Mob Programming in a Software Design Course

Anonymous Author
Department of Computer Science
Anonymous University
Anonymity City
anon@example.com

Abstract

Case study of a Software Design course taught using mob programming. Mob programming (also "team programming", "ensemble programming") is collaborative programming in which the whole team—in this case, the whole class—writes code together at the same time, like pair programming but including everyone. The course combined daily collaborative design and coding with intensive daily readings and oral presentations. Students gained most strongly in skills that are normally hard to teach in colleges: egoless cooperation and "tacit knowledge" of software design.

1 Introduction

I taught a course on software design in the Spring 2025 semester at California State Polytechnic University, Humboldt (course number CS 356), predominantly by a new pedagogy: mob programming. Mob programming, also called team programming or ensemble programming, is "a software development approach where the whole team works together on the same thing, in the same space, and at the same computer." [7] Mob programming is like pair programming, but not limited to two programmers at one computer.

Several reasons suggested mob programming as an effective way to teach software design:

1. Experience from industry suggests that pair programming and mob programming result in knowledge spreading rapidly throughout the team. So why not exploit this happy property of mob programming as a class's principal method of learning?

2. Many industry complaints about skills lacking in recent college graduates revolve around "habitability"—the aspects of design that go beyond correctness and relate to programmers' ability to work with code over many years, such as readability, maintainability, testability, and debuggability [2]. The usual college-course format of lectures, homework assignments, and exams often gives students experience only with writing small, standalone pieces of code that just barely work or partly work (enough to get partial credit) and that no one looks at ever again. Could mob programming on one project, building it for most of one semester by piecemeal growth, give students valuable experience working on each other's code, seeing for themselves how to grow a software design that's habitable for others?

3. Software engineering is a complex and collaborative art. Engineers must constantly learn and constantly adjust to unexpected problems and to other people on the development team. Much of the skill is difficult or impossible to express precisely in verbal representations and procedures, and can be acquired only through real practice. This makes it a good fit for the research by anthropologists Jean Lave and Etienne Wenger on learning in "communities of practice" [5]. Could mob programming create an artificial community of practice for one semester?

4. Only seven students were enrolled in the class. CS 356 is a junior-level course, and these students were among the first cohorts in the university's recently begun Software Engineering degree program. So, this semester provided a good opportunity to experiment with mob programming.

The rest of this article describes how the course went: its structure, problems that arose, solutions that we found or didn't find, and a final assessment.

2 The Semester

2.1 Semester Overview

The semester plan was as follows. (The complete syllabus is available at [4].)

- Each class session was a lab—no lectures, except for an introductory lecture on the first day and occasional improvised mini-lectures as needed. Sessions were one hour and 50 minutes long, twice a week.

- Readings were assigned at the end of each class session, and students had to give brief presentations on the readings at the start of the next session. These presentations usually took up the first 20 to 30 minutes

of class time. The readings came only from sources influential among professionals, such as books by Martin Fowler and Scott Meyers, as well as famous blog posts and journal articles, not from textbooks.

- Students were required to keep notebooks in which they recorded the ideas or techniques they learned in each class session, and to periodically hand these in during the semester.
- The first few class sessions: design exercises, mostly omitting implementation, to teach UML and design as something distinct from code.
- Most of the semester (about 10 weeks) was spent designing and implementing a single piece of software in C++, collaboratively by mob programming together in class.
- Last two weeks: we returned to design exercises without implementation, now introducing distributed system design (load balancers, microservices, etc.).
- Last day of class: each student gave a presentation on what they learned during the semester.

2.2 Initial Design Exercises

The first design exercise was borrowed from a job-interview question, involving designing a state-transition diagram for a controller in a common home appliance (details omitted so I can reuse it). The students did this in about an hour of class time, after which I corrected errors and pointed out that they had worked out a mathematically complete description of the software without writing a line of code.

Then we moved on to implementing the finite state machine in C++. Had this been an ordinary class, I would have written the code at a projector, explaining my thoughts as I went. Instead, we had our first experience with mob programming, which proved much more educational. We set up our mob "rotation" as follows:

- One person, called the "driver", sits at the keyboard of the computer hooked up to a projector.
- One person, called the "navigator", tells the driver what to do next.
- Everyone else ("the mob") watches, checks the code for errors, sees what ideas come to mind, and does anything else they think might be helpful, such as adding to-do items to the whiteboard or searching the Internet for documentation.

- We followed Llewelyn Falco's "strong-style pairing" [1], which has this rule: "For an idea to go from your head into the computer, it must go through someone else's hands." The driver does not enter his or her own ideas into the code. The driver's job is to follow the navigator's instructions. The navigator tries to state each instruction at the highest level of abstraction that the driver can implement. That may take the form of "Now write a test for the null case" or, if necessary, specific keystrokes to help the driver operate an unfamiliar text editor.

- We set a timer so that every 5 minutes, we rotated roles. Someone from the mob becomes the driver, the driver becomes the navigator, and the navigator returns to the mob.

- All the students' names were written in sequence ("the rotation") on the whiteboard, so we always knew who was to drive next. I placed myself eighth (last) in the rotation so that every student would have a chance to navigate before I spoke up.

We drew the state-transition diagram on the whiteboard, and then the students proceeded to do exactly what I think you should not do: they started putting together tricky nested loops with tricky conditions inside—essentially neglecting the state-transition diagram and trying to reproduce the desired output by cobbling together familiar programming constructs in the usual ways.

At one point, the navigator suggested "reversing" a variable for a sensor's input. The driver didn't know how to do that in C++. The navigator said, "Put an exclamation point" (the Boolean negation operator). The driver was surprised by how easy that was, got a brief explanation, and then went on to the next thing. Similar small points came up during mobbing nearly every day. Students came to the class with widely varying prior experience. Some might be lost in a lecture that assumes knowledge they don't have and get further behind as the semester continues. Mob programming effortlessly paused to fill in just the bit of knowledge that a student needed, right when he or she needed it.

Eventually the students hit on the idea of a variable for the state of the state machine. From the mob, I suggested a `switch` statement, but they still had no systematic approach and were writing extremely bug-prone code. Finally, as the navigator, I had the driver make a `case` for the `switch` statement that fully handled the initial state. The students had "aha!" moments and, as they rotated into the navigator role one by one, worked out how to systematically cover all of the infinite set of possible input sequences. On our first attempt to compile the code, it was correct and passed all tests (which the students were now able to design systematically, too). The students had their first

experience with *design* distinct from code, and with a systematic mapping of design elements to program elements.

Some students found the initial mob programming stressful. They'd written almost all of their previous code solo. Now they had to write code in front of a whole class—while thinking it up, unsure of their approach, gaps in their knowledge and skills on display. "Am I doing it right?" some asked me after class, unsure, since I gave them no rubric for "correct" mob programming.

A small problem with the physical environment became apparent: the tables in the lab room were far apart and perpendicular to the projection screen, so it was awkward to look at the screen, and background noise made it hard to be heard across the room. We eventually found a quiet room where the seats faced the projection screen and a new driver could swap into place in a few seconds. Mobbing can be done hovering around a laptop, but these conveniences help.

After that, we moved on to designing an ATM machine, following the method of making classes from nouns in the requirements and assigning responsibilities based on relationships, as explained in [6]. We liked the mob-rotation format so much, we used it for this UML-only, whiteboard-only design process. There was no navigator role, only a driver role that rotated every 10 minutes. The driver wrote on the whiteboard while everyone else suggested items to add or change. Since I was the only one who knew the process, I facilitated.

2.3 The Project

Before the semester, I had asked around campus for software that people needed written and even received some suggestions on LinkedIn. In class, we rated all the options at the whiteboard for feasibility, opportunity to learn design patterns, and suitability for C++, and chose to make a diagram editor. In this as in all our group decision-making, we followed a method similar to "dot voting" as used at Norman Nielsen Group [3] and commonly done in mob-programming circles. We kept a clear boundary between exploration ("divergent thinking") and evaluation or filtering of ideas ("convergent thinking"), doing first one and then the other.

We spent one class session brainstorming for features. This gave the students experience in collaborative software design at the level of requirements or product definition. The first idea was "little boxes in a canvas". Exploration continued through "TiKZ editor" and, most ambitiously, "the VSCode of diagram editors, allowing plug-ins for electrical simulation of schematic diagrams or anything else". We pared that down to "enough for Ben to draw graphs for homework problems in graph theory, with a *vim*-like user interface."

Next we had to make a crucial design decision: which GUI library to use? I assigned readings on two popular and relatively simple libraries, FLTK and

SDL, intending that the students would choose one of those. One student independently researched Qt, a much more complex library—which I had intended to avoid. After presentations on all of them and another group discussion, we chose Qt, attracted by its ability to make a sophisticated GUI in C++ very quickly and run it on both desktop and mobile platforms, deeming this to outweigh the risks of greater complexity and time to learn.

I assigned readings in the Qt documentation and we got back to mob programming. Qt did turn out to be problematic. We spent a lot of time in and out of class getting the Qt libraries to install and work with VSCode on the students' various computers and operating systems. This was chaotic and frustrating—but so is real-life software development, often for reasons just like this. Qt did enable us to quickly and easily try out out radical new user-interface ideas, something we likely could not have done with the other libraries.

We found that mob programming can work at amazing speed—even as it feels like you're working slowly, doing just a tiny bit before passing the baton to the next person. One day, for example, after we implemented snap-to-grid as the user moved an object, we had five minutes left, and our next item was to make the grid visible—too little time, I thought. But the next navigator/driver team dove in, and with a little help from the mob got Qt displaying a visible grid before class was done.

We experimented with the rotation timer and found that we liked a 7-minute rotation best. That was long enough to let the navigator and driver settle in and still allowed each student to get two rounds in during the class period.

One day, a student mentioned that he had no idea how to implement Undo. I assigned readings on the Command and Memento patterns, and the next day, after oral presentations, the class weighed them at the whiteboard and chose Memento because it seemed simpler to implement: just store a vector of snapshots of the current state of the diagram. By the next class session, we found the flaw: coordinating ownership of Qt's diagram objects was unmanageable. It emerged that the students had never fully grasped that in C++, unlike in Java or Python, you distinguish between holding a reference to an object and holding the object—and the pitfalls of having to bear this responsibility. We deleted the code and reimplemented using the Command pattern—and found not only that it worked correctly, it was actually simpler.

This experience—completely unplanned—implicitly taught a fact about software design that is easy to say but hard to believe until you've experienced it: it's often difficult to tell whether a design idea will be good or bad until after you've implemented it. In hindsight, it's easy to say that we "should" have chosen Command on the first day, but of course that's only in hindsight.

2.4 Testing the edges of mob programming

The Memento/Command revision prompted us to stray from strict mob programming and stop the timer for about 45 minutes—the time needed to explain and reach consensus about the problem, including understanding that Qt was written long before C++11's `unique_ptr` and `shared_ptr` and follows different conventions for ownership of dynamic memory.

The navigator role turned out to induce the most stress. As driver, you're sitting at the keyboard typing code in front of everyone, but as navigator, you're standing in front of the everyone and you bear the responsibility of directing the work. Sometimes a student didn't have ideas yet, or his or her ideas weren't yet clear enough to tell someone else how to implement them. Our mobbers tended to call out ideas while the navigator was still gathering his or her thoughts. Our solution: When you're the navigator in this position, ask, "May I have a moment of silence, please?"

We found that some design ideas required more thought, or a different kind of thought, than happens during mob programming. Sometimes, just getting an idea across takes longer than one rotation. So, on some days, I assigned to one person not a reading, but a design problem and a first implementation. In the next class session, that person gave an oral presentation on what they made, and then the class as a whole reimplemented it from scratch. This produced a result that was often simpler and was of course understood by the whole class, since every student literally had their hands in writing it.

2.5 AI-Assisted Programming

The class's AI policy was to use AI mainly for documentation look-up and simple code-completion with Copilot during the first half of the semester, and during the second half, to run as wild as we could with AI and see all that it can do. Right after Spring Break, a student suggested writing a short document describing all the features we wanted and letting ChatGPT rewrite the entire diagram editor from scratch.

So, we mob-wrote a user-interface design document in a wiki page and let ChatGPT rip. Its code did not compile, nor did revising the document to tell it not to make those errors again fix it. We manually fixed the problems and made some rapid progress.

But then the class rebelled! The students found that the fast cycle time was so fast, they did not understand the code that ChatGPT was producing. They wanted to understand the programming and design ideas, not merely see them rushing by. So we reverted to our AI policy from the first half of the semester, and actually used Copilot more sparingly.

3 Assessment and Conclusions

3.1 Results

The students reported two main things that they learned from the class:

1. How to collaborate. The students had all done some pair programming in earlier classes, but they had never seen collaboration at this intensity. I had meant for mob programming to give the students broad practical experience with software design, and it did, but collaboration itself took the starring role.

2. The parts of software development that are too hard to put into words. For example, how much forethought should you put into code before writing it? How quickly does refactoring lead a good design to emerge "spontaneously" and under what circumstances does this happen or fail to happen? How much design is overdesign?

The students reported that they got extraordinary practice in oral communication, partly from the daily oral presentations and partly from the discipline of verbalizing every design decision that mob programming imposes. Mob rotation circumvented the common problem of vocal students dominating a class and quiet students not asking about their concerns.

The biggest disappointment of the course was that 10 weeks was not enough to implement a satisfactory "graph editor for homework assignments". Ten weeks sounds like a long time, but meeting twice a week, that amounts to only about 30 hours of mob programming. That's less than a week of work at a regular job, plus the students were learning many unfamiliar concepts.

Another disappointment was that the students got only a little exposure to unit tests and test-driven design. The students all made a test or two in GoogleTest, but we never used it in the diagram editor because unit-testing a GUI is hard. I would recommend in future mob-programming courses to implement a nontrivial back-end for a web site with a simple interface, or simply a program that has only a textual user interface.

The course was in some ways easy to teach and in other ways hard to teach. The students did most of the lecturing, not me. However, the course also required me to pay close attention and adjust continuously and creatively throughout the semester, more than in other classes. While I had selected a set of readings before the semester, I had to choose and search for readings before each class to suit whatever the programming was leading us to next.

The syllabus listed 78 topics of which I hoped to cover "some large subset". I count about 55 of them touched on or explored to varying degrees: the DRY principle, *git* and version control, refactoring, code smells, information-hiding,

Model–View–Controller and other design patterns, adding a level of indirection, writing the calling code first, YAGNI, coupling and cohesion, and many more. Most of these topics came up naturally, without lectures, as they arose in the course of writing code and talking about it; some needed readings.

The course's focus on tacit knowledge gained through experience made designing the final exam tricky. Nevertheless there was much explicit knowledge that could be tested: designing a finite-state machine to solve a problem, making UML diagrams given requirements for a simple distributed system, articulating a reason for an opinion about software design. The final exam demonstrated weakness in the students' understanding: most answers to the "hard" problems had some serious flaw. Perhaps that's to be expected in a course with more breadth than depth, but I would recommend in future mob-programming courses that a homework assignment to be done solo should follow each central topic, even if this means going a day without new readings.

Course evaluations averaged 4.8 on a scale of 0 to 5, the highest I've received as a teacher. I am most pleased that the students *know* what they learned in the course. Their knowledge is imperfect, but they know it first-hand, not "because I was told that in college."

3.2 Risks and Unknowns

Three important unknowns are unaddressed by this experience. First, what if a student is uncooperative? An occasional problem in industry, more common in people of college age, is the intransigent "prima donna"—the opinionated programmer who knows better than anyone else, sees faster than anyone else what to do, and sees no value in patiently working to build consensus.

Second, what if a student is unable to keep up? Programming is hard, especially at the nearly professional level reached in this course, and not all students have the talent or prior knowledge needed to do it. Some students hold back and let their teammates do most of the work—a common problem in group projects, even at the senior level.

In ordinary classes, a few unpleasant or lazy students don't bring down the class. But in an intensely collaborative class, they could spoil the experience for all. I was fortunate to have a class filled completely with cheerful, dedicated, cooperative students; not all mob-programming classes may be so lucky.

And third, could the mob-programming approach be brought to a larger class? A mob of eight is already quite large, requiring each student to wait a long time between turns as driver; a mob of 30 would be impossible. One idea is to have multiple mobs of about 4 students, with the teacher and/or students "floating" between different mobs, carrying information along and being brought up to speed upon joining each new mob. Would this require multiple projectors? If so, the expense might be prohibitive.

4 Acknowledgements

Much gratitude to the many people who helped make the course a success: Shahzad Aslam-Mir for wide-ranging advice on everything from projects to tools to C++ libraries; Rebecca Wirfs-Brock for inviting me to look into mob programming and guidance finding the right balance between giving too much and too little direction to the students; Woody Zuill for generously making time to answer my many basic questions about mobbing; Austin Chadwick and Chris Lucian for generously making time to answer yet more questions about mobbing and letting me mob remotely at Hunter Industries; Dave Bender for the initial design exercise; Sherrene Bogle for creating the original CS 356 course, making this experiment possible; and most of all to the students, who dove boldly, cooperatively, and creatively into a wildly experimental course.

References

[1] Llewellyn Falco. *Llewellyn's strong-style pairing*. Blog post. June 2014. URL: `http://llewellynfalco.blogspot.com/2014/06/llewellyns-strong-style-pairing.html`.

[2] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. New York: Oxford University Press, 1996. ISBN: 9780195102697.

[3] Sarah Gibbons. *Dot Voting: A Simple Decision-Making and Prioritizing Technique in UX*. `https://www.nngroup.com/articles/dot-voting/`. Nielsen Norman Group, Feb. 2024.

[4] Ben Kovitz. *CS 356, Software Design, Spring 2025: Course Syllabus*. 2025. URL: `https://github.com/bkovitz/cs356-sp25/blob/master/syllabus/syllabus.pdf` (visited on 07/04/2025).

[5] Etienne Wenger-Trayner and Beverly Wenger-Trayner. *Introduction to communities of practice: A brief overview of the concept and its uses*. 2015. URL: `https://www.wenger-trayner.com/introduction-to-communities-of-practice/`.

[6] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990. ISBN: 0136298257.

[7] Woody Zuill and Kevin Meadows. *Software Teaming: A Mob Programming, Whole-Team Approach*. 2nd. Independently published, 2022. ISBN: 9798361186938.