

Chapter 28

Mob Programming and the Transition to XP

—Moses M. Hohman and Andrew C. Slocum

Extreme Programming (XP) development practices emphasize the importance of developer communication. Our team is currently in a transition toward using many of XP's suggested lightweight practices. As part of and to facilitate this transition, we have been developing a collaborative method we call mob programming. The term "mob programming" is whimsically derived from the term "pair programming" and indicates the practice of refactoring code in groups larger than two developers. The purpose we identify for this refactoring focuses less on writing code that we will use later and more on encouraging healthy discussion. In this chapter, we describe our process and its precursors as they have evolved over time. We report the successes realized and the failures suffered in the context of the values of XP.

Introduction

The benefits of pair programming are documented: better, simpler code written in less time; improved interdeveloper communication and feedback; and shared code ownership, to cite a few [Williams+2000; Cockburn+2001]. We have undertaken an experiment on our development

Moses M. Hohman, Andrew C. Slocum. All rights reserved.

team to see if some of these benefits remain when the programming group grows beyond two people. This experiment evolved from regular developer lunch meetings originally run in a presentation format, with one member of the team presenting code familiar (usually only) to him or her. At the time, we were not employing XP practices but had vague plans to do so at some point in the future.

Since then, the team as a whole—not just the developers—has begun to take the first steps toward XP. We now have iterations and some form of continuous integration, we are writing unit and acceptance tests, and we have dabbled in pair programming. Because our team is unfamiliar with and therefore unsure of XP's benefits, it is useful for us to find ways to experiment with and eventually to reinforce beneficial XP development practices.

To this end, we have identified long-term and short-term goals. Regular developer meetings cannot achieve all these goals on their own. However, we intend for the meetings to facilitate this achievement in the context of a broader set of XP practices. In the long term, we want to decrease individual code ownership, to encourage pair programming, and to improve unit test coverage. We want to improve the overall quality of the code we write. Formal mentoring is not an option, so instead we want to provide greater opportunity for developers to work together, enabling them to exchange information about coding standards, helpful programming patterns, and design decisions. Finally, we hope that working together regularly will build our sense of ourselves as a team.

In the short term we want these meetings to be fun and interactive. We want the format to respond to our needs and interests as they evolve over time, so that the meetings do not feel forced. We also typically set a small practical goal before ourselves to provide something as a focus for our attention. In the beginning this practical goal was the short presentation but has recently evolved into what we have capriciously termed “mob programming,” the refactoring of a small piece of code in groups larger than two people.

The formation of this weekly ritual has been an iterative and ongoing process as we continue to strive for a format that feels “natural.” We have by no means arrived at this desirable goal, but we hope that an account of our experiences is nonetheless interesting to the reader. Therefore, we begin with a historical description of our efforts. With this picture, we move on to compare an idealized view of mob pro-

programming with the values of XP and to discuss the lessons we have learned that help mob programming be more productive. Before concluding, we discuss some of the problems we have encountered during our experiments.

Description of Methods

Early Days

In early January 2000, we began a series of weekly developer lunches, whose purpose is best described by quoting the e-mail one of authors sent to the developer team.

Here is the basic idea I propose (subject to any modifications you guys feel are necessary of course): each week . . . someone will give a short (30 min), informal presentation of some piece of development work. . . . The presentation will be followed by group discussion of the material for another half an hour or so. . . . Loosely, these talks will hope to serve to: familiarize us all (esp. new people!) with the . . . architecture, help those giving the talks to come up with new ideas about how to solve architectural problems, and give us a chance to all talk shop together as a group. I imagine that more familiarity with what everyone does might also help when we try . . . [pair] programming . . . in the coming months. . . . You should try to show some real code . . . I emphasize that the point of these talks is to provide an opportunity for informal group discussion. Don't worry about making a powerpoint [sic] presentation . . . [Hohman2000]

When these meetings began, individual code ownership was the rule. Developers who had been around for a while were pushing for the adoption of XP techniques (such as unit testing and pair programming), and new developers had a hard time getting a handle on the already present code base. Frequent and regular code discussion was unanimously received as a good idea.

Our goals were similar to the broader goals of XP. We wanted to increase developer communication, both to increase morale and to aid in the dissemination of crucial knowledge about our software. We wanted to decrease individual code ownership. With a broader knowledge of the overall architecture, each of us would be less afraid to change

“someone else’s” code. We wanted to encourage more feedback within the group on design and coding decisions. And we wanted a lightweight solution. Formal presentations, arduous preparation, and prescribed meeting content were not going to be accepted by the group. We wanted to provide a forum for people to talk about issues they already found interesting.

First Light

The presentations were boring. They resulted in little actual feedback or informal discussion. The first half hour mercilessly took over the second half hour, because the second half hour was often awkwardly silent. After a couple of months, we had already discussed every part of the application, at least at a general level. We needed to make a change. The presentation format did not serve our goals. We needed to increase the emphasis on group discussion.

For a couple of meetings, we completely dropped any official agenda whatsoever and just had lunch together. Discussions arose naturally, often with different parts of the group discussing different topics. However, as often as not, these topics were not related to our work. There was little focus, and as a result, people began to lose interest and leave after they had eaten. A total lack of structure was not good either.

To re-create focus, we wrote a simple utility to pick one Java class at random from our source tree. The day before each meeting, we would pick a couple of classes and announce them to the group. At the meeting, we would project the code on the wall and discuss it. The idea was that, over time, we would have the opportunity to discuss code from all parts of the application while avoiding the arduous presentations.

By looking at the code one class at a time instead of one module at a time, we did not have to worry about running out of material after two months. Familiarity with a particular module could grow over time. In this format the person who had written that particular class still did a lot of the talking; however, we did notice that these meetings elicited more discussion than our original format.

Looking at code at the class level was more instructive than at the module level. At module granularity, a half-hour presentation can cover only the barest summary of the screens, the business logic, and the high-level architecture. Also, module presentations become less helpful and less interesting over the life of the project. As the code complexity of a module increases, the level of detail of a presentation must decrease.

At the class level, developers can see the similarities between modules. Many of our design patterns apply across the application, and a discussion of one module's use of a pattern can explain an entire layer of the application. Additionally, common utility classes can be hard to find or are implemented several times. Class-level discussions with all the developers present can reveal or prevent this kind of redundancy.

Which Brings Us to the Present

One day, a few months ago at the time of this writing, we were particularly distressed by the quality of code under discussion. So we decided to go ahead and make a few changes right then and there. We were not particularly careful, and there were no unit tests written for the class, so we ended up breaking the build. To restore the build to health, someone had to spend time throwing away the changes we made. However, we had learned something. Changing the code during the meeting made the discussion more interesting, perhaps because it forced people to think, and when people think, they tend to have more to say. We decided we were on to something, so we subsequently added structure to make the process safer and to improve on the idea. This process is still a work in progress, so here we present only our current view of its ideal execution.

Some period of time before the meeting—the most effective period is a few days—one or two classes are picked from the source tree for discussion. The person who runs the selection utility vetoes generated code and code destined to be removed soon from the source tree. The remaining results are then filtered based on level of interest, until there are one or two worthwhile candidates. Two people (“pre-viewers”) spend time together looking over these classes to make sure that there is enough material for an hour to an hour and a half's discussion. If there is not, they replace one of the selections. These two people also examine the unit test coverage and augment it if necessary. Finally, these two developers begin to brainstorm issues for discussion and possible refactorings. When the previewing phase is over, the two e-mail the code in the body of an e-mail message to all the developers on the team so they have a chance to glance over it themselves before the meeting.

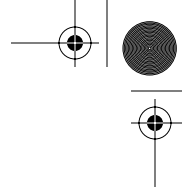
The day of the meeting, the group meets in a closed room to avoid outside distractions. For the first 20 minutes or so, we eat and engage in social discussion. Then we begin.

Developers play different roles in the process. One role is the “driver,” a person who types, compiles, and tests the code. Because we split into two groups, there are two drivers, one at each of a pair of laptops. We connect these laptops to projectors that display the code on opposite walls, giving each group some focus. We intend that all developers share the driver role (though this does not often occur; see the section Shortcomings). Second, the person or persons who originally wrote the code (or are familiar with it) take on a special role as “narrators,” providing a description of the code, its purpose, and how it fits within the context of the rest of the application. Finally, the rest of the group participates by offering suggestions and feedback. These people form the mob, probably the most difficult role to play and the least understood of the three.

At the beginning, the narrator tells the group about the code and its history. Then we split into two groups, one at each laptop/projector, and discuss ideas for refactoring. We discuss with our hands as well, typing in our suggestions to communicate them more clearly. If the refactorings we agree on are simple enough, we carry them out. After about half an hour, or when we feel it is time, the two groups rejoin and discuss the changes each group has made. We then decide which to keep (if any) and which to throw away.

The Intended Benefits of Mob Programming in the Context of XP

The four stated “values of XP” in Kent Beck’s *Extreme Programming Explained* are communication, courage, feedback, and simplicity [Beck2000]. Mob programming provides an opportunity for maximal feedback and communication. During meetings, the entire team shares ideas about all aspects of code design. For part of the meeting, we split into two groups small enough that everyone can be involved in the discussion at some level. Then when we regroup, each developer has an opportunity to share those discussions with everyone else on the team. By looking at a different piece of randomly chosen code each week, all developers get feedback on coding and design decisions they have made in the past, without being singled out. Also, during refactoring and wrap-up, our peers review the programming decisions we make during the meeting. Naturally, discussions that begin during mob programming can continue outside the meetings, fostering an atmosphere of greater



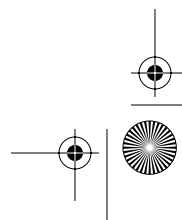
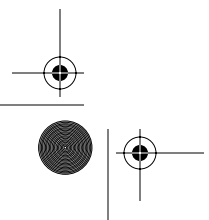
communication. We also hope that working together in this way may make us more open to pair programming outside the meetings.

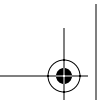
Mob programming also fosters the value of courage. At one time, we merely looked at code during meetings. Now we change it on the spot, as soon as we see something that can be improved. It is important that we actually change code during the meeting, because this active engagement of the material promotes thought and discussion instead of passive spectatorship. We also encourage throwing away the code at the end of the meeting if the group feels that the changes made are not an improvement. We try to rotate the roles of driver and narrator throughout the group so that each developer has an opportunity to be courageous. Honest, open discussion is promoted during meetings, so opinions that may not be expressed outside of meetings (such as a deeply divided opinion within the group about a design decision) are confronted.

To be able to change code courageously, good test coverage is necessary. The test coverage of our code increases as a function of the meetings, because the previewers must make sure it is adequate before the meeting begins. Also, while we are changing code during meetings, the value and practice of testing can be reinforced.

Finally, our meetings value simplicity. The atmosphere has always been informal. We incrementally add structure only when we feel it is necessary. For example, we have found that we must preview the code. If we do not, the meeting is often spent debugging someone's application configuration, trying to get one test to succeed. Such episodes try everyone's patience and are totally unproductive. Also, by ensuring that there are already working tests, previewing makes it much easier to add tests as needed while refactoring. Finally, it is important that a couple of people have already looked at the code in some detail before the meeting, because the person who wrote the code may not be present and thus be unable to play the narrator role. Having a couple of people who understand the present state of the code helps start discussion. E-mailing the code in the body of an e-mail message to everyone in the group has also been successful in acquainting people with the code before the meeting so that people are more ready to begin discussion. We have found this much structure to be useful. We are always ready to remove any structure that turns out to be unnecessary.

We note finally that we have experimented both with two groups/projectors and one group/projector. One projector is a little easier to





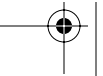
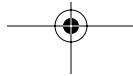
manage logistically. However, because our group is composed of about ten developers, having only one point of focus can make it difficult to include everyone in the process. With two projectors, we can try several different, generally more engaging formats. We can follow the format described earlier: split into two teams, each of which comes up with its own refactoring of the code, which is then compared with the other team's and discussed. We have found this format to be the most successful. We have also tried having one team refactor and the other team write tests. This latter format suffers from the problem that to refactor, you need working tests. Conversely, once you have written the necessary tests for the next refactoring step, often you must wait for that refactoring to occur before it makes sense to write additional tests. One team ends up waiting for the other. In any case, having two points of focus involves more people and shares the roles played more broadly within the group. Also, having two groups increases the diversity of ideas that are presented during the meeting.

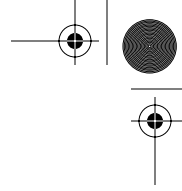
Shortcomings

At the time of this chapter's final revision, we have practiced mob programming about ten times. It has not been an unmitigated success. In this section, we discuss the problems we have encountered.

First of all, we have had trouble following our own guidelines. For instance, it has always been clear that good preparation, in the form of previewing, has a strong effect on the quality of the meeting. However, only once or twice have we found someone sufficiently motivated to complete this preparation thoroughly. This points to a potential weakness of our practice. Mob programming, in some ways, stretches the principle of doing what is natural. For a quality meeting, some discipline is necessary. We are still searching for a degree of discipline that works for our team. It is not clear whether the team will eventually find preparation more natural, or whether we will find a way to avoid preparation. This is an important point. It is not just our guidelines that evolve, but also the team itself. This should not make us uncomfortable. Shooting at moving targets is an integral part of the XP experience.

We have also discovered that people find it difficult to remain focused, especially those who are not driving. The mob plays a role that is not as well defined as the driver role, so it is easy for a mob participant





to feel useless during mob programming. Having more than two programmers in a group leaves the other members of the mob to daydream, to have other, unrelated conversations, and so forth. Unless everyone in the room feels engaged in the practice, the mob inevitably becomes unruly. How, then, could we change the format of mob programming so that we engage everyone? We have not yet found an answer to this question. The lack of engagement may be exacerbated by the fact that the team does not seem to have bought the idea of mob programming completely. At this point the costs are clearer than the benefits. We will expand on this greater issue at the end of this section.

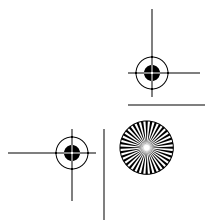
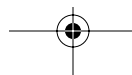
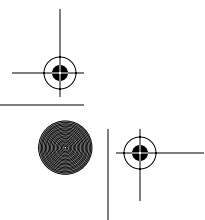
When searching for ways to improve mob programming, we have so far found it difficult to get feedback from the rest of the team. This may just be because we have not been practicing it for very long, and people have not made up their minds one way or the other. However, we also wonder if perhaps this is because the group sees one of the authors (Moses) as the motivating force behind the meetings. We have some evidence that people feel reluctant to criticize the process because criticism may be construed as criticism of Moses.

The Role of “Coach”

One inadequacy of our chapter so far is that we have completely ignored an important role in the whole process, and that is the role of “coach” (to use Beck’s terminology). This role, during and surrounding these weekly meetings, has consistently been played by Moses. All the developers on the team see these meetings and especially see mob programming as “Moses’ baby.”

Moses has made a number of probably classic coaching mistakes that can be summarized as leading too directly. When no one volunteers to act as a previewer, he does it himself. This can lead to this work being done at the last minute and not very thoroughly, because the burden is not shared. Because Moses is excited about seeing these ideas put into action, he often takes too active a role in the process. He is usually one of the drivers during mob programming.

This is clearly problematic. These meetings, in whatever format, will never work if they are not a creative and experimental process for the team as a whole. We have begun to see improvement in this area, however. Several months ago, the meetings would not occur if Moses were out of town or otherwise too busy to organize the meeting. Today this is no longer true. More regularly we observe other developers



suggesting and implementing new ideas for improvement, both during the meetings and outside of them. Progress toward group ownership of this process takes time, and we believe we are beginning to see it happen. New ideas may take us completely away from our current format, but succeeding at mob programming in particular is certainly not the point anyway. It is far more important, and more in line with our goals, that we succeed at doing something owned by the group, developed interactively without one person at its center.

Sometimes, after a particularly unproductive meeting, we wonder whether we should continue having these meetings at all. If they began as unproductive presentations and have resulted in chaotic attempts at mob programming, have we really found anything of value? We believe that the answer is undeniably yes. We base this judgment on our observations of the changes in team behavior not only during developer meetings, but also during our day-to-day operation.

At the first meetings, no one talked except the presenter. Today our meetings are chaotic because everyone talks, and people are usually talking about code. They may not be talking about the code under discussion, but they are talking about code. The fact that at our last meeting four people other than Moses began a discussion criticizing mob programming and seeking a remedy is significant. These people saw the meetings as theirs to control.

Outside the meetings, frequent informal discussion has become markedly more common, even commonplace. We have also become bolder. Recently we used a developer meeting to discuss our level of commitment toward XP practices compared with our current level of achievement of those practices. We decided to speak with our project manager to put a plan in place to carry out the last half of our current release in three two-week iterations, with tasks estimated by the team instead of by a manager. For the first time in the history of our project, most of the development team participated in the planning and feature estimation part of our development process. Our team has become noticeably more courageous, more collaborative, and more open.

We can thus see our meetings, in whatever format, as a “generative” process—a process that “produces the generated quality indirectly” [Gabriel1996]. Meeting together every week to invent and experiment with practices of our own design has changed us as a team. We have not only gotten better at designing and carrying out these practices, we have become closer and more capable as a team. These benefits cannot

be attributed solely to our weekly meetings, but the meetings have played an important role.

Finally, we add that when we actually followed our guidelines for mob programming, the process went relatively well. Also, during periods when we have done less mob programming, we have found that less progress was made toward better and more complete testing.

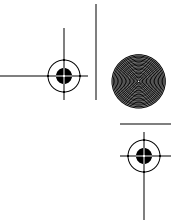
We have decided to continue to pursue mob programming with two improvements. One, we will complete the necessary preparation. Two, the coach will coach more indirectly, yet more firmly (for example, more consistently enforcing the preparation). We have also decided, as a team, to alternate mob programming with other formats. We want to give mob programming a chance to succeed if it can, while experimenting with alternatives.

Conclusion

Our experiment with mob programming, still in its infancy, has so far received mixed reviews. The undeniable benefits of the practice have been more indirect. Communication has been noticeably enhanced. Test coverage is improving. The meetings themselves are becoming more productive, regardless of format. There is a growing sense of ownership of the meetings by the team as a whole. We view these meetings as a work in progress. We do not view the process as it exists today dogmatically, but as a lightweight method of encouraging shared values. If mob programming does not work, we will try something else. We hope, however, that sharing our experiences may help others searching for ways to facilitate the transition to a more people-centered, communication-focused process like XP.

References

- [Beck2000] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.
- [Cockburn+2001] A. Cockburn, L. Williams. "The Costs and Benefits of Pair Programming." In *Extreme Programming Examined*. G. Succi, M. Marchesi, eds. Addison-Wesley, 2001.
- [Gabriel1996] R. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.



[Hohman2000] M. Hohman. Personal e-mail communication. January 2000.

[Williams+2000] L. Williams et al. "Strengthening the Case for Pair-Programming." *IEEE Software*, Volume 17, Number 2, July/August 2000.

Acknowledgments

The authors would like to thank the other members of the ThoughtWorks Cat development team (especially Eric Altendorf, James Newkirk, and Alexei Vorontsov, for their insightful comments), Martin Fowler, Rebecca Parsons (both also of ThoughtWorks, Inc.), and Robert Martin of Object Mentor, Inc., for helpful discussions.

About the Authors

Moses M. Hohman is a software developer for ThoughtWorks, a custom e-business software development consulting firm. Currently, he works out of the recently opened Bangalore, India, office, where he mentors new software developers and teaches a course on agile software development at the Indian Institute of Information Technology Bangalore. He received an A.B. in physics from Harvard University and a Ph.D. in physics from the University of Chicago. Contact him at mmhohman@thoughtworks.com.

Andrew C. Slocum is a software developer for ThoughtWorks. He received an M.S. in computer science from Case Western Reserve University. He is interested in making agile development work on large and distributed projects. Contact him at acslocum@thoughtworks.com.

