Paired Programming in the Software Factory

Questions and Answers

**Thomas Meloche, James Goebel
and Richard Sheridan © 2003
The Menlo Institute**

**The Menlo Institute LLC
212 N. Fourth Ave
Ann Arbor, MI 48104
www.menloinstitute.com
(734) 665-1847**

TABLE OF CONTENTS

Paired Programming in the Software Factory

Upon entering the Software Factory it is immediately apparent to most visitors that things are very different.

The Software Factory

The space is large, two stories tall, open and accessible. There are no walls and there are no cubes. The floor is painted cement and not carpeted. The space is broken into work areas by long folding tables that are clustered in groups of three or four. On each table sits a single computer and in front of that station sits two developers. In the Software Factory there is only one computer for every two developers. All software development is done in pairs.

Today is the first day after a new iteration has begun. As you look around the room you notice a variety of different behaviors. At one station one person is typing and talking while another is reading the code being typed, occasionally making comments or suggestions. At a white board at one end of the room a group of four appears to be talking about software design, a class diagram appearing on the board. At a cork board at the other end of the room one person is describing screen shots and user behavior to two more people. At yet another workstation three people are seen sitting together, two looking on as one is typing at the keyboard. At yet another station one person is typing while the other person is controlling the mouse.

The factory is busy, alive and energetic. There is also a fair amount of noise. Yet work is clearly getting done. Extreme Programming as it is practiced in the Software Factory has a simple rule about paired programming – all production code must be typed into the system by a pair of developers.

Paired Programming in the Software Factory

Paired Programming

In the Software Factory the pairs are remixed by the manager at the beginning of each iteration. The pair is generally expected to work together for that iteration to accomplish the stories that they have been assigned. The manager has assigned pairs to insure that no one on the team becomes the only person who knows how to do something. The manager is also assigning pairs to combine talents, experiences and skills and facilitate communication across the team. The assigned pair is free to break up and assist other pairs as needed, but they still are expected to work to complete their own stories as well.

There are many benefits experienced by teams that do paired programming. And we are always quick to point them out to our visitors. These benefits include:

- All production code is always reviewed for bugs, clarity, simplicity and compliance with code programming conventions.
- All production code is designed and understood by at least two team members.
- All production code is made robust. At least two people have concentrated on trying to identify how the code may fail. They have designed, built and run an extensive set of unit tests for all production code.
- Team members automatically share skills, tips, patterns, idioms and experience.
- New team members can be easily integrated into the team.
- Fewer meetings are required.
- Fewer bugs are generated.
- The team is never held hostage to the ability or knowledge of any one individual.

If you think it is this easy to convince most people to try paired programming, think again. During our tours the practice of paired programming, more than any other, leads to skepticism and resistance from our visitors. Following are some of the most frequently asked questions and frequently raised objections.

Doesn't paired programming cut your productivity in half?

When describing the benefits of paired programming, managers often comment:

"Gee, that sounds great but I'm surprised you're willing to cut your productivity in half to achieve these results."

We used to be those managers. We used to believe that whenever we saw two programmers "collaborating" at a single machine that at least one them was "wasting" their time. We don't think that way anymore.

In order to consider the productivity question, one must first answer the question: "What is software productivity?" or better yet: "What do we want our programmers doing all day long?"

Many fall into the trap that "lines of code" is the answer. As if programmers have all the lines of code in their head and productivity is simply measured by the speed at which they can type them into the computer. Lines of Code, Function Points, Classes coded or even Use Cases completed is not a good measure of productivity. Unfortunately, as we have observed it, productivity for the development team as a whole is much more ethereal and is better described by the following question:

How much customer value can we create per dollar spent?

We are so single-mindedly focused on individual productivity that we tuck our developers away in cubes, create separate status reports and review processes so that we can be sure we are measuring a single individual's productivity, regardless of how the team is doing.

It took us a while to get to this point, but we don't really care about individual productivity anymore. It is the team's productivity that is most important. It doesn't matter if I have the world's most productive developer on a project if the project fails anyway. The same is true for any team sport, great players are great to have but the ***team*** still has to win.

Our experience and observation is that most teams focus on individual productivity and miss the real point "Is the team shipping software that adds customer value?" The end result of this misplaced focus is that the team, in fact, is not very productive at all. At least not in the way we measure productivity:

How much customer value can we create per dollar spent?

Paired Programming in the Software Factory

Try answering the following questions:

1. Do you have a very senior team with little fresh blood? Is your idea of a *dream team* one that consists of all senior developers?
2. Do your programmers have a difficult time moving to new projects or to new areas within a project?
3. Is it difficult for your programmers to grow their skills in new areas and do new things because their knowledge in a particular area is too valuable to allow them to do anything else?
4. Do your programmers have uneven work loads? Are some programmers continually overworked while others have little to do?
5. Do your programmers look busy, but you actually have no idea what some of them are doing?
6. Do you suspect some of your team members are building and inventing things that nobody has asked for?
7. Is it difficult to figure out what new programmers should do even though it is quite clear that you need to grow your team?
8. Is it difficult to make a new programmer productive?
9. Is it difficult to find interesting work for a new programmer?
10. Are you in a race to make new programmers productive before you demoralize them? Is the happiest day of a new programmer's tenure the day of the interview?
11. Do your programmers write detailed weekly status reports?
12. Do your programmers write analysis, design, review or other documents that no one seems to read or that are only read once?
13. Do your programmers write documents that are quickly out of date and hence useless?
14. Do your programmers spend eight or more hours a week in conference rooms or other meetings?
15. Is your software buggy?
16. Does your software have bugs reappear after they have been fixed?
17. Do you enter a long QA cycle of release, then fix bugs, then release again, then fix more bugs; where you struggle to get the quality product out the door?
18. Is your software installer difficult to build?
19. Is your software difficult to install?

How many of the above questions did you answer in the affirmative? Every item above answered in the affirmative is costing you productivity. Your critical path is not the ability of your individual programmers to write code! It is the ability of your team to release working software. Paired programming will provide a significant increase in productivity for you and your team.

O.K., say you believe us. It still leaves these interesting questions.

Why is paired programming more productive?

Where is the "compensation" that results in greater productivity for a paired team over a traditional team where people work alone?

Improved Quality of Result: If the programmers' productivity is a function of their typing speed then give them typing classes. However, if the programmers' productivity is a function of the quality of their thinking and their ability to communicate it to their co-workers then pair the programmers. In paired programming you have two people actively engaged and focused on the result.

How is productivity positively impacted if the second programmer simply finds one extra bug per week? Per day? Per hour? How difficult/costly is it to find those bugs later?

We've never seen a traditional code review meeting work, for a variety of reasons. As a result most teams don't have them, or if they do have them they are ineffective. Paired programming has two people committed to the quality of the resulting code – always.

Less "Stuck" Time: If an individual developer gets stuck, they usually stay stuck for too long before seeking help. Often seeking help is seen as a sign of weakness, perhaps enough of a sign to show up on the next performance review. A partner will actively work to get their partner unstuck and accept help when needed. On many occasions in our programming past we have experienced a partner finding an answer in a few minutes to a problem a person working alone had spent hours trying to resolve.

Less "Wander" Time: A partner will be less likely to wander off to the net to check today's stock process, etc. if she knows she is dragging her partner along with her.

Always Two Perspectives on Any Solution: Even if one of the partners is very senior and the more junior person is spending a lot of time asking questions, both partners benefit. Have you ever noticed how much better you understand something whenever you have to verbalize the answers? We also challenge the idea that anyone in our industry is truly more senior than anyone else. The fact of the matter is that our team members who have many years of experience have a lot to offer, but our newest and sometimes youngest team members have the benefit of having learned in a different environment and therefore have a different perspective when it comes to problem-solving.

An Atmosphere of Assistance. Now that we've broken down the barriers related to asking for help, an almost magical quality invades the entire team. Now everyone's willing to help everyone else and everyone is willing to ask for help when needed.

Far Less "Fear of the Unknown". Our experience with paired teams is that two people are far more willing and excited than a single developer to dive into a new unknown technology. Often individual developers when presented with a new technology want to attend a class before really diving in. My experience with pairs is that a book from Border's or a helpful web site is plenty to get them started on a new topic.

Far Less Apt to Drift off Task. Different than wandering into obviously unproductive activities such as surfing the net for best air fares or stock prices, wandering off task is a result of a developer "missing their exit on the highway" or not stopping when the task is completed. This results in increasing the scope of the task, many times without realizing it. We have found that this will be less likely to happen with a pair, since the partner will usually challenge the inclination to increase the scope.

So do we see our productivity cut in half? No. We see pair programming significantly enhancing our productivity. We believe most managers are fooling themselves into thinking that they currently have a very productive team. Industry performance statistics tend to support our point of view.

We encourage managers to carefully observe their current team. See how many times a status report describes how long a developer has been stuck on a problem instead of describing a long list of results completed and incorporated into the build. How often are developers wandering off task?

I once had a developer challenge all of this saying what they really enjoyed doing was working on "hard" problems. They said they'd rather work on a problem for four hours than have someone else help them solve it in five minutes. Their manager was in the room when they said it. Their manager thought this was one of the most productive members of his team. Consider this story the next time you think about your team's productivity.

Why do you assign pairs?

Kent Beck describes pairs naturally forming and breaking so we are often asked why we assign pairs. This is especially interesting to us because the practice seems to really rub some XP enthusiasts the wrong way. In an environment that I would describe as almost fully compliant XP otherwise, every programmer was assigned a default pair for each iteration. The individuals were never prohibited from changing partners if they felt it would help them to meet the iteration goals, but it could be very awkward if someone decided to simply “abandon” their partner in mid-iteration.

We originally assigned pairs, as we were adopting XP, in order to better control the experience for the team. This seemed to relieve some amount of social tension.

The pairs were assigned in order to maximize the social connections within the team. The team started with about ten developers and grew to over thirty. Early in the process a matrix was prepared with each individual's name across the top and down the side. Each time a pair was assigned, a notation was made in the appropriate rows and columns. In some ways the pairings could have been random as long as they maximized the number of new connections within the team. This created many pairs of developers that might not otherwise have self selected to work as a pair. After working with each other for two weeks (one iteration) they had a better appreciation for each other's strengths and weaknesses. And thereby we found ourselves with a more self-aware team.

The manager did not actually use a dartboard to make random assignments. Instead he would write team member names on the back of his business cards and then mix and match the names into pairs. He would then check to see if the "line-up" (coaching analogy) met the criteria of maximizing new pairs (or at least pairs that had not worked together for awhile). This also allowed him to select specific matches where an individual might need mentoring on a particular skill. Clearly this is a form of social engineering. But my understanding of a good manager is that they spend their time working on building the social dynamics of a good team, while letting the team get their work done.

Assigning pairs facilitates our ability to pick-up temporary resources for an iteration or two. Developers were very practiced at working closely with new people and the new resources were assigned to a more seasoned and supportive partner, thus enabling us to effectively scale the size of the team and the amount of output for short periods of time as needed.

I would be very interested in hearing feedback from any XP team that decides to run an assigned (or random assignment) experiment for some number of iterations.

1. Did new pairs occur that had never occurred before?
2. Did you find that working with someone different made bad habits less comfortable?
3. After working with a partner, are you familiar with a broader set of code than you were before?
4. Is there another project with resource slack?
5. Did you try adding someone new for an iteration or two?
6. Was this someone who might not have been assertive about finding a partner otherwise?
7. Did the longer stretch of pairing help you, or would you have been better off changing partners more frequently (as yet another experiment)?

“Test it - then measure it.”
- James Goebel

What do you do when two people don’t get along?

Well, we’ll start by answering your question with a few questions of our own:

1. What do *you* do when two people don’t get along?
2. Do you know when two team members disagree?
3. How does it impact your development?

4. Do they interact and cooperate? Or are they alone in their offices month after month not sharing their conflicting views of the system they are suppose to be building together?
5. How well do you think their code will integrate?

To build software effectively, the team members must be able to get along – at least professionally. The difference between our teams and most teams is that if we have two developers who don't get along, we will know it clearly within a few days. We act as coaches and work to solve the problem. Our Java Factory team grew to over 34 developers, some of whom had worked together for over fifteen years; there was plenty of opportunity for personality conflict and tension. However, in practice there were two main causes of conflict:

1. A person who had a hard time taking advice.
2. A person who wanted to over invent the solution and not do the simplest thing that achieved the business goal, thus risking the delivery date and the team goals.

In our environment it quickly becomes clear to the coaches who have these problems. These are the developers who most likely will sink your projects, not save them.

In practice, we never had to dissolve a pair during an iteration. Below are a few reasons we could identify why this was the case for our team:

1. Lots of Partners: Our team members knew they would likely only be assigned the same pairing partner only once or twice a quarter.
2. Public Work Area: Any arguing or other pettiness would have to be performed in front of the whole team.
3. Professionals: We hire professional software developers, not children. Although a key ingredient is the ability to play well with others.

When the original team members were first introduced to the idea of paired programming we asked them what they thought it would be like. The team was remarkably silent for a few moments until one programmer finally said "Blood, Murder and Mayhem." This is a fairly typical reaction to paired programming. Our results were quite the opposite, "Milk and Cookies, Cooperation, and Order."

How do you measure and reward individual performance?

Do you do performance reviews?

Isn't it difficult to tell how someone is doing?

As stated previously, more and more we find the question of individual performance uninteresting. What really matters is the performance of the team as a whole. That being said, your company may still require individual performance reviews on a yearly schedule. If so, don't worry. It is remarkably easy to do individual performance assessments in an Extreme Programming environment. It is remarkably easy to tell how someone is actually doing:

- Ask their partners
- Observe how often they are in the factory working with a partner
- Observe how often they are alone
- Observe how often they complete their stories in an iteration
- Observe how well they follow the unit testing practices
- Observe how others treat the code they have written
- Observe how willing others are to pair with them
- Observe how willing they are to help others who are stuck

If you are doing Extreme Programming you will have more real and relevant data about individual performance than you ever had before. It is not difficult at all.

What do you do if someone is sick or on vacation?

What do you do if you have an odd number of developers on the team?

Be creative. Team members are encouraged, if they find themselves alone, to look for another person without a partner and pair with them. The odd person is the designated hitter, filling in for someone who has had to leave. The goal is for the team to be successful; the team members look to see if there is someone else alone who they can help or who can help them.

If you can't find someone to partner with then:

- Write unit tests for the code you anticipate writing
- Write unit tests for existing code and try to break it
- Write functional tests for the functional test team
- Or, worst case, write non-production code. Do the quickest, dirtiest proof of functionality. Then work with a partner later to write it as production code.

What do you do if team members assigned to a pair have different schedules?

We let the team members solve this problem themselves. Since an iteration is only two weeks long the solution generally does not have to dramatically change someone's lifestyle. Of course, the goal is to have as many developers in the collaborative space at the same time as possible. So the general rule is to be in the Software Factory at least between the hours of 9am and 4pm. The pair is free to set their schedule to be 9am to 6pm one day and 7am to 4pm the next.

Isn't pairing really about senior developers mentoring junior developers?

No. Pairing is not simply about mentoring. Mentoring is a natural result of doing paired programming but it is just one of many benefits. We would still pair our developers even if the entire team consisted of senior developers for the code review, communication and flexibility benefits.

It is interesting to point out that learning is almost always a two way street. We often see junior programmers teaching senior programmers. In fact, a team of all senior developers is no longer our "dream team." The reason is that many *senior* developers tend:

- To write code that is too complex for others to read or maintain
- To over invent and produce more than is required by business sponsors causing late delivery
- To use old approaches that have worked in the past but are no longer as effective

Seek to have a least half the team consist of junior and middle level talent. It is the best way to stimulate fresh ideas within the team and ensure that the code will, in fact, be maintainable by the junior talent that is likely to inherit it in the future anyway.

Do you always pair junior and senior developers together?

No. We heard advice from an early XP Guru saying that we shouldn't pair two junior programmers together. Of course, being researchers and contrarians, we decided to run the experiment. We paired two junior programmers together who had both been on the team less than two months. It worked fine. So much for the *experts*. Of course there are a few reasons why it likely worked. Both junior people had already been paired at least twice with other people on the team. They now knew well at least 4 other team members who were all within ear shot. It was a simple matter for them when they got stuck to ask for help from others on the team. Were the two junior programmers as productive as two senior programmers or as a senior and a junior? No, be serious, but the interesting lesson was that it was possible. So, pair two junior programmers if you need to.

When we add someone new to the team we always pair them with someone experienced with the process and tools. But they aren't necessarily senior developer with years of experience.

Do you keep track of who has paired with whom?

Yes we do. One of our goals is social engineering. As managers trying to protect business interests we believe it is our job to ensure that the knowledge is spread throughout the team. In addition, we must ensure that no one on the project becomes too specialized and that no one on the project is indispensable. All of these reasons leads us to control and track the assignments people work on and the partners with which they program.

To track pairs we keep a simple spreadsheet with the team members listed along the top and the side. When members are paired at a given iteration we simply record on the sheet the iteration they were paired. This gives us a good history of the pairings and allows the coaches to insure that the knowledge of the system is spread across as many people as possible.

Conclusion

As you leave the software factory, consider what you have seen. We have experienced that the concept of "paired programming" has worked well for us. The benefits have been significant; far outweighing the "cost" of having two programmers work on the same code. We encourage you to try it in your organization and share your experiences with us.