A Field Study of Developer Pairs: Productivity Impacts and Implications

Allen Parrish, Randy Smith, David Hale, and Joanne Hale, *University of Alabama*

In a previous industrial field study, we examined programmer productivity as team size varies. We showed that regardless of size or experience, the more time team members spend concurrently working on the same code modules, the less productive they are.[1,2] In other words, teams are more productive when their members work independently.

As our research was underway, the phenomenon of agile software methods exploded, promising increased quality, shorter time to market, better efficiency, and greater customer satisfaction. In *pair programming*, a key ingredient in agile projects' success, pairs of programmers working together handle all development tasks—each pair sharing one terminal, keyboard, and mouse.[3–5] Researchers have since reported that pair programmers produce higher-quality code than do developers working alone.[6–8] This finding is intuitively appealing; you'd expect collaborators who continuously review each other's work to find better design and construction solutions than one programmer working alone. More surprising is their finding that pair programmers produce code with little or no significant productivity loss.[6–8]

This research contrasts starkly with our field study findings. We decided to reexamine our earlier productivity data from teams of two and ask: Would our previous findings of concurrent-work productivity loss be reversed if we look only at programming pairs rather than teams of all sizes? If so, we could conclude that pairs are naturally more productive than larger teams, regardless of the collaborative process. If not, we could conclude that the collaboration mechanisms prescribed in pair programming might overcome a natural loss of productivity from concurrent work. We offer these findings and their implications as a benchmark against which we might measure the potential of pair programming practices.

Pair programming purportedly delivers quality code with little productivity loss. The authors' field study, outside the pair programming environment, shows that two-person teams working independently are more productive than those working concurrently. Agile methods may overcome inherent productivity losses of concurrent development.

0740-7459/04/$20.00 © 2004 IEEE

The conundrum of pair programming and productivity

Pair programming exemplifies the highest level of collaborative effort, prescribing two programmers to work side by side at one computer, jointly responsible for producing one artifact. One person is the driver, controlling the equipment and physically writing the code; the other is the navigator, performing ongoing peer reviews and acting as consultant and advisor. Team members regularly swap driver and navigator roles.[3–5]

In our earlier work, we examined how assigning development tasks to programmers affects development effort.[1,2] One result from that work is a *concurrency* metric—the degree to which different programmers report working on the same module during the same day. Concurrency measures collaboration *potential*: in a high-concurrency situation, there is the potential (and some likelihood) that the team members are working together. On the other hand, in a low-concurrency situation (team members working on the same module but not on the same day), this probability obviously diminishes. Although concurrency isn't a perfect measure of collaboration, it's a necessary precursor, positively correlated with collaboration, and relatively easy to noninvasively collect in many industrial situations.

Pair programmers exhibit very high concurrency, working collaboratively at the same time on the same task. But they also follow the prescribed role-based protocol described earlier. To understand pair programming's impact, one strategy is to establish certain baselines in the absence of those practices. Specifically, what's the impact of concurrency in a situation where pair programming isn't practiced? If concurrency has a positive effect on productivity in this context, we can conclude that working concurrently in teams of two offers a natural productivity benefit. On the other hand, if concurrency has a negative effect in this context, the pair programming protocol must counteract this loss. So, how important is role-based collaboration in maximizing pair programming productivity?

To establish a baseline in this study, we separated the concurrency issue from the role-based protocol. In particular, we examined high-concurrency and low-concurrency programming pairs in a situation where no role-based protocol was prescribed. Our findings show that the former (that is, potentially highly collaborative pairs) are dramatically less productive than the latter (pairs working on the same task but not at the same time). Although this certainly doesn't imply a productivity loss with pair programming, it does mean that any productivity gains reported with pair programming are likely due entirely to the role-based protocol rather than to any inherent consequences of working closely in pairs.

Reexamining our research results

We reexamined our previous data isolating and analyzing the productivity of two-person teams who were developing a statewide time-accounting system. The analysis covered 48 modules developed by professional programmer pairs. For each such module, we examined the effect of concurrency on pair productivity.

The contractor was rehosting a legacy system to a distributed environment. The legacy system had over 3,000 screens and approximately a million lines of code.[1,2] The new system supports over 400 distributed users and accommodates departmental accounting and personnel systems, inventory control, and a large-scale bidding system for state infrastructure. The developers used a fourth-generation tool, a modern relational database, and a Microsoft Windows-based development environment that included report generators, COTS libraries, database systems, and other new components.

From preliminary design information, we measured unadjusted function points (UFPs) for each module, then recorded actual development effort using the development team's time-accounting system. The system records each developer using a unique identifier, a module identifier, time spent (in 15-minute increments) on the module, and the date the work was done.[1,2] Although the recording process might be inaccurate, using a noninvasive contractor accounting system let us collect a much more extensive set of productivity data than otherwise would have been possible.

We isolated three factors for this study:

- *Team size*: modules with development teams of size two (n = 48)
- *Concurrency*: the degree to which programmers reported working on the same module during the same day[1,2]
- *Productivity*: the average number of UFPs completed per unit of time

Although concurrency isn't a perfect measure of collaboration, it's a necessary precursor, positively correlated, and relatively easy to collect.

Table 1

T-test* results: Productivity versus concurrency level

Concurrency level	No. of developed modules	Mean productivity	Standard deviation
Low	23	4.709	3.973
High	25	1.125	0.726

* μ: population mean; 95% confidence interval for $\mu_{low} - \mu_{high} = (1.844, 5.323)$
T-test of hypothesis, $\mu_{low} = \mu_{high}$: t-value = –4.26; p-value = 0.000

We measured concurrency by the average number of programmers reporting work on a module over all days that the module was under development (see elsewhere for additional details[1]). Although the data analyzed doesn't provide posterior measures of quality at the module level, we tested all the modules using standard criteria prior to completion. So, from a functional perspective, all modules met a minimum quality standard.[1,2]

Concurrency and productivity

We measured productivity by the number of UFPs developed per hour of programmer effort. We classified teams reporting concurrency levels below the median as low-concurrency and those reporting levels above the median as high-concurrency.

We ran a simple t-test to determine if a significant difference exists in these two groups' average productivity. This statistical procedure tests the validity of the hypothesis that two samples (in this case, a low-concurrency sample and a high-concurrency sample) are drawn from populations with the same mean (in this case, with the same average productivity). Rejecting this hypothesis is equivalent to concluding that the probability (denoted by the p-value) is very low that any observed differences in average productivity between low- and high-concurrency groups is due to chance.

As Table 1 shows, the t-test results are compelling. Low-concurrency pairs (those working most independently) developed an average of 4.7 UFPs per hour, while high-concurrency pairs (those potentially working most closely together) developed an average of 1.1 UFPs per hour. So, the more independent pairs were over four times more productive than the more concurrent ones, with a near-zero probability that such a wide difference could have occurred by chance alone. This is in stark contrast with the reported productivity effects of the role-based pair programming protocol. In these studies, the productivity loss was either moderate or statistically insignificant.[6–8]

Implications for programming practice

Outside of a prescribed pair programming protocol environment, we could make intuitively appealing arguments for the productivity of highly collaborative programming pairs on the basis of two claims:

- A pair of experienced, methodology-trained professional developers shouldn't need a prescribed collaboration protocol to effectively and productively plan, communicate, execute, and review tasks.
- Programming pairs can learn over time to work more productively together by devising their own productive collaboration process.

The data from our field study indicates that neither argument holds true. Refuting the first argument is our finding that the high-concurrency pairs were significantly less productive despite the fact that they were experienced, methodology-trained programmers. On average, these professional programmers had 10 years of work experience. Our assessment of pair productivity over time refutes the second argument. We found that pairs were no more productive on later modules than on earlier modules.

There are at least two intuitively appealing explanations:

- Pairs working together aren't naturally productive—they need the collaborative role-based protocol that pair programming provides to combat this productivity loss.
- Pairs working concurrently on the same task might be unproductive because of duplicate work—simultaneously working on the same problem or making conflicting changes that must be reconciled later.

Our field study supports the first rather than second explanation. Although we recognize the potential for the second cause of lost productivity, it doesn't appear to be a strong factor in this case. The team of 16 developers worked at the same location and used a version control system with source code tracking and check-in and check-out file locking. Although this doesn't preclude the possibility of duplicate effort, it

significantly reduces its potential. So, we conclude that the role-based protocol prescribed in pair programming overcomes a natural productivity loss from working in pairs.

Some prior empirical evidence is available regarding the productivity effects of pair programming. However, these studies involved student programmer pairs[6,8] or professional programmers working on short, 45-minute tasks.[7] Particularly given the increased use of pair programming—in a recent worldwide survey,[9] 35 percent of 104 development projects said they incorporate pair programming—more empirical evidence from real industry projects is needed.

A key to establishing realistic expectations will be industry participation in data collection and experimentation. This requires significant effort and might involve releasing valuable proprietary data. In the interim, our study provides an industry baseline: Pair collaboration requires significant resources, and a prescribed pair programming protocol shows promise for reducing this cost and, better still, improving productivity.

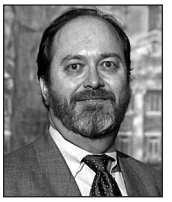
The concurrency metric provides a valuable framework for future work in this area. In particular, we can control for it. Thus, among high-concurrency pairs, we could compare pairs employing the role-based pair programming protocol (side by side on a single machine) with other interaction models and thus evaluate the typical process model that agile methodologies employ. Future work must also explore other factors that might alter the effectiveness of pair programming practices, including programmer expertise, problem complexity, problem domain, and development environment. This work is an important next step in the quest to quantify the overall utility of agile methods.

Setting aside any productivity impacts in an industrial setting, pair programming's other valuable benefits are already well established. Pair programming teams not only create higher-quality code but also find greater enjoyment in their jobs and more confidence in their work.[7,10] Robert Glass lauds agile methodologies for their emphasis on constant unit testing, continuous integration, customer participation, and programmer-friendly workload expectations.[11] This study certainly doesn't try to refute these claims. We expect to see mounting evidence that these benefits are real and significant.

About the Authors

Allen Parrish is an associate professor in the Department of Computer Science and director of the CARE Research and Development Laboratory at the University of Alabama. His research interests are in software testing, software deployment, data analysis and visualization, and highway safety information systems. He received his PhD in computer and information science from Ohio State University. Contact him at the Dept. of Computer Science, Box 870290, Univ. of Alabama, Tuscaloosa, AL 35487-0290; parrish@cs.ua.edu.

Randy Smith is an assistant professor in the Department of Computer Science at the University of Alabama. His research interests are in software effort estimation, software process, and data analysis. He received his PhD in computer science from the University of Alabama. Contact him at the Dept. of Computer Science, Box 870290, Univ. of Alabama, Tuscaloosa, AL 35487-0290; rsmith@cs.ua.edu.



David Hale is the director of Management Information Systems programs in the Manufacturing Information Technology Center at the University of Alabama, and in the Information Technology-Workforce Resource Center of Alabama. His research interests include aging infrastructure, enterprise integration and modeling, component-based software development, and software maintenance. He received his PhD in MIS from the University of Wisconsin, Milwaukee. Contact him at the Dept. of Information Systems, Statistics, and Management Science, Box 870226, Univ. of Alabama, Tuscaloosa, AL 35487-0226; dhale@cba.ua.edu.

Joanne Hale is an associate professor of management information systems at the University of Alabama. Her research interests include software development metrics, software maintenance, software cost estimation, and component based development and reuse. She received her PhD in MIS from Texas Tech University. She is a member of the IEEE and ACM. Contact her at the Dept. of Information Systems, Statistics, and Management Science, Box 870226, Univ. of Alabama, Tuscaloosa, AL 35487-0226; jhale@cba.ua.edu.

References

1. R. Smith, J. Hale, and A. Parrish, "An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation," *IEEE Trans. Software Eng.*, vol. 27, no. 3, 2001, pp. 264–271.
2. J. Hale et al., "Enhancing the COCOMO Estimation Models," *IEEE Software*, vol. 17, no. 6, 2000, pp. 45–49.
3. K. Beck, "Embracing Change with Extreme Programming," *Computer*, vol. 32, no. 10, 1999, pp. 70–77.
4. K. Beck, *Extreme Programming Explained: Embracing Change*, Addison-Wesley, 2000.
5. L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison-Wesley, 2003.
6. L. Constantine, *Constantine on Peopleware*, Yourdon Press, 1995.
7. J. Nosek, "The Case for Collaborative Programming," *Comm. ACM*, vol. 41, no. 3, 1998, pp. 105–108.
8. L. Williams et al., "Building Pair Programming Knowledge through a Family of Experiments," *Proc. 2003 Int'l Symp. Empirical Software Eng.*, IEEE CS Press, 2003, pp. 143–152.
9. M. Cusumano et al., "Software Development Worldwide: The State of the Practice," *IEEE Software*, vol. 20, no. 6, 2003, pp. 28–34.
10. L. Williams and R. Kessler, "All I Really Need To Know about Pair Programming I Learned in Kindergarten," *Comm. ACM*, vol. 43, no. 5, 2000, pp. 108–114.
11. R. Glass, "Extreme Programming: The Good, the Bad, and the Bottom Line," *IEEE Software*, vol. 18, no. 6, 2001, pp. 112–111.