Read by Refactoring

Mindshift 1: The Insight Loop

Facilitator's Guide

Contents

Advance Planning

Get permission to spend effort learning to Read by Refactoring.

- Development manager agrees to expect that programmers exhibit this skill.
- Product Owner allocates a story for the sprint.
- Team Lead agrees that this skill is worth learning this sprint.
- Scrum Master agrees to support learning this skill this sprint, rather than focusing on other team skills.
- Most skilled and respected developer agrees to show up to learn, and to help others learn together.

Schedule a ½ day mob (4 hours min).

Prepare the agenda.

- Mob Cycle #1: 2 hours
- Mob Cycle #2: 45 min
- Insight Loop Cycle: 20 minutes
- Mob Cycle #3: 30 minutes

Send a communication to developers that outlines:

- Empathy of time it takes to de-mystify code
- Intention of facilitating method that will reduce time for stories, reduce bugs, and make it less frustrating for them.
- Explain that this is a legitimate project and is supported by authority.
- **Request that they select the ugliest long method they can find, noting** the specific code doesn't matter.
- Attach the agenda.

Add a story to the board that represents the team mob.

Ensure the team brings a long, ugly method to work with.

Room Set-Up

- [ ] Prepare mobbing station based on visual below. The intention is to have one developer with hands on keys and anybody else may make suggestions. Rotation is important so everybody is in the code.

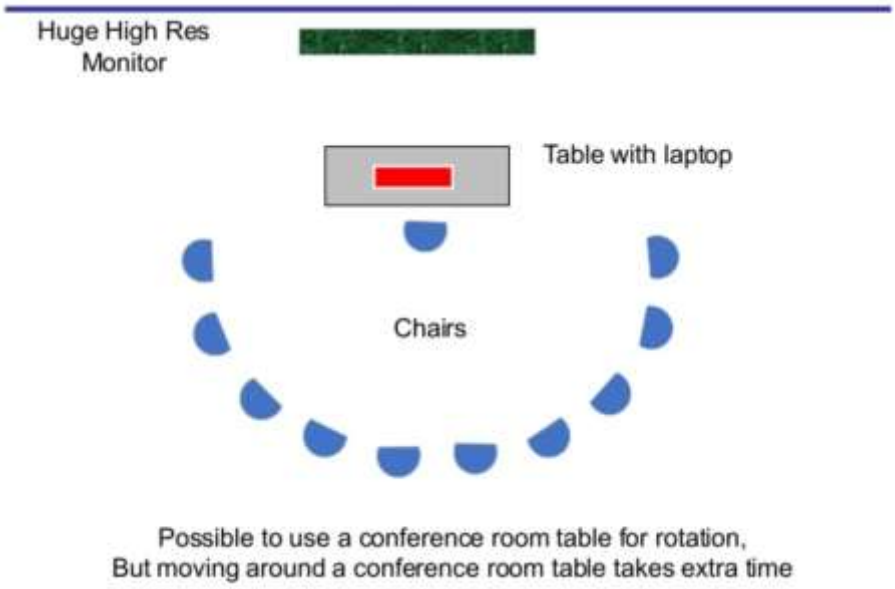

Figure 1: Image Credit /in/camillebell

- [ ] Have whiteboard accessibility or flipchart available.
- [ ] Have audio/visual prepared for sharing the expertise videos.
- [ ] Have the expertise videos prepared for easy access at the right times.
    - Video #1: Honest Names
    - Video #2: Committing – Safe
    - Video #3: Committing – Better
    - Video #4: Committing – No Worse
    - Video #5: Committing – Just Enough Process
    - Video #6: Committing – Deep Safety
    - Video #7: The Insight Loop

Facilitating Mob Cycle #1 – 2 hours

Phase 1: Facilitator Notes

The first phase is identifying code and pulling it out. The developers will naturally want to find exactly the right code and pull it out in exactly the right way. Don't let them **do that.**

The mindset for this mob cycle is that it doesn't matter if they get the wrong thing or extract it incorrectly. As they practice, extracting will become very fast, and when it's found that the wrong thing has been extracted, they can always undo it with inline method and extract the right thing.

Instructions

Habit Shift

One of These Things is not Like the Other Ones

- ☐ Verify that they brought painful code and ask them to bring it up.
- ☐ Ask them to zoom out to the point they can't read it.
- ☐ Now that they can't read it, ask them what parts are interesting.
- ☐ When they identify a part, have them zoom back in and extract it.
- ☐ Make a point that this is not safe; this is just getting them used to **finding things that are not like the others**, and we'll practice safety in the next cycle.
- ☐ Recommend that they make it a nonsense name (such as "foo") because the next step is to name it better.

Habit Shift

Names that Don't Suck

- ☐ Note that "Foo" is not very descriptive, so have them scan through and generally identify at least one thing it does.
- ☐ Have them name it:
  probably_[whatever that thing is]_AndStuff

Reaction

They will be very concerned about that name!

Technical Expertise: Naming

To alleviate their concerns, show them Video #1: Honest Names. Once they have watched it, have them name the method as you instructed earlier.

Success Check
We have a second method and it's named according to the pattern provided.

Phase 2: Facilitator Notes

The second phase is very simple: getting them to check-in. At this point, all the objections you'll experience from the developers are addressed as they will occur through the content below. Your job will be to facilitate through their concerns with curiousity, knowing that they will be categorized and addressed one by one.

Instructions

Ask them to check it in on to the main branch, such that will deploy to production. If they are willing, they probably feel safe because it will be just checked in on a branch. Clarify that the check in is to the **main**.

Reaction

They will undoubtedly refuse to check-in. Your next hour will be spent facilitating through all the very valid objections!

Facilitate

- ☐ Ask them to list all the reasons why we shouldn't check it in and write them down on the white board. They WILL fall within at least a couple of these categories.
  - ✓ TOO DANGEROUS No testing; may introduce bug
  - ✓ TOO CRAPPY Not worth it; name is bad; code isn't ready; embarrassing
  - ✓ SYSTEMATICALLY BAD Hurt performance; hurt debuggability
  - ✓ NO PROCESS No code review or process

- ☐ If any of these categories do *not* come up, be sure to add them.
- ☐ Tell them that we will address the categories one at a time.

Success Check
Everybody agrees that if those four criteria were met, then it would be safe to commit.

Phase 3: Facilitator Notes

The third phase addressing each of these four categories so that the developers can feel safe about checking in the method they extracted and named with the new naming convention.

As facilitator, know these categories match these solutions.

- ✓ Too Dangerous – Safe
- ✓ Too Crappy - Better
- ✓ Systematically Bad – No Worse
- ✓ No Process – mobbing

Addressing these categories will **later** allow you to introduce the mantra "safe, better, and no worse" that they can practice as a new standard to what is committable.

Technical Expertise: Safe

- ☐ Tell them that the **too dangerous** issue will be addressed first, by understanding safe.
- ☐ Show them Video #2: Committing – Safe.

Facilitate

- ☐ During the video, draw levels of proof on the white board (name and how to verify).
- ☐ After the video, note that yes, this first time we won't achieve top proof. We WILL next time. Meanwhile, let's make this as safe as possible to commit.
- ☐ Ask them to use higher levels to prove safe enough.

For this context, safe enough means that it's no more likely to cause a bug than anything else already checked in. As such, allow the developers to do anything they need to meet the definition of "safe enough". Suggest that they use techniques from the levels of proof that are higher than running tests.

Success Check
The developers agree that the objections that fell within the "too dangerous" cluster have been resolved.

Technical Expertise: Better

- Tell them that we will now address the **too crappy** issue, with a focus for making it better.
- Show them Video #3: Committing – Better.

Facilitate

- After the video, ask what the aspects were that made it better.
  Examples include security, performance, readability, and testability.
  *Your goal is to get developers to list at least **one**.*

- If they do not feel anything was made better, offer the following items.
  - ✓ Is the long method easier to read?
  - ✓ Is the new method easier to read and test?
  - ✓ What data flows in this new method and comes out of it?
    When they answer, ask how obvious that was before they extracted it.
  - ✓ Does the name represent **exactly what is known** about that method?

They will likely agree that it's better, but that performance is worse. Assure them that this will get addressed in the next category of making code no worse.

Success Check
The developers feel that the objection of "too crappy" has been resolved.

Technical Expertise: No Worse

- Tell them that we will now address the **systematically** bad issue, with a focus for making it no worse than it was.
- Show them Video #4: Committing – No Worse.

Facilitate

- After the video, ask what are all the aspects of code that matter.
- Write on whiteboard or flipchart each aspect they list.
  Examples will include security, performance, readability, and testability, and you can use these to prompt the start of that list.

- If they feel that things were made worse, **ask what the minimum change would be to make to the commit in order to not make it worse**.

Success Check
The developers agree that the objections that fell within the "systematically bad" cluster have been resolved.

Technical Expertise: No Process

- ☐ Tell them that we will now address the **no process** issue, based on the processes they have been using.
- ☐ Show them Video #5: Committing – Just Enough Process.

Facilitate

- ☐ Lead a discussion with the team on the following questions.
  - ✓ What piece of the process are no longer necessary because of the way we're working (mobbing and committing only one provable refactoring)?
  - ✓ Is there anything else we need to do?

- ☐ Have them perform the minimum necessary to address any remaining concerns.

This is the opportunity for the developers to realize how much they can bypass in their regular process when they are 1) working in mob and 2) checking in a single safe refactoring.

There are parts of the process they will likely feel are necessary, but you should challenge them to reconsider. These are code reviews, checking with designer or Product Owner, or running the full test suite again.

Success Check
The developers agree that the objections that fell within the "no process" cluster have been resolved.

Instructions

Ask them to check it in.

Success Check	Habit Shift
A commit was executed.	Tiny Commits

Facilitating Mob Cycle #2 – 45 min

Mob Cycle Introduction

This entire cycle is about safety. The first mob was about covering better and no worse, and we did safety as much as we could after the fact. Now we are going to focus on safety from the beginning.

Instructions

- Ask them to identify one thing to extract.
- Explain that now we want to do this with deep safety: bug-for-bug compatability.

Technical Expertise: Deep Safety

In preparation to extract with safety, show them Video #6: Committing – Deep Safety.

Safely Extracting: Facilitator Notes

Developers will be very happy now that deep safety is on the table! They will also become more pedantic, but so will you through these instructions. There are some actions for you to notice and adjust their behavior. These include:

- **Changing code by editing text.**
  Instead, have them copy and paste or use tool capabilities.
- **Combining or skipping steps.**
  Instead, ask them how they would prove that their transformation is bug-for-bug compatible. Since it's not possible, they will undo the combination step.
- **Reading, analyzing, or understanding the code in order to create safety.**
  Instead, remind them that we are creating a **process** for safety that will works without understanding the code.

Instructions

Habit Shift
Bug-for-Bug Compatibility

- ☐ Ask them if they will be using the tool approach or the recipe approach.

  RECIPE APPROACH

  If it's the recipe approach, open the recipe on the screen and do the following.
  - Have them fork it on Github and clone. If they don't have experience on Git, then fork on Github and edit on Github.
  - Going one step at a time through the recipe, have them do each action below for **every step**.
    - Remove, adjust, or change to fit their language
    - Commit any changes to the recipe
    - Ensure everybody in the MOB is comfortable with the step)

  TOOL APPROACH

  If it's the tool approach, have them talk about how they will discover and handle tool limitations.
  - Have them write the limitations down as a set of check steps and what kinds of code to check for.

- ☐ Extract to a method using the tool or recipe and name just like last time.

Success Check
Everybody agrees that this extraction is safe and doesn't require running tests.

Instructions

- □ Check if ready to commit. Is it:
  a. Safe?
  b. Better?
  c. No Worse?

- □ Ensure the commit is performed.

Success Check
A commit was executed.

Facilitating Insight Loop Awareness – 20 minutes

Facilitator Notes

Now that you have practiced a set of techniques twice, it's time to articulate how they all come together. While each action is a good independent habit shift, all four of them combined culminate into a bigger mind shift, called the Insight Loop.

Technical Expertise: Insight Loop

In preparation for the de-brief, show them Video #7: The Insight Loop.

Facilitate

- □ While the video is playing, draw the insight loop cycle as described in the video on the whiteboard or flipchart.
- □ Lead a discussion that ensures they realise these two things:
  - ✓ Each of the first 2 mobbing iterations was one cycle of the insight loop.
  - ✓ It is cognitively easier to work this way.
- □ Keep the insight loop drawn on the whiteboard through remaining rounds of mob cycles for this session.

Success Check
Developers exhibit an aha moment.

Mob Cycle #3 and Beyond! – 30 minutes

The Fast Commit Game: Facilitator Notes

This cycle is about fluency. The Insight Loop is the most effective when developers do many cycles that are small instead than trying to do a lot in a single cycle. During this session, developers will likely get down to 6-8 min per cycle. With practice over the next two weeks, they will get to 1-3 minutes each cycle.

Instructions

- Set it up as a game:
  How many iterations can we complete in the next 30 min?
- Extract another method but guide them only by helping them keep track of where they are in the insight loop (it's on the whiteboard).
- Have them repeat the cycle as many times as possible.

Success Check
The last iteration is less than 8 minutes long.

Glossary: Tech Terms

There are many terms that developers used that you won't know, and as long as you have a curious approach for asking what it means, it will be fine. However, there are some terms that are very commonly used and practically in the developer's subconscious that will also give you a great deal of context during your facilitator. Below are the common words, what they mean, and a metaphor to help you visualize how developers perceive the concept.

Technical Term	Meaning	Metaphor
Check-In or Commit	Save the file in a way that shares to some other developers.	Sending a document with track changes to another author.
Commit Message	A description of the intent behind the check-in (commit)	When your file directory looks like this: talk.agile2019draft.doc talk.withimages.doc talk.readyforreview.doc The bolded would be the commit messages.
Extract	Modularize a big chunk of code so that you can refer to the smaller pieces.	In a pie recipe where it says "make crust", the crust instructions were *extracted*.
Refactor	To reorganize code that does the same thing but reads more easily.	Precise translation to another language.
Branch	A place to check-in code, with a designated purpose and audience.	Each alternative version of a movie, i.e., director's cut, with commentary, etc.
Main	The branch that gets shipped to customers.	The movie that is released in theatres.
Source Control	A tool that holds onto all changes made to the code.	The history that is available in Dropbox.