Pair Programming and the Factors Affecting Brooks' Law

Laurie Williams[1], Anuja Shukla[2], Annie I. Antón[1]
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8207

[1]{lawilliams, aianton}@eos.ncsu.edu
[2]anuja@idealake.com

Abstract

Software project managers often add new team members to late software projects with mixed results. Through his law, "adding manpower to a late software project makes it later," Brooks asserts that the assimilation, training and intercommunication costs of adding new team members outweigh the associated team productivity gain in the short term. Our studies show that pair programming reduces these three costs and enables new team members to contribute to a team effort sooner when compared to teams where new team members work alone. Adding manpower to a late project will yield productivity gains to the team more quickly if the team employs the pair programming technique.

1 Introduction

Documented in 1975, Brooks' Law asserts "adding manpower to a late software project makes it later"[7]. Brooks' advice is a debilitating statement to project managers who increase manpower in a desperate attempt to salvage a late project. Brooks stresses this is akin to "dousing a fire with gasoline" [7] because adding manpower increases training costs, assimilation time (time lost learning about project) and intercommunication overhead (work required to communicate, formally or informally, among the team members). Only when increased effective manpower outweighs these costs will the addition of new team members lead to earlier project completion. *Pair programming* [25], the practice whereby two programmers work together at one computer, collaborating on the same algorithm, code or test, has the potential of reducing these three costs. The practice increases software quality.

Conference '00, Month 1-2, 2000, City, State.

In our research, we assessed pair programming as a viable mechanism to reduce these costs. The purpose of this research was to answer three questions: (1) *What is the impact of pair programming on a project's training costs?* (2) *What is the impact of pair programming on the time to assimilate new project team members?* and (3) *What is the impact of pair programming on a project's intercommunication overhead?* To answer these three questions, we administered two surveys to professional software developers to investigate the effects of pair programming on training costs, assimilation time and intercommunication time. We further substantiate these findings with the results of late projects at Motorola and Menlo Innovations.

The remainder of this paper is organized as follows. Section 2 provides an overview of related and relevant work. Section 3 discusses our survey of professional software developers. Sections 4 and 5 present our findings via statistical analysis and mathematical modeling. In Section 6, we share from the results of projects at Motorola and Menlo Innovations that demonstrate the potential of collaborative techniques, particularly pair programming. Finally, Section 7 summarizes our findings and plans for future work.

2 Background and Related Work

This section provides an overview of Brooks' Law, discusses previous analyses of Brooks' Law and discusses pair programming within the context of Brooks' Law.

2.1 Brooks' Law

As previously mentioned, Brooks' Law asserts that, "adding manpower to a late software project makes it later" [7]. Studies of Brooks' Law have focused primarily on the following three factors.

Increased training costs. In the software industry, new team members are generally trained by a mentor who is familiar with all project details and who provides guidance about how to be effective and successful in the organization. During this training time, the mentor must reallocate his or her own valuable time away from his or her own project responsibilities. The cost of the mentor's time varies linearly with the number of new team members.

Increased assimilation time. This is the time required for a new team member (who already possesses the necessary background knowledge and skills) to become an effective, contributing team member. Assimilation time includes the time necessary for the new team member to understand project-specific facts, such as facility layout, policies, procedures, project domain, and the development and test environment. This cost also varies linearly with the number of new team members.

Increased intercommunication overhead. Inter-communication overhead is defined as the average team member's drop in productivity below his nominal productivity as a result of team communication [2]. Intercommunication time includes verbal communication, documentation, and any additional work required for formal or informal communication among team members. This cost increases non-linearly (N^2), as more people are added to a team.

These three factors also form the basis of our own study in which we examined how pair programming can alleviate the effects of training, assimilation, and inter-communication in software development.

2.2 Brooks' Law Examined

Every project manager eventually faces a decision as to how to keep a project on schedule, and adding manpower is invariably a plausible solution. Because of its pertinence in software development efforts that frequently run behind schedule, Brooks' Law has been the subject of several studies by researchers and practitioners.

Brooks' Law is often studied from a system dynamics perspective. We will discuss several such models. These models provide valuable information, yet are limited in their ability to accurately predict the outcome of actual software projects. The models can produce the illusion that project management is deterministic, when in fact no human process, particularly software development, is deterministic. Abdel-Hamid classifies software development as a *multiloop nonlinear feedback system*; these types of systems produce puzzling behavior, requiring more than intuitive judgment alone for estimation [3]. As a result he, Forrester [10] and others have demonstrated the feasibility and utility of studying "computer-based microworlds" to learn about complex social systems such as software development. Several such models will now be discussed.

In 1977, Gordon and Lamb studied Brooks' Law's three factors (training, assimilation and intercommunication). They advocated adding more than the expected number of people to a project early on and then not changing anyone's task/responsibilities until the project is completed [13]. Abdel-Hamid and Madnick developed a quantitative model of project dynamics to study Brooks' Law in 1991. They concluded that adding people to a late project will not always cause it to be late, but will always cause the project to overrun its budget [1, 2]. The Abdel-Hamid and Madnick model spurred the development of other similar models [15, 19, 20]. More recently in 1992, Weinberg argued that training and communication coordination overhead are the two factors that most contribute to Brooks' Law because an increase in both factors manifests itself as added work [23].

In 1994, Stutzke observed that Brooks' Law does not always hold true [22], contending that under certain conditions, manpower can be added to a late project to meet a specified product delivery date (or even accelerate the delivery date). He developed a simple model to determine the conditions under which adding staff will benefit a project and to predict the amount of additional useful effort delivered to a project. Stutzke's model considers (1) the time required for a new team member (who possesses the necessary background knowledge and skills) to be assimilated into the workforce; (2) the time remaining to complete the project; and (3) the amount of mentoring the existing staff must provide to a new team member. This model is detailed in Section 5.

In 1999, Hsia *et al.* argued that Abdel-Hamid and Madnick based their assertion on two unrealistic assumptions: that development tasks can be partitioned without consideration of the sequential constraint (e.g. adding more people will not help if tasks must be completed sequentially) and that when project managers sense a shortage of manpower they will continuously add new people to a project [14]. Consequently, Hsia et al. incorporate the sequential constraint and assume that people are only added to a project once during a project lifecycle. They found that although adding people to a late project always increases its cost, the project may not always be late [14]. According to Hsia *et al.*, every project has a critical point in its schedule, *T,* typically about one-third of the way through the schedule. If enough manpower is added before *T*, the project can still finish before its scheduled deadline. Alternatively, if manpower is added after *T*, then the project will be late.

None of the models discussed above consider the effects of intercommunication overhead or the nontrivial process of repartitioning. Stutzke contends that assessing the effect of intercommunication overhead on project schedule is challenging because communication is a second-order effect; it is therefore not possible to quantify its effects on the project schedule directly [22].

The quantitative models discussed above are used to study project dynamics. Instead, we use these models to study the potential of pair programming to mitigate the costs of bringing new team members into a project; Stutzke's model supports this kind of analysis and serves as the basis for our investigation.

2.3 Pair Programming

As previously mentioned, pair programming [25] refers to the practice whereby two programmers work together at one computer, collaborating on the same algorithm, code,

or test. One of the pair is the *driver* who is actively typing at the computer or recording a design or architecture. The other plays the role of *navigator*. The navigator watches the work of the driver, attentively identifying defects and making suggestions. The two are also continuous brainstorming partners. Pairing with one partner for the entire project duration is beneficial, but may not be optimal. The term *pair rotation* [25] is used to denote when programmers pair with different team members for varying times throughout a project. Ideally, a team member pairs with the person who can most help them on a particular task.

Previous research with senior-level undergraduate students showed that pairs developed higher quality code faster in about half the time as solo programmers [24, 26]. Anecdotes and industry case studies support the results of this study [25]. The higher quality pair-produced code supports Harlan Mills' belief that "programming should be a public process" [7]. Exposing programs to other programmers helps quality control, both because the practice initiates the constant peer pressure to do things well and because peers spot flaws and bugs [7].

Pair programming offers additional benefits, including:

- *Increased Morale.* Pair programmers are more satisfied with their work arrangements [25, 26].
- *Increased Teamwork.* Pair programmers get to know their teammates much better because they actively collaborate [25].
- *Increased Knowledge Transfer*. Pair programmers transfer their knowledge to their partners as a normal part of collaborating [16]. As a result, there are always at least two developers who are knowledgeable about a particular piece of code. With pair rotation, often more than two team members understand a particular piece of code.
- *Enhanced learning.* Pairs continously learn by watching how their partners approach a task, how they use language capabilities, and how they use development tools [25].

3 Examining the Role of Pair Programming in Alleviating the Effects of Brooks' Law

We designed two surveys to investigate the effects of pair programming on training costs, assimilation time and intercommunication time. We administered these surveys to professional software developers. Using the resulting survey data, we tested the following four hypotheses:

H1. Pair programming reduces *training time* when new members are added to a team when compared with teams which practice solo programming.

H2. Pair programming reduces *assimilation time* when new members are added to a team when compared with teams which practice solo programming.

H3. Pair programming reduces *intercommunication time* within a team when compared with teams which practice solo programming.

H4. Through pair programming, software managers can more feasibly add manpower to a late project with beneficial results when compared with teams which practice solo programming.

Two surveys were emailed to 78 individuals who had previously answered a web-based survey compiled as research for *Pair Programming Illuminated* [25]. Additionally, the surveys were posted on an Extreme Programming message board[1]. We sought to obtain responses from project managers as well as senior and mid-level software developers with experience in any programming language or platform. For both Survey I and Survey II, our selection criteria eliminated from analysis any respondents that did not have five years of programming experience in both pair and solo programming environments.

Survey I was related to pair programming and training, whereas Survey II examined intercommunication overhead and pair rotation. Both surveys appear in [21].

3.1 Survey I: Assimilation and Training

The 30 responses received for Survey I met our selection criteria. This survey consisted of two open-ended questions. The questions were designed to test H1 and H2, which hypothesized that pair programming reduces training (H1) and assimilation (H2) time when new members are added to a team.

Respondents were asked to estimate three values (expressed in number of work days) for both assimilation time (*a*) and mentoring time (*m*). The three estimated values were "lowest", "most likely", and "highest". We then determined the average value for *a* and *m* using the PERT formula [18], which is often used for determining best estimates:

*PERT average = (lowest + 4*most likely + highest) / 6*

The estimated values for *a* and *m* were analyzed using a one-sided t-test.

3.1.1 Assimilation time

Survey results demonstrated a mean assimilation time with pairing of 12 workdays and mean assimilation time without pairing of 27 workdays. The one-sided t-test demonstrated that the difference in the mean assimilation time for pairing versus non-pairing was statistically significant ($p < .02$) [21]. *We hypothesized that pair programming reduces assimilation time when a new team member is added to a team. These survey results support this claim.*

3.1.2 Mentoring Time

Survey results demonstrated that the mean percentage of total time spent mentoring with pairing was 26% and the

[1] http://groups.yahoo.com/group/extremeprogramming/

mean percentage of total time spent mentoring without pairing was 37%. The one-sided t-test demonstrated the difference in the mean mentoring time for pairing versus non-pairing was statistically significant ($p < 0.03$) [21]. *We hypothesized that pair programming reduces mentoring time when a new team member is added to a team. Our survey results support this claim.*

3.2 Survey II: Intercommunication and Pair Rotation

Survey II consisted of three questions and we sent it to the same audience as Survey I. The three questions focused on pairing combinations (the mix of experience level of each member of the pair), pair rotation, and intercommunication overhead. We received 35 responses. Of these, 15% were from individuals who had also participated in Survey I. The remaining responses were from new individuals. Only 30 respondents met our survey experience criteria.

3.2.1 Pairing Combinations

In Survey II, respondents indicated the typical kinds of pairing teams when pairing new team members. Figure 1 shows the software development experience level of new team members' pairing partner according to the survey responses [21]. According to the respondents, partners with more than five years of experience are used much more often than inexperienced partners are used. Pairing a new team member with an experienced partner accelerates the new team member's learning curve; however, pairing a new team member with another new team member has been found to be better than leaving that person alone [25].

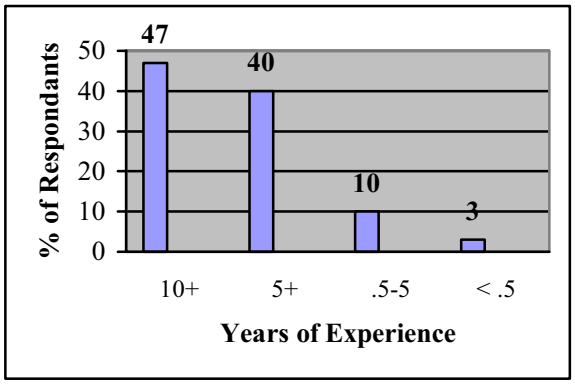

Figure 1: Experience level of new team members' pairing partner

3.2.2 Pairing and Intercommunication Overhead

In any software development project, human communication is an essential component. The intent of the second question in Survey II was to investigate hypothesis H3, which theorizes that pair programming reduces intercommunication overhead. Respondents indicated whether they agreed or disagreed with this hypothesis. Seventy-five percent of the respondents agreed that pair programming reduced intercommunication overhead by making essential, low-level communication informal/on-the-spot (12.5% of the survey respondents refrained from answering the question). Using normal approximation, we observed that the proportion of respondents who indicated that there is a reduction in intercommunication overhead is greater than the proportion that indicated no reduction. This difference is statistically significant ($p < 0.01$) [21].

Respondents also quantified the typical reduction in overhead due to pairing. Interestingly, 18 respondents refrained from answering this question; we surmise that they did not provide quantitative figures because it was difficult for them to estimate the normal amount of time spent on intercommunication. This is consistent with the limitations we observed in other models discussed in Section 2.2. However, qualitatively, 12 respondents stated that the real benefit of pair communication comes from the improvement in the quality, not the quantity, of the communication [21]. *We hypothesized that pair programming reduces intercommunication time within a team. These survey results support this claim.*

3.2.3 Pair Rotation

The next survey question sought to develop an understanding of the teams' use of pair rotation. The results indicate that pair rotation is widely practiced; 94% of developers cited its use. Using normal approximation, we observed the proportion of respondents who do practice pair rotation is greater than the proportion who do not practice pair rotation. This difference is statistically significant ($p<0.01$) [21].

4 Brooks' Modeling of Intercommunication Overhead

In the previous section, survey results supported our hypothesis that pair programming reduces intercommunication cost within development teams. In this section, we further examine this hypothesis with a model using Brooks' original reasoning.

If each part of a project must be separately coordinated with each other part, Brooks states that the total communication effort expended by the team increases as $N * (N-1)/2$, where N represents the number of people on the team [7]. This equation models a team's nonlinear intercommunication overhead and indicates that the effort increases with the square of the number of people on the team [7]. Each team member (N) must communicate with every other team member (N-1) to create one seamless product.

Pair programming enables efficient on-the-spot transfer of information between the developers. When pairs work together, they make decisions on dependencies, technical aspects, and interfaces as they design and implement code. No separate coordination activities need occur to change to dependencies and interfaces. Instead of breaking the project into N parts, it is broken into (N/2) parts and the

maximum communication effort (i.e. if all parts are interrelated) is reduced from N*(N-1)/2 to N*(N-2)/8. Although the communication paths still increase non-linearly (N^2), they increase at a slower rate.

The efficiency of their communication is supported by the fact that pair programmers have been shown to finish their programs in approximately half the elapsed time when compared to solo programmers [24-26]; the industrial experiences in Section 6 provide further substantiation. To reduce intercommunication cost further, we advocate pair rotation with programmers consistently collaborating with the programmer with whom their task is interdependent. *We hypothesized that pair programming reduces the intercommunication required within a team. Our survey results from the previous section and Brooks' intercommunication model support this claim.*

5 Stutzke's Model for Adding Team Members

Mathematical models, such as those discussed in Section 2.2 help estimate the effects of adding new team members to a project that is already behind schedule. These models consider factors such as training and assimilation time before a new team member can make a positive contribution to overall team productivity. They offer several alternatives to adding new team members to meet the project deadline. Stutzke's model, in particular, is especially well suited for demonstrating the impact that pair programming can have on the training and assimilation factors influencing Brooks' Law.

Stutzke performed his research in three steps. First, he developed equations for his model. Second, he administered a survey (similar in content to our Survey I) to obtain values for assimilation time (*a*) and mentoring time (*m*). Other project-specific data collected via the survey provided the data for a case study. Finally, he validated his model through a case study of an actual software development project. The purpose of Stutzke's research was to develop and validate a mathematical expression for Brooks' Law. In our research, we use this validated model to assess the potential of pair programming in aiding software development managers with their late projects. Using the model and our survey results, we examine our hypothesis that through pair programming, software managers can more feasibly add manpower to a late project with beneficial results (H4).

5.1 Applying Stutzke's Mathematical Model

As previously mentioned, Stutzke's model [22] determines the conditions under which adding new team members will benefit a project and predicts the amount of additional useful effort delivered to a project as a result of this additional manpower. Stutzke has demonstrated the model's capabilities within the context of a case study based upon an actual project with a Software Engineering Institute Capability Maturity Model (SEI CMM) Level 3 organization [17]. In the case study, a software development project that had a specified completion date fell behind schedule. Additionally, the project manager knew the remaining work and the net effort needed to finish the project. The manager decided to add more team members to accomplish the required tasks by the specified completion date. The manager needed to predict the total effort and duration needed to complete a project, considering training and assimilation.

The case study validated that Stutzke's research provided a valuable quantitative model for helping the manager to determine the following:

- the useful effort delivered by the total staff as a function of the number of people added;
- the maximum number of new team members that can be added; and
- how late team member size can be increased and still produce a net gain.

In our research, we employed the most relevant parts of Stutzke's model to assess the potential of pair programming within late software projects. We now provide a brief description of these parts of the model.

5.1.1 Notation

Stutzke uses the following notations in his model's equations [22]:

N_i = initial number of staff
N_f = final number of staff
f = fractional increase in staff = $(N_f - N_i) / N_i$
r = remaining time (workdays) to complete the project from the day additional team members arrive
m = training cost i.e. the fraction of staff member's time spent mentoring one new person
a = individual assimilation time i.e. the number of workdays the new team member spends learning the project
a' = the effective assimilation time, which includes the time spent by both the mentor and the trainee = a * (1+m)
E_u = useful effort delivered to project (person days)
E_{gain} = net gain in effort with the augmented staff (person days)

5.1.2 Assumptions

The labour costs are assumed to be the same for both the trainee and mentor, which allows the combining of the trainee and mentor's effort to obtain *a*, the assimilation time. Figure 2 shows the useful effort delivered to the project's tasks before, during and after assimilation takes place.

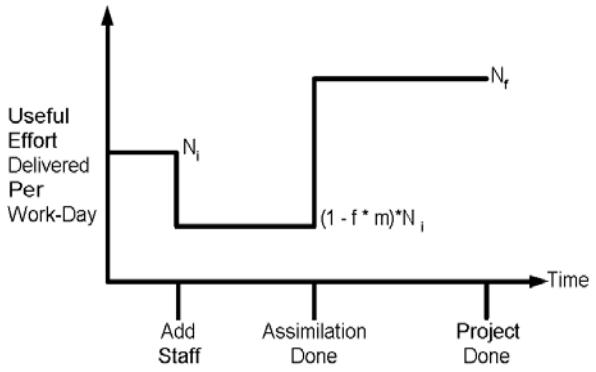

Figure 2: Delivery Rate Versus Time (Adapted from [22])

Before assimilation, $E_u = N_i$.

During assimilation, $E_u = (1 – f * m) * N_i$. This takes into account the proportion of time that some of the initial staff members spend mentoring rather than producing results for the project. Thus, the project will see an initial drop in the rate at which the useful effort is delivered to the project's tasks.

After assimilation, $E_u = N_f$.

Once all the team members are assimilated, useful effort will be delivered at a higher rate. As we observe from Figure 2, there is ultimately an increase in the delivery rate.

5.1.3 Conditions for Net Gain from Adding Team Members

The useful effort delivered to the project is the effort applied to perform the tasks needed to complete the work. We represent this as the sum of work delivered during the mentoring period (duration *a*) and the work delivered after new team member assimilation. The useful effort delivered over the entire time interval *r* is:

$E_u = N_i*(1-f*m)*a+(1+f)*N_i*(r-a)$

$= f * N_i * (r-a')+N_i * r$

[where a'= a * (1+m), as discussed above]

Stutzke's model examines the trade-off between the number of people added to the project and the amount of useful output delivered to the project. Managers should not add staff unless there is a net gain in a team's output after the cost of adding new team members is included. The net gain in effort provided to a project is the difference between the useful effort provided by the augmented staff, E_u, and the original staff's potential total effort:

$E_{gain} = E_u-(N_i * r)$

$= f*N_i*(r-a')+N_i*r- N_i*r$

[substituting E_u from previous equation]

$= f * N_i *(r-a')$

The above equation demonstrates that there is a net gain if $f > 0$ and $r > a'$. The condition $(f > 0)$ means that managers must add some additional team members. The condition $(r > a')$ means that the project must have enough time remaining to assimilate the new team members.

We interpret breakeven graphically from Figure 2; the area under the curve represents the breakeven work delivered (rate * duration). To produce a net gain in effort, the effort provided by the additional staff (after assimilation) must be more than the effort lost during assimilation.

5.2 Pair Programming versus Solo Programming Using Stutzke's Model

We will now use these equations with the values obtained from our statistically significant survey results, as discussed in Section 3. Through our survey, we obtained values for assimilation (*a*), mentoring (*m*), and effective assimilation (*a'*) values as shown in Table 1:

Table 1: Summary of Survey Values

With Pairing	Without Pairing
a = 12 workdays	a = 27 workdays
m = 26%	m = 36%
a' = a (1+m) = 12 * (1+ .26) = 15 workdays	a' = a (1+m) = 27 * (1+ .36) = 36 workdays

These values were then used for the parameters in the case study conducted by Stutzke in [22]. The conditions of the project in the case study were as follows:

- This project was considered to be approximately half complete.
- The effort needed to complete the project could be estimated, since the following were well known: the software architecture, all component modules, their sizes, and the needed modifications.
- There was an initial staff of ten software engineers and testers.
- The time remaining to complete the project on schedule was 5.5 calendar months (about 121 workdays).
- All staff would work on average of 15% overtime.
- The effort remaining was about 20,000 person hours.

Using Stutzke's expressions, we estimate how many new team members need to be added to complete the project as per schedule. The original staff would be able to deliver the following effort during the 5.5 months (121 work days) without any additional help:

$E_i = N_i * (1+ \text{Overtime}) * r * 8$

$= 10 * 1.15 * 121 * 8$

$= 11{,}132$ person hours

The new team members must thus deliver an additional 8,868 person hours (20,000 – 11,132). We refer to this as E_{add}.

The fractional increase in staff is given by *f*. The additional effort (E_{add}) is divided by the overtime effort that will be expended in the remaining days. Using our survey results, we observe that effective assimilation time *a'*

without pairing is 36 workdays and *a'* with pairing is 15 workdays.

$$f = E_{add} / (N_i * (1+ \text{Overtime})* (r\text{-}a')*8))$$
$$= 8868 / (10 * 1.15* (121\text{-}a')*8)$$

f (with pairing) = 0.91
f (without pairing) = 1.13

According to the model, a project manager needs to add 9 new team members if the team practiced pair programming (10 * 0.91 = 9 new team members) to complete the project on time, or 11 new team members if the team did not practice pair programming (10 * 1.13 = 11 new team members). In either case, the model shows that new team members can help complete the project on time. However, the team can save the expense of two new team members if the team utilizes pair programming, at a saving of approximately $56, 000[2].

Our results show that the effective assimilation time and mentoring time is reduced due to pairing. Since assimilation is faster, more work can be achieved in the remaining days because (*r-a'*) is a larger number. The net gain in effort, E_{gain}, is given as $f * N_i * (r\text{-}a')$. This equation implies that breakeven ($E_{gain} = 0$) occurs much earlier because effective assimilation time is reduced with pairing.

We hypothesized that through pair programming, software managers can more feasibly add manpower to a late project with beneficial results. Our findings support this claim.

5.3 Additional Considerations

Producing high quality code is especially important when schedule pressure rises. During these times, planned QA activities are often skipped, allowing defects contained in the work produced to remain undetected [14]. Through pair programming, teams have been shown to produce higher quality code, [24-26] and the results of our study show the technique may be particularly valuable for late projects.

During training time, the model assumes that the trainer's time is lost, consistent with many mentoring models [8]. However, with pairing new team members actually help their mentor during the training process by asking questions and by identifying defects in their work. In this way, new team members are contributing from the very first day [24]. They perform the role of Fagan's "phantom inspector" [9].

Boehm's COnstructive RAD schedule estimation MOdel (CORADMO) [6] considers the effect of collaboration efficiency (CLAB) on software construction costs. The rationale behind the COCOMO CLAB factor is that teams and team members who can collaborate effectively can reduce both effort and schedule [6]. The CLAB factor is positively impacted by high Team Cohesion (TEAM), Multisite Development (SITE) and high Personal Experience (PREX) values. Pair programming teams are likely to attain very high-extra high TEAM values; Boehm describes these teams as being highly cooperative with seamless interactions. Pair programming teams also earn extra high SITE values because they are fully collocated. PREX, defined as the average experience level of the team members, has not been classified for pair programmers.

6 Industrial Projects: On Defying Brooks' Law

We now discuss software development projects at Motorola and Menlo Innovations in which team members practiced collaborative techniques, including pair programming. While these were not case studies of our research, they serve to further substantiate our findings on the potential of pair programming to aid teams in handling late projects.

6.1 Motorola

Figure 3 portrays much of the history of an integration project at Motorola that employed Extreme Programming (XP) [4] for its development paradigm, eventually using all core practices. The task was to integrate their internal Java implementation of Diameter with CMU Monarch Mobile IP. From the start of the project, the team practiced short (2-3 weeks) iterations, the Planning Game, and pair programming. In XP projects, teams measure their progress via project velocity; the Motorola team used an abstract estimation metric, Ideal Engineering Days (IED), to measure user story cards. Their velocity metric was IEDs/day.

Initial estimates suggested the team might have difficulty meeting the targeted deadline with the three people initially assigned to the project. However, the team

[2] A systems analyst II in Research Triangle Park currently makes an average of $57,672 or $231/day. $56,000 is the cost of two systems analysts for 121 days each.

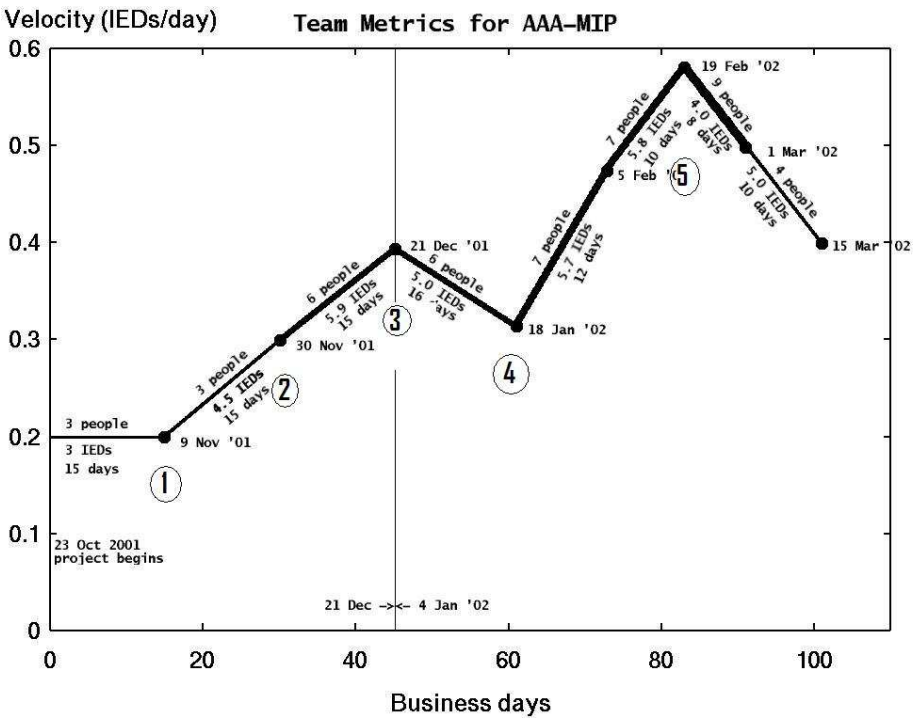

Figure 3: Motorola Project History (Adapted from [27])

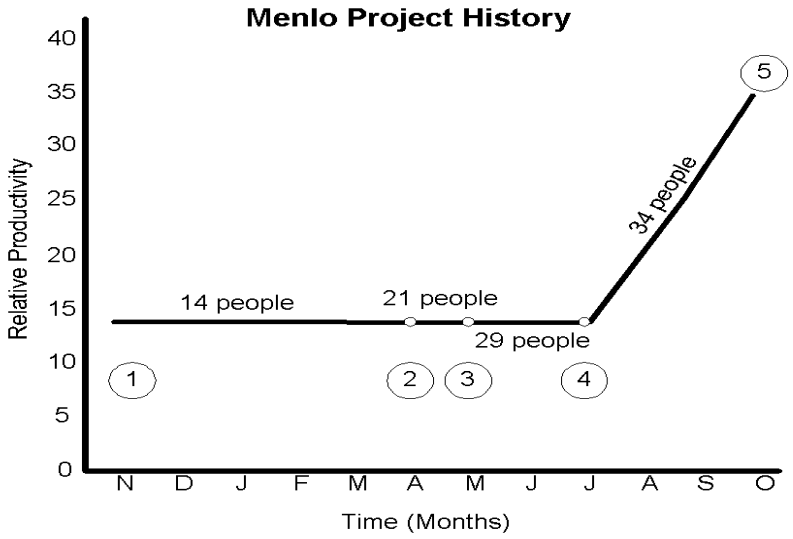

Figure 4: Menlo Project History

waited until the end of the first iteration (Point 1 on Figure 3) to ask for help. Management responded by providing three additional team members, who started at the beginning of the third iteration (Point 2 on Figure 3). Each of the new team members paired with one of the existing team members. The team expected a Brooks' Law slowdown for at least one iteration. They were very pleased that they did not experience a slow down; instead they maintained their initial acceleration. They also added new team members at several other points in the project while maintaining acceleration as shown in Figure 3.

The team believed that the slow-down after the holidays (between Point 3 and Point 4 on Figure 3) was simple post-holiday malaise. There were several changes after the holidays that improved collaboration and communication. They moved their offices from separate cubes to a single shared space. Their customer representative started working with them more often and finally began to sit with them for the fifth iteration (starting at Point 4 on Figure 3). The team's acceleration between Point 4 and Point 5 was almost exactly twice the acceleration experienced from Point 1 to Point 3. During this time, they were practicing pair programming and working in a common collaborative environment shared by a participatory customer.

Halfway through the iteration begun at Point 5, management told the team that the project would end early and that they would remove five people from the project. This accounts for the sharp decline in velocity. The velocity, shown in Figure 3, is only approximate for the last iteration because the team created and destroyed story cards throughout the iteration.

6.2 Menlo Innovations

Similar to the situation at Motorola, the founders of Menlo Innovations faced a similar challenge at a client site [11, 12] as that faced by Motorola. Figure 4 depicts the Menlo Innovations project history. The productivity values shown in this figure illustrate relative productivity per time period and are not exact values. After working on a project with 14 developers for six months using the XP process, the manager was asked to hasten the progress of the project by more than doubling the team size. As shown in Figure 4, in two months (between Points 2 and 4) almost 20 new programmers joined the team. With each addition of new team members, the manager broke apart existing pairs and paired new persons with an existing employee possessing strong mentoring skills. Each two weeks, pairs were reassembled. After just six weeks, new team members mentored programmers just joining the team. This was possible because of the strong network/working relationships formed by pairing in the previous six weeks. The mentoring was further facilitated by an open, collaborative workspace where access to those people and others allowed free flow of questions and answers.

During the three months following the recruiting effort, (between Points 2 and 4) the augmented team continued to produce approximately the same level of results as the earlier team. That is to say, a team of 34 was producing the same amount of work as a team of 14. They were not producing less in spite of the significant cross training and mentoring that was required. During this period, there was a decrease in quality of the work product, but it was detectable and correctable, given strong XP unit testing practices [5]. Then, productivity began to soar, and the team began experiencing a nearly linear increase in results (story cards completed) based on the increase in team size. The team noted that cross-training and mentoring was occurring in two directions as the new team members were bringing as many new skills and experiences to the pairing relationship as those on the original team.

To summarize, both teams faced late projects. The teams were able to more than double their team size without any decline in productivity. Both teams indicated that the pair programming was an essential enabler that prevented them from suffering from the Brooks' Law phenomenon. Their experiences further substantiate our findings that pair programming reduces training, assimilation, and intercommunication costs.

7 Conclusions and Future Work

Software projects continue to be under pressure to deliver quickly. As a result, managers still consider alternatives of how to deal with schedule pressures and late software projects. Our experiences with pair programming led us to believe that pair programming can help in these unfortunate, but unavoidable, situations. Through our research, we examined the potential of the pair programming practice for helping late projects.

We sent two surveys to a wide cross section of professionals in the field of information technology to examine the effects of pair programming on the three factors affecting Brooks' Law. We analyzed 30 survey responses. Using these results, we examined the effects of pairing on Brooks' Law. We employed a mathematical model developed by Stutzke. This model analyzed the process and costs of assimilating new team members, including the costs associated with the diversion of his or her mentors from the project task itself.

We hypothesized that pair programming reduces the intercommunication time within a team. Based on halving N in Brooks' equation N * (N-1)/2, we demonstrated that pair programming reduces intercommunication. Statistically significant survey results fortify these results, indicating that pair programming reduces intercommunication overhead. The reduction is only by a constant factor, however, can represent real cost savings to a development team.

We hypothesized that pair programming reduces the mentoring time when new members are added to a team. The survey results yielded statistically significant evidence that pair programming is effective in reducing the mentoring time.

We hypothesized that pair programming reduces the assimilation time when new members are added to a team. Analysis of the survey results also showed support for the hypothesis that pair programming helps in reducing the assimilation at a statistically significant level.

Lastly, we hypothesized pair programming can make it more feasible to add manpower to a late project with beneficial results. We examined the equations in Stutzke's model to understand the effect of pairing on the total useful effort delivered to the project. Because of the reduction in assimilation and training times, the team was able to

deliver more effort in the time available for project completion.

Through these results, we provide advice to project managers who are dealing with late software projects. Adding manpower to a late project will yield productivity gains to the team more quickly if the team employs the pair programming technique.

In the future, we plan to augment the Brooks' Law models to reflect the effects of intercommunication and the repartitioning of tasks. We are currently collaborating with two companies in Research Triangle Park. These companies are allowing us to monitor their use of various agile software development practices, in particular pair programming. The results of these two case studies will further validate our results and findings to date by enabling us to measure the actual affects of pair programming on training, assimilation, and intercommunication costs.

8 Acknowledgements

Many thanks to La Monte H.P. Yarroll for sharing the Motorola report and for making many suggestions on this paper and to Richard Sheridan of Menlo Innovations for sharing his XP experiences. We also thank: Sarah Doster, Qingfeng He, J. Fernando Naveda, William Stufflebeam, Richard D. Stutzke, and the Fall 2002 NCSU S&RE Reading Group for discussions leading to improvements in this paper. Thanks to Michael Gegick for creating Figures 2 and 4 and Nachiappan Nagappan for the cost savings example.

References

[1] Abdel-Hamid, T., "The Dynamics of Software Projects Staffing: A System Dynamics Based Simulation Approach," *IEEE Transactions on Software Engineering*, vol. 15, pp. 109-119, 1989.

[2] Abdel-Hamid, T. and Madnick, S. E., *Software Project Dynamics: An Integrated Approach*: Prentice Hall, 1991.

[3] Abdel-Hamid, T. K., "The Slippery Path to Productivity Improvement," *IEEE Software*, vol. 13, pp. 43-52, 1996.

[4] Beck, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[5] Beck, K., *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.

[6] Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D., and Steece, B., *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall, 2000.

[7] Brooks, F. P., *The Mythical Man-Month*: Addison-Wesley Publishing Company, 1995.

[8] DeMarco, T., "Human Capital, Unmasked," in *New York Times*. New York, 1996, pp. F13.

[9] Fagan, M. E., "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, vol. 12, 1986.

[10] Forrester, J., *System Dynamics and the Lessons of 35 Years*. Cambridge, Mass.: MIT, 1991.

[11] Goebel, C. J., "Extreme Programming Used to Establish the Culture of a High Performance Team," *http://www.menloinstitute.com/freestuff/whitepapers/Management%20Case%20Final.pdf*, 2002.

[12] Goebel, C. J., Meloche, T., and Sheridan, R., "Extreme Interviewing," *http://www.menloinstitute.com/freestuff/whitepapers/Extreme%20Interviewing%20Final.pdf*, 2002.

[13] Gordon, R. L. and Lamb, J. C., "A Close Look at Brooks' Law," *Datamation*, vol. June, pp. 81-86, 1977.

[14] Hsia, P., Hsu, C., and Kung, D. C., "Brooks' Law Revisited: A System Dynamics Approach," presented at Twenty-Third Annual International Computer Software and Applications Conference (COMPSAC '99), 1999.

[15] Kellner, M. I., Madachy, R. J., and Raffo, D. M., "Software process simulation modeling: why? what? how?," *Journal of System and Software*, vol. 46, pp. 91-105, 1999.

[16] Palmieri, D., "Knowledge Management through Pair Programming," in *Computer Science*. Raleigh, NC: North Carolina State University, 2002.

[17] Paulk, M. C., Curtis, B., and Chrisis, M. B., "Capability Maturity Model for Software Version 1.1," Software Engineering Institute CMU/SEI-93-TR, February 24, 1993 1993.

[18] Riggs, J., *Production Systems Planning, Analysis and Control*, 3 ed: Wiley, 1981.

[19] Rodrigues, A. G. and Williams, T. M., "System dynamics in software project management: towards the development of a formal integrated approach," *European Journal on Information Systems*, vol. 6, pp. 51-56, 1997.

[20] Ruiz, M., Ramos, I., and Toro, M., "A simplified model of software project dynamics," *Journal of Systems and Software*, vol. 59, pp. 299-309, 2001.

[21] Shukla, A., "Pair Programming and the Factors Affecting Brooks' Law Master's Thesis," in *Computer Science*. Raleigh: North Carolina State University, 2002.

[22] Stutzke, R. D., "A Mathematical Expression of Brooks' Law," presented at Ninth International Forum on COCOMO and Cost Modeling, Los Angeles, CA, 1994.

[23] Weinberg, G. M., *Quality Software Management: System Thinking*, vol. I: Dorset House Publishing, 1992.

[24] Williams, L., Kessler, R., Cunningham, W., and Jeffries, R., "Strengthening the Case for Pair-Programming," in *IEEE Software*, vol. 17, no. 4, July/August 2000, pp. 19-25.

[25] Williams, L. and Kessler, R., *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.

[26] Williams, L. A., "The Collaborative Software Process PhD Dissertation," in *Department of Computer Science*. Salt Lake City, UT: University of Utah, 2000.

[27] Yarroll, L. M. and Hari, S., "Extreme Programming Practices Promote Teamwork," presented at Motorola System Engineering Symposium, Schaumburg, IL, 2002.