 LOC/month, to see the distribution when developers are being the most productive.
- **T2:** $PairSpeedAdvantage$ and $PairDefectAdvantage$ were set to their maximum values shown in the literature, 50% and 30% respectively, to see the distribution when pairs are operating at their greatest rate of advantage over individuals.
- **T3:** $PairSpeedAdvantage$ was allowed to range from 0.1 to 1.0 to see if the outcome would change.
- **T4:** $PairSpeedAdvantage$ was allowed to range from -0.5 to 0.5 to see the effects if pairs sometimes worked slower individuals.
- **T5:** A communications factor was added to the model to explore the idea that the improved communication that results from PP leads to faster development times.
- **T6:** To expand on the concept of dividing tasks among developers and constricting the number of developers available, some additions were made to the model. As with the **NoPool** studies described above, the $NumberOfDevelopers$ was the same for both the conventional and PP methods. Also, a random utilization factor, ranging from 0.5 to 1.0, was generated representing the the percent of developers actually being utilized (ie the number of parallel tasks into which the project could be divided).

5.4 Results (our study)

5.4.1 Summary

A summary of our results from the random testing, **T1**, and **T2** is shown in Figure 2. In those plots, pair programming is at an advantage over conventional approaches when $\frac{Conventioanl}{PP} < 1$.

Plot *a* in Figure 2 shows the output of our model from 10,000 runs across the distributions shown in Table 2. Note that in only 8% of the runs is there an advantage to PP.

Plot *b* in Figure 2 shows the output from **T1**. When $DeveloperProductivity$ was set to the maximum value, PP was advantageous in approximately 20% of the cases.

Plot *c* in Figure 2 shows the output from **T2**. From this plot it can be seen that even when the two attributes that have the most effect on the advantage of PP over conventional methods are at their highest values, still only about 37% of the cases resulted in an advantage for PP. This is the greatest advantage we found, even after using *TAR2* to exhaustively search all the attribute ranges. That is, in order for PP to perform better than conventional, we found that the Pair Speed Advantage needed to have a value of 50% or above. To be best of our knowledge, no claim larger than 50% for Pair Speed Advantage has been made (claims in the literature as to the advantage of PP over conventional methods are greatly varied, PP's development time is said to be anywhere from 50% faster [17] to 15% slower [3] than that of conventional methods). Therefore, we would argue against claims for the superiority of PP over conventional based on the **T2** situation.

The results from **T3** and **T4** revealed nothing new about our model. The treatments found were the same as those found in the previous tests.

5.4.2 Details

In both **S1/Pool** and **S1/NoPool** it was found that increasing $PairSpeedAdvantage$ and $PairDefectAdvantage$

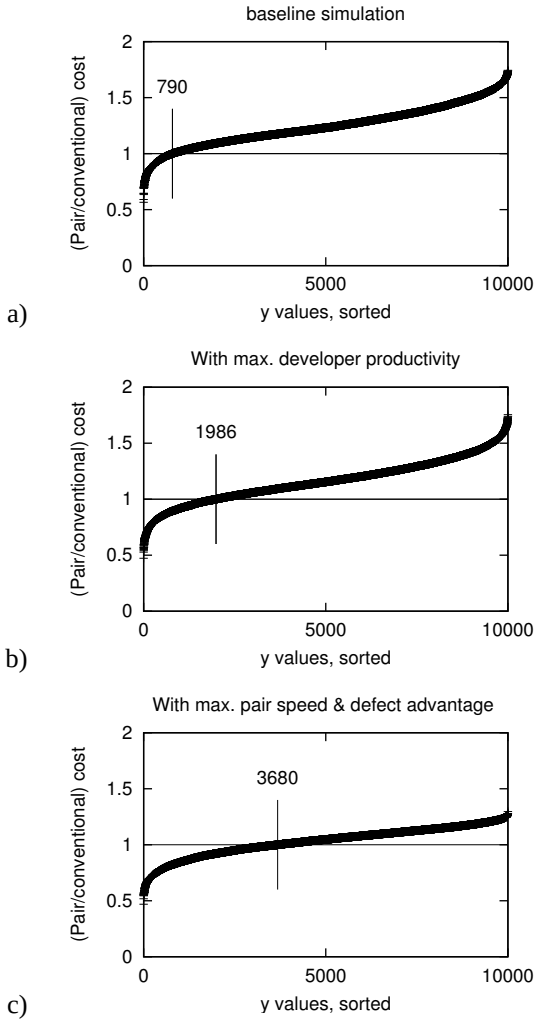

Figure 2. Raw data plots of a) completely random cases, b) cases with $DeveloperProductivity$ set to maximum (**T1**), and c) cases with $PairSpeedAdvantage$ and $PairDefectAdvantage$ set to maximum (**T2**). The vertical line indicates the point where PP is no longer advantageous

was the way to most improve PP's advantage over conventional methods. This is not surprising since $PairSpeedAdvantage$ and $PairDefectAdvantage$ represent advantages that PP has over conventional methods. Another attribute property that showed some influence in increasing the advantage of PP in **S1/NoPool** was a high $DefectRemovalTime$. This too makes sense because conventional methods are prone to more defects, and therefore, a high $DefectRemovalTime$ would result in conventional developers working significantly more than developer pairs.

Considering that it may not be possible to simply

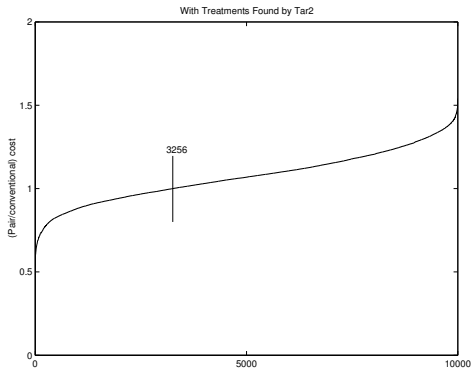

Figure 3. Raw data plot of the outcome when the treatments found by Tar2 were applied to the model

change $PairSpeedAdvantage$, $PairDefectAdvantage$, and $DefectRemovalTime$ as needed, studies **S2/Pool** and **S2/NoPool** ignored these attributes as possible treatments, and looked for additional features that could lead to an advantage for PP. In **S2/Pool** it was found that there were three things that significantly contributed to an advantage of PP over conventional methods:

- A high $DiscountRate$; i.e > 0.42;
- A small $ProductSize$; i.e. 10,000 to 60,000 LOC
- A high $AssetValue$; i.e. $\approx$ \$1.6M to \$2.0M LOC

These are the same three factors that were found by Müller and Padberg when a pool of developers was present. Hence, our pre-experimental concerns that Müller and Padberg study had not surveyed enough of the options was not correct. Nevertheless, even if we arrive at the same conclusions, our study adds much to the rigor of that prior study since we can defend our conclusions from a wider range of criticisms that the original Müller and Padberg study.

Figure 3 shows the distribution when these changes were made to the model. As you can see, in approximately 33% of the runs PP had an advantage over conventional methods. This is a significant improvement over the 8% found in the baseline distribution in Figure 2.

In **S2/NoPool**, no significant treatments were found. Therefore, when a pool of developers is not present, or when additional tasks are present as described by Müller and Padberg in section §4.4, PP does not have an advantage over conventional methods.

In test **T5**, a communications factor was added to explore the idea that PP increases effective communication among developers, thereby reducing development time. To implement this addition, the $DevTime$ variable for conventional programming was replaced by $DevTime * CommunicationsFactor$. Allowing this communications factor to range from 0% to 50%, a significant advantage

for PP was found when the factor was $\geq$ 40%. So if increased communication (or any other factor that is not modelled here) reduces development time by 60% (1 - 40%) or more, then there is a strong case for PP. Note, however, that this is a 60% reduction in development time in addition to the factors already modelled, such as Pair Speed Advantage and Pair Defect Advantage. So the actual advantage of PP over conventional methods must be even greater. With 50% faster being the upper bound we have seen in the literature [17] and the communications factor mentioned above needing to be 60% plus the other factors in the model, it appears that a communications boost alone is not a valid argument for the use of PP. However, this does not count out the possibility of other factors that were not modelled in this experiment that may impact the performance of PP.

The final test, **T6**, was conducted to expand upon the idea of the maximum number of tasks a project can be divided into, and developer utilization. To explore this, a utilization factor, which ranged from 0.05 to 1.0, was added to the model to represent the number of tasks a project could be divided into (i.e. the number of developers or pairs that could be working at once). To implement this change, the $NumberofDevelopers$ variable in all the calculations of the timing values ($DevTime$ and $QATime$) was replaced by $NumberofDevelopers{*}UtilizatonFactor$ (and if this resulted in a number less than one, then a value of one was used). In this test, number of developers was the same for both the conventional and PP methods (as it was in the **NoPool** experiments above). While at first this appears to be a reasonable extension to the model, it is actually unnecessary. When the utilization factor is $\leq$ 0.5, there is no difference from the **Pool** studies above. For example, if there were 20 developers and the utilization factor was 0.25, there would be $20 * 0.25 = 5$ individual developers working with the conventional method, and 5 pairs (i.e. 10 people) with PP. On the other hand, when the utilization factor is $>$ 0.5, the model behaves no differently than the **NoPool** studies above. For example, if there were 20 developers and the utilization factor was 0.75, then there would be $20 * 0.75 = 15$ individuals developers with the conventional method, but only 10 pairs with PP (i.e. 20 people= $min(20, 20 * 1.5)$).

6 Conclusions

Our tests have shown PP's ability to potentially out perform conventional development methods when certain circumstances exist. When the project is relatively small, an abundance of developers exists, and a rapid development time is needed, or when PP significantly increases development time due to improved communication, our study endorses the use of PP.

This is not to say that PP is only advantageous in these specific situations. This study is limited to the components of its model. Constructing a complete economic model of pair programming would be an enormous, if not impossible, task. Because of the vast number of details to be considered, models are limited to looking at a set number of attributes. So from our model, we cannot conclude that a convincing case can be made in favor of PP in every situation based solely on the attributes we examined; i.e.

- increased productivity of pairs over conventional approaches
- productivity vs. cost
- lower error rates

As time passes and more concrete observations from real-world experiences become available, the true benefits of AP/PP will be seen. But until then, proponents of PP must base their case on other factors not modelled in this paper, possibly in conjunction with our results, such as (e.g.) increased performance in rapid changing environments or decreased cost due to conventional requirements reworking to accommodate changes.

Finally, it is important to also reiterate the fact that our study focuses primarily on the AP practice of pair programming. If an AP method does not include pair programming as part of its process, then the results of our study are not applicable.

Acknowledgements

This research was conducted at West Virginia University under NASA contract NCC2-0979 and NCC5-685. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] B. Boehm. Get ready for agile methods. *IEEE Computer*, pages 2–7, 2002.

[3] A. Cockburn and L. Williams. The costs and benefits of pair programming, June 2000.

[4] M. B. et. al. Manifesto for agile software development, 2001. Available from `http://www.agilemanifesto.org/`.

[5] M. Fowler. The new methodology, 2002. Available from `http://martinfowler.com/articles/newMethodology.html`.

[6] J. Kliijnen. Sensitivity analysis and related analyses: a survey of statistical techniques. *Journal Statistical Computation and Simulation*, 57(1–4):111–142, 19987.

[7] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In T. M. Khoshgoftaar, editor, *Software Engineering with Computational Intelligence*. Kluwer, 2003. Available from `http://menzies.us/pdf/02itar2.pdf`.

[8] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. Available from `http://menzies.us/pdf/00ijait.pdf`.

[9] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://menzies.us/pdf/00iesoft.pdf`.

[10] T. Menzies and Y. Hu. Just enough learning (of association rules): The TAR2 treatment learner. In *Journal of Data and Knowledge Engineering (submitted)*, 2002. Available from `http://menzies.us/pdf/02tar2.pdf`.

[11] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

[12] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In *2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February*, 2001. Available from `http://menzies.us/pdf/00maybe.pdf`.

[13] M. M. Mller and F. Padberg. Extreme programming from an engineering economics viewpoint. In *Proceedings of the Fourth International Workshop on Economics-Driven Software Engineering Research (EDSER)*, 2002.

[14] D. M. Raffo, J. V. Vandeville, and R. Martin. Software process simulation to achieve higher cmm levels. *Journal of Systems and Software*, 46(2/3), April 1999.

[15] K. Schwaber, 2002. Quote from the First eWorkshop on Agile Methods. Available from `http://fc-md.umd.edu/projects/Agile/Summary/Summary1.htm`.

[16] T.Menzies and Y. Hu. The TAR2 treatment learner, 2002. Available from `http://www.ece.ubc.ca/twiki/pub/Softeng/TreatmentLearner/intro.pdf`.

[17] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, July/August 2000.

[18] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.