The Joy of Coding

> Contemporary software development is seen as a “team sport” in which multiple people in multiple roles contribute to outcomes that no individual could achieve alone. Even programming, long considered a solitary pursuit, sees benefits when multiple people work in real time to make decisions, write code, and validate results.

Working with Others to Develop Software

- Pair Programming
- Ensemble (“Mob”) Programming

Collaboration in Software Development

Agile Software Development promotes collaboration to enable rapid feedback loops

- People with different roles (customer, development, quality, etc.) work together to constantly understand, implement, and refine
- Unit tests and continuous integration provide feedback on code changes shortly after they occur
- The user’s experience is assessed and determines what is done next
- Different people in different roles contribute their points of view to inform decisions

People work together all of the time to get work done and learn from the experience.

Putting some structure in place to guide collaboration helps, especially in the beginning.

All outcomes (positive and negative) result from the *behaviors* of the individuals that contribute to those outcomes. Acting deliberately will help influence outcomes.

Pair Programming

Pairs of people have coded together since the beginning of computing. The term “pair programming” was first popularized in Kent Beck’s “Extreme Programming” (XP) book in the late 1990s.

- Two people sit in front of the screen
- Together they discuss and explore the problems that need to be solved
- The working model is very fluid and dynamic
  - Fits well with complex problems
- Just enough structure to accomplish goals without becoming exhausting or confining

Roles in Pair Programming

Traditionally, there are two roles in Pair Programming, the “Driver” and the “Navigator”

The “Driver” is the “hands” on the keyboard who writes the code and is only focused on the next task

- The Driver acts tactically: types on the keyboard and talks about the code they’re writing

The “Navigator” is the “brains” who observes what is written and offers feedback

- The Navigator thinks strategically about the larger picture and is on the lookout for code that could be refactored

Pairs exchange roles every couple of minutes to avoid exhaustion and provide multiple perspectives on the work

- Pairing is most effective when the pair works on very small goals one after another

Avoid “analysis paralysis” and deep diving by having a bias for action

Potential Benefits of Pair Programming

Knowledge is shared and build across individuals

- Look for opportunities to pair with others who have different expertise than you do
- New team members pair to learn about the code
- This is why pairing doesn't double the cost of coding. There's a lot more to programming than typing!

Thoughts are articulated verbally to create understanding

- Communicates the "why", "what", and "how" of the code
- Requires safety to speak up when you don't understand (and safety to offer feedback)

There are fewer distractions and detours when pairs focus

- Constant communication helps avoid rabbit holes

Code is constantly reviewed and refactored

- Feedback comes early. Easier to maintain discipline.

Potential Benefits of Pair Programming

Both tactical and strategic thinking occur concurrently

- Increases quality because entire picture is in view

Team resiliency is built through collective code ownership. Removes silos and bottlenecks.

- More people feel comfortable working with the code

Removes multi-tasking to maintain focus

- Fewer things start and more things finish
- Minimizes work in progress (and keeps it visible)

Leverages the natural diversity on the team

- People with different backgrounds and experiences must be more explicit about their actions in order to move forward together
- Results in simpler and better-understood outcomes

Patterns for Pair Programming

"Strong-Style Pairing" emphasizes knowledge transfer

- Helpful when the Navigator has much more experience than the Driver
- More "Driver's Ed" than "Road Trip"
- "For an idea to go from your head into the computer it **must** go through someone else's hands"
- Navigator gives instructions as soon as Driver is ready to follow them
  - Driver does not "take dictation" for the Navigator
  - Navigator talks in the highest level abstractions that the Driver will understand
- Driver must be comfortable with being inexperienced with the tools and terminology
- Inevitable "why?" questions and challenges to solution are tabled until after pairing session completes

Patterns for Pair Programming

"Ping Pong" pairing emphasizes Test Driven Development

- One Developer writes a unit test. The other Developer makes it pass.
- The pair refactors together

"Pair Development" emphasizes pairing in all aspects of software development

- Establishing a larger goal for the work
- Understanding the problem. Checking your understanding against your "definition of ready"
- Exploring potential approaches to possible solutions
- Researching unfamiliar concepts, tools, and techniques
- Documenting decisions and what's been learned

Pairing is Intense. Pace Yourself.

Working with someone else often requires a different kind of energy than working alone. It can be draining.

- Consider the “Pomodoro Technique” in which work is time-boxed to 15-25 minutes with breaks (short and long) in between
- Over time, rotate new people into the pair. Don’t pair with the same person forever.
- Have the courage to express your feelings (tired, confused, frustrated, bored, etc.)
- Take time to reflect as a pair on the experience
- Acknowledge that talking with others is very draining for some people
- When you’re stuck, working on a different task might provide new insights
- Make sure that the working area is comfortable
- Celebrate what you accomplish

Common Pitfalls when Pairing

Be on the lookout for:

- Navigator is silent (or can’t get a word in) while Driver does all the typing and talking
- Navigator tries to grab the keyboard. Driver easily relinquishes it.
- More experienced Driver moves fast and Navigator can’t keep up
- Planning farther ahead than the next move. You’ll miss opportunities to learn.
- Only talking about options and not trying them out
- Going too long without switching roles or resting
- Pairs that become impatient with each other
- Organizational or cultural power dynamics
- Pairs think they’re invulnerable and don’t acknowledge what they don’t yet know
- Natural friction that is not resolved. Pairs don’t develop techniques to create a safe environment.

When does it make sense to (not) pair?

Boring or rote tasks won’t benefit from pairing because they are already well-understood

- These tasks might be good candidates for automation, which can be implemented using pairing

When learning, trying something by yourself can give you good feedback

- “Can I really do this by myself?”
- Trying pairing with others with similar amounts of experience

Code reviews provide later feedback than pairing

- The nature of ownership is different with an after-the-fact reviewer: “gatekeeper” or “rubber stamper” instead of collaborator
- Might be afraid to suggest rework because the code is already “done”
- Code reviews can be distracting to individuals and to the team

Mob and Ensemble Programming

Pair Programming is a product of its time

- Different roles and focuses in software development were purposefully isolated
- The team-based approaches of Agile approaches had yet to be popularized

In the 2010s, “Mob Programming” was codified

> “All the brilliant minds working together on the same thing, at the same time, in the same space, and at the same computer”

Mob Programming evolved from real developers building real software who adapted Pair Programming practices to focus the entire team solving real problems.

- Several people contribute to a solution in real time
- A spirit of “do what works right for you and your team”
- Can be applied to work beyond programming (research, testing, etc.)

Similar team programming techniques are described as “Ensemble Programming”

Mobbing to Achieve High Performance

Multiple people often “swarm” on complex problems

- Break down a complex task into smaller ones
- Rallying around simpler problems leads to innovation and automation
- A team will not tolerate toil like an individual will
- Why not “swarm” on all of the problems?

Working together provides immediate feedback and accelerates learning

- The best ideas of all team members make it into the solution
- Gaps in knowledge are filled for all team members
- The cognitive load is spread among multiple people
- A high-quality result is achieved sooner

Words are hard

We use them interchangeably here, but there are some largely unintentional connotations of the words “Mob” and “Ensemble”

“Mob”	**“Ensemble”**
Dynamic	Controlled
Chaotic	Organized
Leaderless	Conducted
Riotous	Collegial
Jazz Music	Classical Music
One syllable	Three syllables
	Translates to non-English better

These connotations aren’t “good” or “bad”, per se.

Reality is somewhere in the middle.

The techniques, values, and attitudes ascribed to “mob” and “ensemble” programming are essentially the same.

Getting Started with Mob Programming

A little structure can help get Mob Programming off to a good start

- Team size of about 6 people working on a small problem (e.g. kata)
- Participation is voluntary
- One computer projected where everyone can see the same thing
- Comfortable chairs, multiple keyboards, multiple mice
- Rotate roles every few minutes to give everyone a chance to code (and a chance to **not** code)
- Try to show kindness, consideration, and respect
- Reflect and retrospect after each session

Remote Mob Programming can be very effective and requires some slightly different rules and tools

Who, exactly, is part of the mob?

It depends: Whoever feels they are contributing value can be part of the mob?

- Whoever has the knowledge to contribute to decisions, implement them, and validate the results
- People can come and go, as is effective
- If someone is no longer adding value, they should leave the mob
- Developers, testers, product experts, operations engineers, domain experts, facilitators, etc. can be part of the mob
- After a certain size, a large mob will lose some of its effectiveness

Programming is about translating user needs into working software. Writing code is a very small part of that.
There’s plenty of work to go around.

One Driver, Multiple Navigators

Mob and Ensemble Programming have multiple Navigators

- Navigators think, discuss, diagram, decide, and share ideas of how to move forward (“Brains”)
  - Navigators take turns talking. Not a free for all.
- The Driver translates those ideas into code (“Hands”)
- A “Designated Navigator” channels the ideas from the other Navigators to the Driver (“Voices”)
  - The “Designated Navigator” may be referred to as the “Navigator” and the other Navigators as “Mobbers”

Every couple of minutes, roles change

- Driver becomes a Navigator
- Someone else becomes the Driver and the Designated Navigator

Guidelines that Inform the Mobbing Mindset

Some simple rules help the Ensemble work together

- Rotate on a timer
- “Yes, and” to continue and improve current work
  - A new Driver shouldn’t immediately throw away work in progress
- No decisions are made at the keyboard. Navigators determine what the Driver codes.
- Navigators speak in the highest level of abstraction possible.
- Bias to action. Better to experiment than to ponder.
- At all times, each participant should be learning, contributing, or both.
- Show kindness, consideration, and respect for all involved.
  - Being *kind* isn’t always the same as being *nice*

The Values that Inform Mobbing Behaviors

Kindness

- Be warm-hearted, gentle, pleasant, and polite towards others
- Show genuine concern for (and interest in) those around you

Consideration

- Listen to others with humility. Hold your ideas loosely.
- Work to figure out *how* to try other ideas, instead of reasons to not try them.

Respect

- Act without judgement to allow others to maintain their self-esteem
- Disagree respectfully and in a way that lets ideas flourish.

Behaviors that Inform the Mobbing Mindset

Strive to build an environment where individuals feel safe expressing their ideas without fear of ridicule or retribution

- Acknowledge good and bad. Work to amplify the good.
- Driver refrains from making decisions while at the keyboard
- Call out peoples’ ideas and give credit for ones that work
- More-experienced Navigators yield to less-experienced Navigators
- Try as many ideas as you can. Start with less likely candidates.
- Stay focused on the ensemble. Take a break when you need one.

Frequently retrospect on the team’s experiences: identify areas for improvement and inform how to move forward

The Driver in an Ensemble

A Driver is an “intelligent input device” who can help the Ensemble by

- Asking clarifying questions and requesting navigation on a different level of abstraction
- Typing some controversial code to focus attention
- Trying a new keyboard shortcut or a new tool
- Ignoring instructions from non-Designated Navigators
- Being a “sponsor” who amplifies voices of least privilege
- Being the “nose” who highlights duplicate, overly complex, or hard-to-read code

Behavioral Patterns for a Mob’s Driver

It’s difficult for the Driver to be merely a “typist” and not exert considerable influence over the mob’s direction

Understanding these patterns (and anti-patterns) may help a Driver be more effective in their role:

- “Plowing Through”
- “Driving on Autopilot”
- “Thinking Out Loud”
- “Tell Me What to Write”

Behavioral Patterns for a Mob’s Driver

A Driver may be “Plowing Through” when they silently write code without taking input from the Mob

- More likely when the Driver is more experienced or passionate about the work
- Do others in the Mob agree with the approach? Are they confused? Have the lost focus?
- Likely to result in “Distracted Non-Participants”

Similarly, a Driver who is “Driving on Autopilot” ignores input from the Navigator to implement what they believe to be correct

“Thinking Out Loud” might be a good antidote to “Plowing Through” and “Driving on Autopilot”

- Driver articulates (narrates) their thinking as they type
- Actively checks for understanding with the Mob

Behavioral Patterns for a Mob’s Driver

When only one or two people on a mob have context on the problem, “Tell Me What to Write” can help a Driver understand the context quickly

Driver uses simple prompts like:

- “Where is the code that needs to change?”
- “What class performs that action?”
- “How should we test this?”

When answering these questions for the Driver, be careful to speak at the high-level abstraction that the Driver is familiar with

- If the Driver knows how to write the code, don’t treat them like a stenographer

The Navigator in an Ensemble

The (Designed) Navigator can help the Ensemble by

- Filtering ideas from everyone to provide high-level intent to the Driver
- Describing a failing test and guide the Driver towards implementing a passing one
- Being a “researcher” who finds and shares relevant information from the outside (e.g. Internet)
- Being an “automationist” who points out repeated processes or code
- Being a “conductor” who enhances other contributions
  - E.g. Seek out ideas from people who are quiet or who have the least privilege
- Proposing a bad idea to solicit better ones

The Other Navigators in an Ensemble

The other Navigators (“Mobbers”) can help the Ensemble by

- Contributing their ideas and making room for less privileged voices to be heard
- Asking questions until they understand
- Being the “rear admiral” who gently guides the Designated Navigator towards potential paths forward
- Recording alternative approaches so that they are not forgotten
- Being the “archivist” who describes observations and insights about how the Ensemble is performing

Many times the “slowest” Navigator provides the most insight

Recognize that *everyone* contributes to the solution and deserves credit for work well done

Advanced Mobbing Roles

As more people experienced Mob Programming, common and recurring behavioral patterns were codified into additional roles

Researcher	Sponsor	Rear Admiral
Automationist	Nose	Nostalgist
Archivist/Recorder	Traffic Cop	Anthropologist
Disciplinarian	Major Pain	Doctor Feel Good
Facilitator	Devil’s Advocate	

These more advanced roles should be introduced once people have experience working in the fundamental three roles (Driver, Navigator, and Mobber).

Remember that individuals switch roles every few minutes. These roles aren’t “always and forever”.

No mob will have all of these roles all of the time. It would be too big to be effective.

Advanced Mobbing Roles

A “Researcher” helps by looking outside of the mob for information left by others

- Looks in documentation, blogs, coding forums, etc.
- Looks in other parts of the code for clues

A “Sponsor” amplifies voices to uncover blind spots and unlock greatness in others

- Amplifies the unheard voices by encouraging participation
- Supports the contributions of the mobber with the least privilege (gender, race, class, seniority, etc.)
- Celebrates moments of excellence within the ensemble

Advanced Mobbing Roles

A “Rear Admiral” quietly advises the Navigator to move them forward through the problem

- Often a Rear Admiral is more experienced than the Navigator
- Gives the smallest cue necessary
- Speaks at the highest level of abstraction that the Navigator can work with

An “Automationist” identifies opportunities to replace toilsome work with automation and amplify the overall capabilities of the mob

- Points out repeated coding/tool actions or aspects of a process
- Notices boilerplate code that the Driver duplicates often
- Proposes that the mob implement automation

Advanced Mobbing Roles

A “Nose” is on the lookout for “smelly” code that adds complexity

- Points out long lines of code, complex conditionals, multi-screen methods, awkward names, etc.
- Proposes actions (often code refactorings) to clean the code

A “Nostalgist” encourages the mob to reflect on their experiences to identify how they want to work going forward

- Proposes the mob take a quick break to retrospect on their work
- Helps the mob identify the effective things they want to continue and ineffective things they want to diminish

Advanced Mobbing Roles

An “Archivist” (also known as a “Recorder”) makes the mob’s work, ideas, and discoveries visible for the present and for the future

- Articulates the task at hand to create shared understanding among the mob
- Expresses and records solution alternatives in a visible place (whiteboard, shared document, etc.) so that they are not forgotten
- Captures design decisions and other technical details (especially the “why” behind them)
- Notes external resources that informed the mob’s work
- Creates future plans for testing and deployment
- Sketches out the first draft of the user/technical documentation

Advanced Mobbing Roles

A “Traffic Cop” helps establish (and enforce) the mob’s “rules of the road”

- Suggests a relevant new way of working
- Points out when the mob is not working in the way it said it wanted to
- Reminds the mob of the current task at hand
- Captures technical decisions and expectations for working in the future
- Notices when the mob gets fatigued and suggests a break

An “Anthropologist” maintains the history of the team to enable future learning

- Facilitates reflection on a new mob habit
- Captures knowledge and learning in an accessible artifact

Advanced Mobbing Roles

A “Disciplinarian” reminds the mob about their technical practices

- Encourages strong test-driven development practices
  - First write a failing test, then make it pass
  - Refactor code, but not until the tests are green
- Alerts the mob when the code isn’t clean
  - Method and variable names are unclear
  - Code segments are duplicated
  - Code styles are not consistent

Advanced Mobbing Roles

“Major Pain” highlights pain felt in processes, interactions, or tools

- Shows empathy while asking about things that appear painful
- Amplifies (repeated) pain that is being ignored by the mob
- Calls for a retrospective to inspect the pain and adapt

“Doctor Feel Good” amplifies the mob’s good and effective behaviors and result

- Shows appreciation to a mobber or stakeholder
  - Articulates the *situation*, *behavior*, and *impact*
  - “When we were all stuck on how to use the `File` API, it really helped that you posted the link to the documentation. As a result, we were able to fix the broken unit test quickly.”
- Inquires about happiness expressed by others
- Calls for a retrospective to explore how to amplify the good

Advanced Mobbing Roles

A “Facilitator” guides the mob (especially new mobs) towards working better together

- Encourages participation by all members of the mob
- Keeps the mob on task and calls out when goals are not clear
- Monitors energy and calls for a break

A “Devil’s Advocate” purposefully contradicts the prevailing wisdom or current approach to ensure best possible outcomes

- Questions the reasoning behind designs and implementations
- Advocates from a place of respect and kindness
- Often has to remind the mob that they are being the Devil’s Advocate

Mobbing Collaboration Patterns

In addition to roles, several collaboration patterns (and anti-patterns) have emerged from the experiences of effective mobs

Patterns	Anti-Patterns
Punch List	Ridin’ Shotgun
Splinter Group	Fight Club
Mute Your Mic	Distracted Non-Participant
Natural Swap	
Forced Swap	
Forced Break	
Pause for Clarity	

“Punch List” Collaboration Pattern

When ideas for what to do next pile up, capture them in a “punch list”

- Helps prioritize which items are explore next, tomorrow, and never
- Empowers the mob to decide where to focus their efforts
- Visualizing where a task impacts the codebase can help identify and group similar work

“Ridin’ Shotgun” Collaboration Anti-Pattern

The term “Ridin’ Shotgun” refers to the person who sits in a car’s passenger’s seat and has access to many of the controls (temperature, radio, etc.)

This anti-pattern appears when an individual dominates the mob

- Easy to do when someone is very knowledgeable or passionate
- May result in “Distracted Non-Participants” who are present, but don’t participate
- Discourages questions and decreases learning
- Even if it helps in the moment, can be detrimental in the long term
- May devolve into a “Fight Club” in which guiding principles of mutual respect and consideration are discarded

“Mute Your Mic” Collaboration Pattern

Sometimes it makes sense for an individual to step back from their involvement with the mob

- One or two people are dominating
- The rest of the mob is quiet or deferring to one voice

Explicitly call out that you’re “muting your mic” (or request that someone else “mute” theirs)

- Good opportunity to take on a different role (e.g. Recorder or Researcher)

“Pause for Clarity” Collaboration Pattern

In some situations, the mob must pause and regroup

- Mob is blocked from making forward progress
- The current solution is no longer fit to purpose
- Mob realizes it doesn’t understand the problem well enough to code a solution
- The code is getting messy
- It’s difficult to articulate the solution it wants to pursue

Mob moves away from the keyboard and to a whiteboard or collaboration tool

Sometimes, the mob needs to take a longer break to refresh and let ideas percolate into a better solution

Patterns for Changing Mobbing Roles

Consider the following patterns for changing who is the Driver

- A “Natural Swap” when
  - The Driver asks for someone else to drive
  - Navigator asks for the keyboard
- A “Forced Swap” when
  - The timer has expired
  - Energy is low (“Distracted Non-Participants” begin to appear)
  - Driver wants to share knowledge with someone else

Consider a “Forced Break” when energy is waning or the Facilitator senses that the mob needs to recharge

How can Mobbing Possibly Work?

Mobbing removes many of the causes of lost productivity

- “Context switching” to other work (distractions)
- The bad code you write when you’re tired
- Waiting on others (or having to call a meeting) to get questions answered or decisions made
- Unclear or missing requirements
- Defects and missing tests
- Providing status to other team members
- Long-unresolved interpersonal conflicts
- Waiting for review/approval to integrate code
- Insufficient tools and environments

The focus is on being *effective* (getting work done) than being *productive* (doing work)

Final Thoughts

Great results come from people who work together to deliver the best software they can, as fast as they can. In addition to tools and technologies, *how* people work together determines success.

A structured approach to Pair and Mob/Ensemble Programming can

- Spread knowledge and learning to maximize the capability of the greater group
- Speed decisions to identify innovative solutions faster
- Establish a sustainable pace of work
- Strike a balance between a solution that is “not enough” and one that is “too much”
- Reinforce effective practices (technical and behavioral) in the moment (and not just in hindsight)
- Focus collaboration, communication, and documentation on the things that matter

Sources and Further Reading

Pair Programming

- https://martinfowler.com/articles/on-pair-programming.html
- https://llewellynfalco.blogspot.com/2014/06/llewellyns-strong-style-pairing.html

Mob and Ensemble Programming

- “Mob Programming: A Whole Team Approach” by Woody Zuill and Kevin Meadows
- “Ensemble Programming Guidebook” by Maaret Pyhäjärvi https://ensembleprogramming.xyz
- “Mob Programming RPG” by Willem Larsen https://github.com/willemlarsen/mobprogrammingrpg
- “The Mob Mentality Show” podcast
- https://www.agilealliance.org/resources/experience-reports/harvesting-mob-programming-patterns-observing-how-we-work/
- https://github.com/michaelkeeling/mob-programming-patterns