chapter four

Overcoming Management Resistance to Pair Programming

Software development managers often have the knee-jerk reaction "Why in the world would I pay two programmers to do something that one programmer could do?" This is not a surprising reaction.

In this chapter, we're talking to two distinct audiences. First, we're talking to programmers who would really like to get their team into pair programming. We'll give you some "ammunition" as you appeal to your manager's motivations. Second, we're talking to the managers who are trying to decide whether to give it a go. We aim to give you some data that you can use in making that decision.

There are also two levels of pair programming: One is a very casual, noninvasive use of the practice. You and your peer work together a lot, and you both manage to get your job done. You seek out someone to work with you when you have something difficult to do. You offer to help your peers when they have a tough time. We're not going to address this casual use of pair programming in this chapter.

Rather, we're addressing the prevalent use of pair programming throughout a team, where pairing is an integral part of the team dynamics and the team development practices—the kind of use that a manager would generally be aware of. Hopefully, the manager strives to have an "I don't care how you do it, just do it" attitude. But, we know that at the end of the day each manager has a set of objectives that needs to be met so the manager can be rela-

tively assured of being able to feed his or her family. Managers need to know that pairing will help them achieve their objectives.

You might be concerned that no matter what we say, it will double the amount of time you spend on a task, and you can't afford that risk. To you, our recommendation is to have a very small, minimal-risk pilot. Laurie was working with a development team at IBM. She had already told them much of what is included in this chapter. Then she had a meeting with the manager and six of the employees in the department who were interested in trying pair programming. When it really came down to getting started, all were nervous. She encouraged them to pick only two from among them to pair program for only one week. They would treat this as a pilot and see how it went.

This was a minimal-risk pilot that couldn't set a schedule too far back, one person-week at most. By the second day of the pilot, they e-mailed, "It's incredible how much we're teaching each other about our areas of the project." At the end of the week, they had completed a task they estimated at 90 hours in only 60 hours, so they did it in less, not more, time by pairing. This module went on to test with only two reported test defects—one of them occurred during the 10 hours that one of the pair worked alone. One of the programmers commented on the defects, "They could have been caught by [partner] and me during our initial design, code, and debugging, but they weren't because of our inexperience with the function we were providing. They could have been caught by the formal code review we had and fixed before testing found them, but they weren't. No one at the code review asked questions about the areas of the code where the defects occurred."

Pilot experiences like these can give a team more confidence in trying pair programming on a larger scale.

Motivations

When she worked at IBM, Laurie used to teach a class called "Increasing Human Effectiveness." (The Edge Learning Institute developed the course.) Throughout the course, we learned that whenever you set a goal, you should also write down "What's In It For Me" (WIIFM). The only way you'll change your behavior to reach your goal is if there's something in it for you, something you personally gain by changing the behavior. So we consider the goals

of you and/or your manager to appeal to the WIIFM thoughts. What are those goals?

1. I want to complete my projects on time with high-quality code.
2. I want to reduce my risk of losing a key person.
3. I want my employees to be happy.
4. I want to reduce the amount of time it takes to train a new person.
5. I want my teams to work well together and to communicate more effectively and efficiently with each other.

Throughout the rest of the chapter, we'll explain our findings in each of these areas. Some of our results are quantitative and some are qualitative.

Goal: I want to complete my projects on time with high-quality code.

WIIFM: My bosses are happier because our business objectives are met or exceeded and because we have improved customer satisfaction.

We need to build the case that the use of pair programming can help managers complete their projects on time and with high quality, a case that says that pair programming is economical. Here are some findings.

In 1996, there was a report from Hill Air Force Base:

> The two-person team approach places two engineers or two programmers in the same location (office, cubicle, etc.) with one workstation and one problem to solve. The team is not allowed to divide the task but produces the design, code, and documentation as if the team was a single individual. . . . Final project results were outstanding. Total productivity was 175 lines per person-month (lppm) compared to a documented average individual productivity of only 77 lppm. This result is especially striking when we consider two persons produced each line of source code. The error rate through software-system integration was three orders of magnitude lower than the organization's norm. Was the project a fluke? No. Why were the results so impressive? A brief list of observed phenomena includes focused energy, brainstorming, problem solving, continuous design and code walkthroughs, mentoring, and motivation. (Jensen 1996)

In 1998, Temple University Professor Nosek reported on his study of 15 full-time, experienced programmers working for 45 minutes on a challenging problem, important to their organization, in their own environment and with their own equipment. Five worked individually; ten worked collaboratively in five pairs. Conditions and materials used were the same for both the experimental (team) and control (individual) groups. This study provided statistically significant results, using a two-sided t-test. "To the surprise of the managers and participants, all the teams outperformed the individual programmers, enjoyed the problem-solving process more, and had greater confidence in their solutions." Combining their time, the pairs spent 60 percent more time on the task. However, because they worked in tandem, they were able to complete the task in less "wall-clock" time, in addition to producing better algorithms and code in less time. The majority of the programmers were initially skeptical of the value of collaboration in working on the same problem and thought it would not be an enjoyable process. However, results show collaboration improved both their performance and their enjoyment of the problem-solving process (Nosek 1998).

After reading these results, we decided to do an experiment of our own, one that ultimately produced groundbreaking results. In 1999 at the University of Utah, students in the Senior Software Engineering course participated in a structured experiment. The students were aware of the importance of the experiment, the need to keep accurate information, and the importance of each person (whether in the control group or the experimental group) to the outcome. All students attended the same classes, received the same instruction, and participated in class discussions on the pros and cons of pair programming. When asked on the first day of class, 35 of the 41 students (85 percent) indicated a preference for pair programming. (Later, many of that 85 percent admitted they were initially reluctant, but curious, about pair programming.) The students also understood that grades would be curved in separate pair/individual groupings, so that no one had to be concerned that the working arrangement would affect his or her final grade.

The students were divided into two groups; both groups were deliberately comprised of the same mix of high, average, and low performers. Thirteen students formed the control group in which all the students worked individually on all assignments. Twenty-eight students formed the experimental group in which all worked in two-person collaborative teams; collaboratively, they com-

pleted the same assignments as the individuals. (The collaborative pairs also did additional assignments to keep the overall workload the same between the two groups.) All 28 students in the experimental group had expressed an interest in pair programming, and some of the students in the control group had wanted to try pair programming. It is important to note that prior to enrolling in this class, students had had significant coding practice. Most students had had industry/internship experience and had written small compilers, operating system kernels, and interpreters in other classes.

Cycle time, productivity, and quality results were compared between the two groups. Students recorded information in a Web-based tool about the time they spent on each project. Quality was measured by the results of automated testing executed by an impartial teaching assistant.

As reported (Cockburn and Williams 2000; Williams et al. 2000; Williams 2000), our experimental class produced quantitative results supporting the pair programming results in industry. The students completed four assignments over a period of six weeks. Thirteen individuals and fourteen collaborative pairs completed each assignment. The pairs passed, on average, 15 percent more of the automated postdevelopment test cases (see Table 4.1). The difference in quality levels is statistically significant.

The pair results were also more consistent, while the individuals varied more about the mean. Individuals intermittently didn't hand in a program or handed it in late; pairs handed in their assignments on time. This result can be attributed to "Pair Pressure" as discussed in Chapter 3. The programmers admitted to working harder and smarter on programs because they did not want to let their partner down. Individuals did not have this form of pressure and did not perform as consistently.

Table 4-1 Percentage of Test Cases Passed

	Individuals	**Collaborative Teams**
Program 1	73.4%	86.4%
Program 2	78.1%	88.6%
Program 3	70.4%	87.1%
Program 4	78.1%	94.4%

Not only did pairs write programs that were of higher externally visible quality (for example, they passed more test cases), but their programs were consistently more than 20 percent shorter than their individual counterparts. Implementing functionality in fewer lines of code is commonly viewed as an indication of better design quality and lower projected maintenance costs (Boehm 1981). The individuals were more likely to produce "blob class" (Brown et al. 1998) designs—just to get the job done. The design from the pairs exploited more of the benefits of object-oriented programming. Their classes demonstrated more encapsulation and had more cohesive classes with better class-responsibility alignment. The individuals tended to have fewer classes that had many responsibilities, which would probably be more difficult to enhance and/or maintain.

Lots of nagging, low-severity software defects can become a big annoyance to customers. Even one high-severity software defect can come with a very high price tag for customers. William Malik, research director at Gartner, reports that 40 percent of all system crashes are software related. Hardware problems cause system outages of a few seconds to a few minutes, while software crashes typically bring down networks for several hours to several days. Some businesses, such as the New York Stock Exchange, can lose anywhere from $10,000 to several million dollars a minute when networks go down, which often happens due to software problems. Alan MacCormack, Harvard Business School professor, points out that systems are so complex now that they are impossible to test fully (Krantz and Iwata June 11, 2001). As a result, we must build quality into our products. Pair programming can help us do this.

The other piece of the economic equation is time. The gut reaction of many people is to reject the idea of pair programming because they assume there will be a 100 percent programmer-hour increase by putting two programmers on a job that one can do. If pair programming does, indeed, double the time, it certainly would be difficult to justify, even given the dramatic, expensive effects of poor quality just discussed. The University of Utah students recorded how much time they spent on their assignments via a Web-based data recording and information retrieval system. During an initial adjustment period in the first program (the "jelling" assignment, which took approximately ten hours), the pairs spent approximately 60 percent more person-hours to complete the program. Thereafter, the pairs spent on average only 15 percent more than the individuals, which was no longer statistically

significant, because the average was driven up by two of the 13 pairs. The median amount of time spent by the individuals and the pairs was essentially identical. As a side note, the two pairs that spent the most time also spent the most time when they completed pre- and postexperiment programs individually. (Williams 2000; Cockburn and Williams 2000; Williams et al. 2000)

This still begs the question: Why would we ever invest an additional 15 percent on code development by introducing pair programming? The higher quality that is obtained in initial code development reduces future test and field support resource requirements. Typically, in systems tests it takes between one-half (Humphrey 1995) and two (Humphrey 1997) workdays to correct each defect. Industry data reports that between 33 and 88 hours are spent on each defect found in the field (Humphrey 1995). When each defect saved during code development can save defect correction time of between .5 and 88 hours, pair programming quickly becomes the cost-saving and time-saving alternative (Williams 2000). Appendix B demonstrates the overall life-cycle affordability of pair programming based on an in-depth economic analysis.

It should also be noted that if time-to-market/cycle time is a prime motivator for you, pair programming can get the job done in about half the time. Could the same thing be done with two programmers working independently? Not likely. Increased communication and training time would increase their total time, as Brooks has told us for over a quarter of a century with his "Brooks's Law" (1975). Additionally, the quality would not be as high.

Data with students is a start in convincing you about improvements using pair programming. But what about industrial studies? It is difficult to obtain statistically significant results from industry. We do offer two case studies, both received via personal communication to Laurie.

- Bill Wood and Bil Kleb of NASA Langley report that in 2001 a pair of programmers re-implemented a numerical algorithm for wave propagation that was originally developed in 1997. The individual programmer worked for 6 weeks to produce 2144 lines of code. The pair worked a total of 3 programmer weeks, implementing the same functionality in only 866 lines of code, comprised of 403 lines of production code and 463 lines of testing code. It is important to note that the individual did not write any testing code and that while the individual was very experienced in the language, the pair was learning a new language. By combining pair programming with extensive

testing techniques, Wood and Kleb have much higher confidence in the new code.

- A technology company in India reports very impressive pair programming results. The prototype of a Voice-Over-IP project was done without pairing, although the actual project was done utilizing pair programming. The actual project was much more complex because it had to deal with meeting Quality of Service parameters, scalability, uptime, and so on. The paired project showed significant increases in productivity and quality. It is important to note that the manager for this project indicated that this project was a high-priority project. As a result, the teams worked nearly round the clock. The extremely high productivity numbers shown in Table 4-2 should not be considered as representative of pair programmers but rather should be considered in comparison to the solo programmers only. The data is summarized in Table 4-2.

Goal: I want to reduce my risk of losing a key person.

WIIFM: My stress level is reduced because I am not as concerned about the implications of losing a key person on the team and the resulting debilitation of my project. I feel more in control because key personnel cannot exercise their power by threatening to leave.

Table 4-2 India Technology Project Data

	Project One: Solo Programmers	**Project Two: Pair Programmers**
Project Size (KLOC)	20	520
Team Size	4	12
Effort (Person-Months)	4	72
Productivity (KLOC/Person-Month)	5	7.2
Productivity (KLOC/Pair-Month)	n/a	14.4
Unit Test Defects	107 (5.34 defects/KLOC)	183 (0.4 defects/KLOC)
System Integration Defects	46 (2.3 defects/KLOC)	82 (0.2 defects/KLOC)

Weinberg (1998) has a maxim, "If a programmer is indispensable, get rid of him as quickly as possible." His contention is that a project should not be a house of cards that collapses when a single "key" person is removed.

With pair programming, the project risk associated with losing this key programmer is reduced because there are multiple people familiar with each part of the system. If a pair works together consistently, then there are two people familiar with this particular area of the program. If the pairs rotate, many people can be familiar with each part. A common informal metric (invented by Jim Coplien) is referred to as the "truck number." "How many or few people would have to be hit by a truck (or quit) before the project is incapacitated?" The worst answer is "one." Having knowledge dispersed across the team increases the truck number and project safety.

We'll discuss the use of pair programming as a knowledge management technique more in Chapter 9.

Goal: I want my employees to be happy.

WIIFM: Undoubtedly, happier employees stay in their jobs longer. Turnover is very costly in both recruiting and training costs. Also, happier, less frustrated people are more positive about their jobs and are more eager to help the team meet the objectives.

The incorporation of pair programming has been shown to improve the engineers' job satisfaction and overall confidence while attaining the quality and cycle time results discussed earlier. Based on seven independent surveys of self-selected pair programmers, over 90 percent of pair programmers agreed that they enjoyed their jobs more when pair programming. The groups were also surveyed on whether working collaboratively made them feel more confident about their work. These results are even more positive—96 percent indicated that pair programming made them more confident.

Goal: I want to reduce the amount of time it takes to train a new person.

WIIFM: My training costs are reduced, which helps me manage my budget. New people can actually contribute to projects much earlier.

In *The Mythical Man Month* (1975), Brooks states his law: "Adding manpower to a late software project makes it later." He believes that communication costs are the major driver leading to this phenomenon. Brooks breaks these communication costs into training and intercommunication. Certainly, reducing training costs is a worthy objective.

Traditionally, people new to an organization are shown different parts of a system by senior staff personnel. This dedicated training time costs the senior personnel valuable hours. During these hours, neither the new person nor the trainer is making contributions toward the completion of the project. Through pair programming, the trainer teaches by *doing* (not showing), and direct contributions are actually made during the training time. Additionally, training seems to go much faster, and the new person learns the system more intimately.

We'll explore the beneftis of pair programming as a training technique further in Chapter 9.

Goal: I want my teams to work well together and to communicate more effectively and efficiently with each other.

WIIFM: My team works together better because they know each other and like each other. This makes them happier employees. It also greatly reduces information "islands" because people are more likely actually to talk to each other more often, sharing problems and solutions.

There are many stories of teams that started with pair programming. Before pair programming, people would walk into work in the morning at different times with a brown-bag lunch in their hands. They'd walk into their office or cubicle, put their lunch down and their headphones on. They'd tap, tap, tap on the keyboard all day. At some point, they'd take their lunch out of the brown bag and eat it. At the end of the day, they'd take off their headphones and head home. They would mainly communicate with other team members during meetings and via e-mail.

After pair programming, these teams were profoundly transformed. Pairing, the team members got to know each other better through the idle chitchat that goes on during pauses while pairing. A programmer might mention that he was going to a ball game or to her child's recital that night. The

next day, whether they were pairing together, one might ask how the recital went or comment on the outcome of the game when meeting at the vending machine. As the team gets to know each other better, they are far more likely to talk with each other about both personal and technical matters. The communication barriers between each other start to crumble. Team members find each other much more approachable. They will struggle with questions or lack of information for less time before getting themselves out of their chair and going to ask the right person a question—because now they know that person quite well. The rapport and trust built between team members gives them the courage to ask each other for advice and guidance without feeling vulnerable and insufficient. Additionally, they feel better about their jobs because they know their teammates on a personal level.

We contend that communication is made more efficient in another important way. As we said earlier, Brooks considers training and intercommunication costs to be major cost factors. Brooks (1975) asserts that "if each part of the task must be separately coordinated with each other part, the [communication] effort increases as $n(n–1)/2$." It's easy to think about the items that need to be done in order to coordinate two interdependent parts: Dependencies need to be identified and scheduled accordingly, interfaces need to be specified, technical decisions might need to be made jointly, change management activities need to be accomplished, and so on. Additionally, progress might be slowed or completely halted if some critical coordination needs to occur when a team member is missing.

Let's think about how pair programming can make this communication more efficient. Consider first if a team does not rotate pairs but assigns larger pieces of functionality to static pairs. Instead of breaking the project into n parts, the project is broken into $(n/2)$ parts, and the communication effort increase is reduced from $n(n–1)/2$ to $n(n–2)/8$. When pairs work together, they make decisions on dependencies, technical aspects, and interfaces as they go. No separate coordination activities need to take place; no dependencies and interfaces need special documentation, improving the efficiency of team communication. If pairs do rotate, and programmers partner with the programmer with whom their task is interdependent, we believe this intercommunication cost can be even further reduced because needed communication about interfaces and other issues will happen during the natural course of pairing.

References

Boehm, B. (1981). *Software Engineering Economics*, Prentice Hall.

Brooks, F. P. (1995). *The Mythical Man Month: Anniversary Edition*, Addison-Wesley.

Brown, W. J., Malveau, R. C., McCormick, H. W., and Mowgray, T. J. (1998). *AntiPatterns*, Wiley Computer Publishing.

Cockburn, A. and Williams, L. (2000). "The Costs and Benefits of Pair Programming," *eXtreme Programming and Flexible Processes in Software Engineering—XP2000*, Cagliari, Sardinia, Italy.

Humphrey, W. S. (1995). *A Discipline for Software Engineering*, Addison-Wesley.

Humphrey, W. S. (1997). *Introduction to the Personal Software Process*, Addison-Wesley.

Jensen, R. W. (1996). "Management Impact on Software Cost and Schedule." *Crosstalk*, July, http://stsc.hill.af.mil/crosstalk/1996/jul/manageme.asp.

Krantz, M., and Iwata, E. (June 11, 2001). "Companies Bleed Cash when Computers Quit," *USA Today,* B1.

Nosek, J. T. (1998). "The Case for Collaborative Programming." *Communications of the ACM,* 105–108.

Weinberg, G. M. (1998). *The Psychology of Computer Programming, Silver Anniversary Edition*, Dorset House Publishing.

Williams, L. A. (2000). "The Collaborative Software Process," Ph.D. dissertation, University of Utah.

Williams, L., Kessler, R., Cunningham, W., and Jeffries, R. (2000). "Strengthening the Case for Pair-Programming." *IEEE Software,* July/August 2000, 19–25.