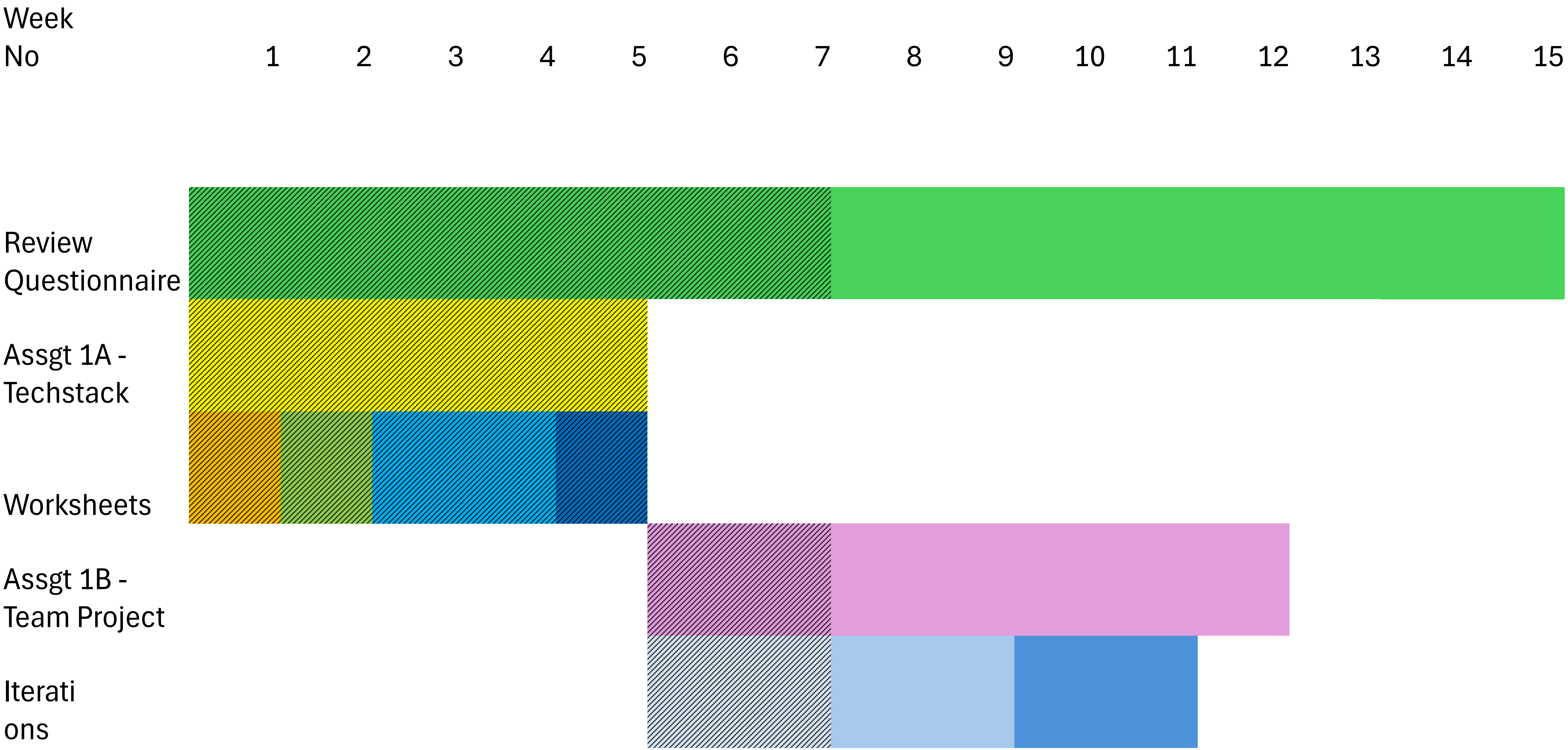
Review Techniques

Week 8

Taking Stock



Code Quality

What are the quality criteria to test the quality of code against

What are the code quality testing practices to run the tests against these criteria

What practices will help to ensure test fails do not happen?

Code intentionality is clear – good naming conventions
This means it is easy to understand (read) how it works, what it is supposed to do,
and easier to change, review, test, debug,

Code structure is high quality – code units are **loosely coupled and highly cohesive**
Object oriented – S.O.L.I.D principles

Do not repeat code – multiple places to change (**DRY Principle**)

Code Reviews will improve code quality
Test Driven Design will improve code quality
Test Automation will improve code quality

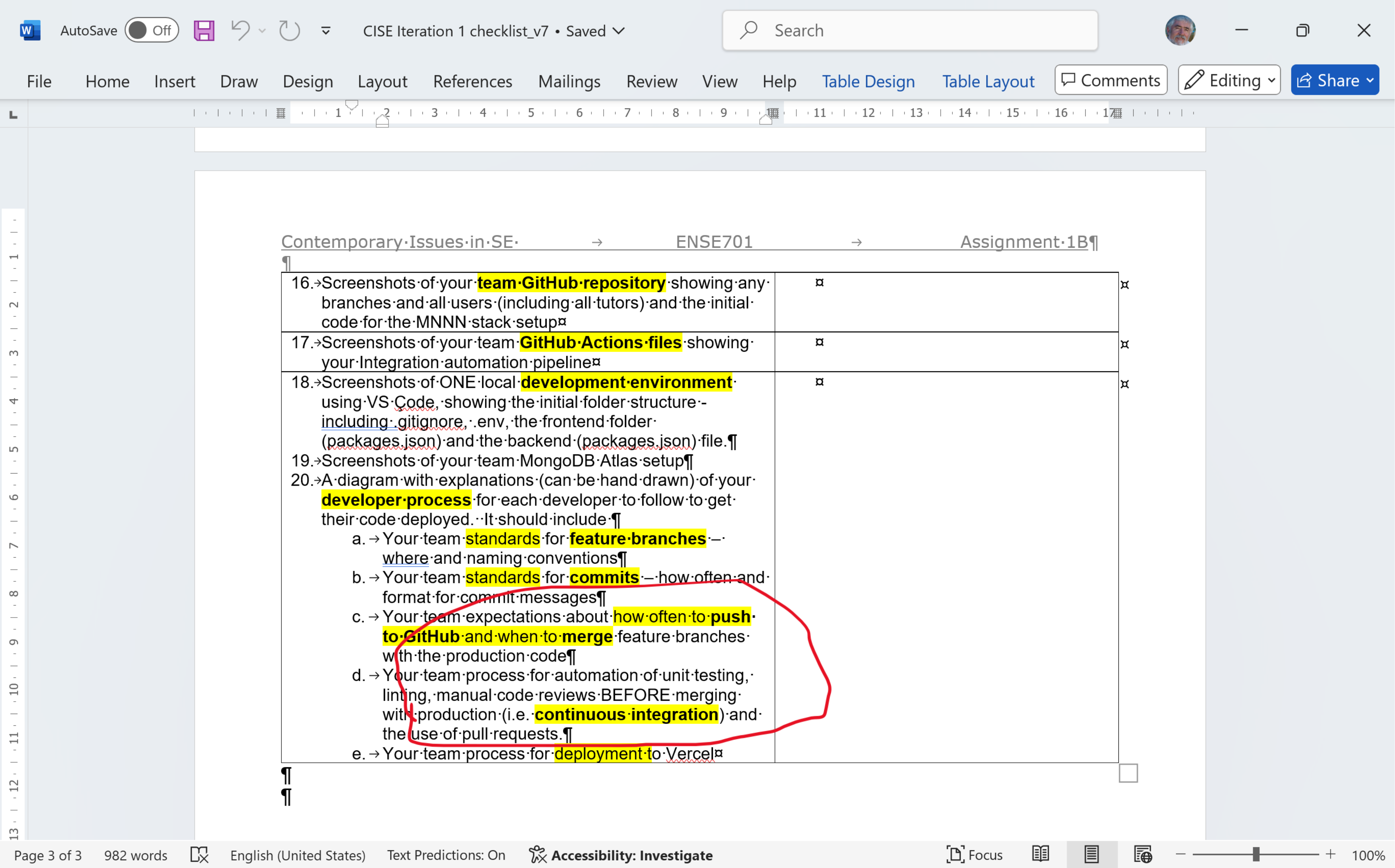
Code reviews ….So many benefits!

https://betterprogramming.pub/5-ways-code-reviews-helped-my-career-8d72aa1d2474

https://medium.com/inside-league/how-one-code-review-rule-turned-my-team-into-a-dream-team-fdb172799d11

DEV community analysis [8,15]

RQ3: Applying empathy in software engineering activities	Communication and Collaboration - practitioners consider empathy useful or important when communicating with colleagues, clients, and users.	software developers play different roles in their organizations…would involve talking to people and wherever you need to deal with people, empathy can play a key role
	Coding - practitioners discuss the need for empathy when they are coding or maintaining the code of other developers,	Something that I learned as the time passes was to have empathy with another developer's code."
	Management and Leadership - practitioners, view the need for empathy to successfully coordinate, communicate, motivate, and work with their teams and colleagues.	"To make an impact, our SRE leaders need to lead with empathy and help the rest of the organization engineer with empathy."
	Code review - practitioners consider empathy necessary in the code review process	Empathy for other engineers - … Be mindful that…asking for their input is essentially asking them for their time



16. Screenshots of your **team GitHub repository** showing any branches and all users (including all tutors) and the initial code for the MNNN stack setup	
17. Screenshots of your team **GitHub Actions files** showing your Integration automation pipeline	
18. Screenshots of ONE local **development environment** using VS Code, showing the initial folder structure - including .gitignore, .env, the frontend folder (packages.json) and the backend (packages.json) file. 19. Screenshots of your team MongoDB Atlas setup 20. A diagram with explanations (can be hand drawn) of your **developer process** for each developer to follow to get their code deployed. It should include a. Your team standards for **feature branches** – where and naming conventions b. Your team standards for **commits** – how often and format for commit messages c. Your team expectations about how often to **push to GitHub** and when to **merge** feature branches with the production code d. Your team process for automation of unit testing, linting, manual code reviews BEFORE merging with production (i.e. **continuous integration**) and the use of pull requests. e. Your team process for deployment to Vercel	

How to have high quality code reviews

https://medium.com/better-programming/how-to-review-code-in-7-steps-98298003b7ec

https://betterprogramming.pub/5-rules-for-every-code-review-98bf60dd5dbe

https://curtiseinsmann.medium.com/ive-code-reviewed-over-750-pull-requests-at-amazon-here-s-my-exact-thought-process-cec7c942a3a4

https://medium.com/swlh/3-problems-to-stop-looking-for-in-code-reviews-981bb169ba8b

Inspections

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 3*, 182-211.

https://www.proquest.com/openview/dd282e91ad39c894cc36b0ec56c23c7f/1?pq-origsite=gscholar&cbl=35072

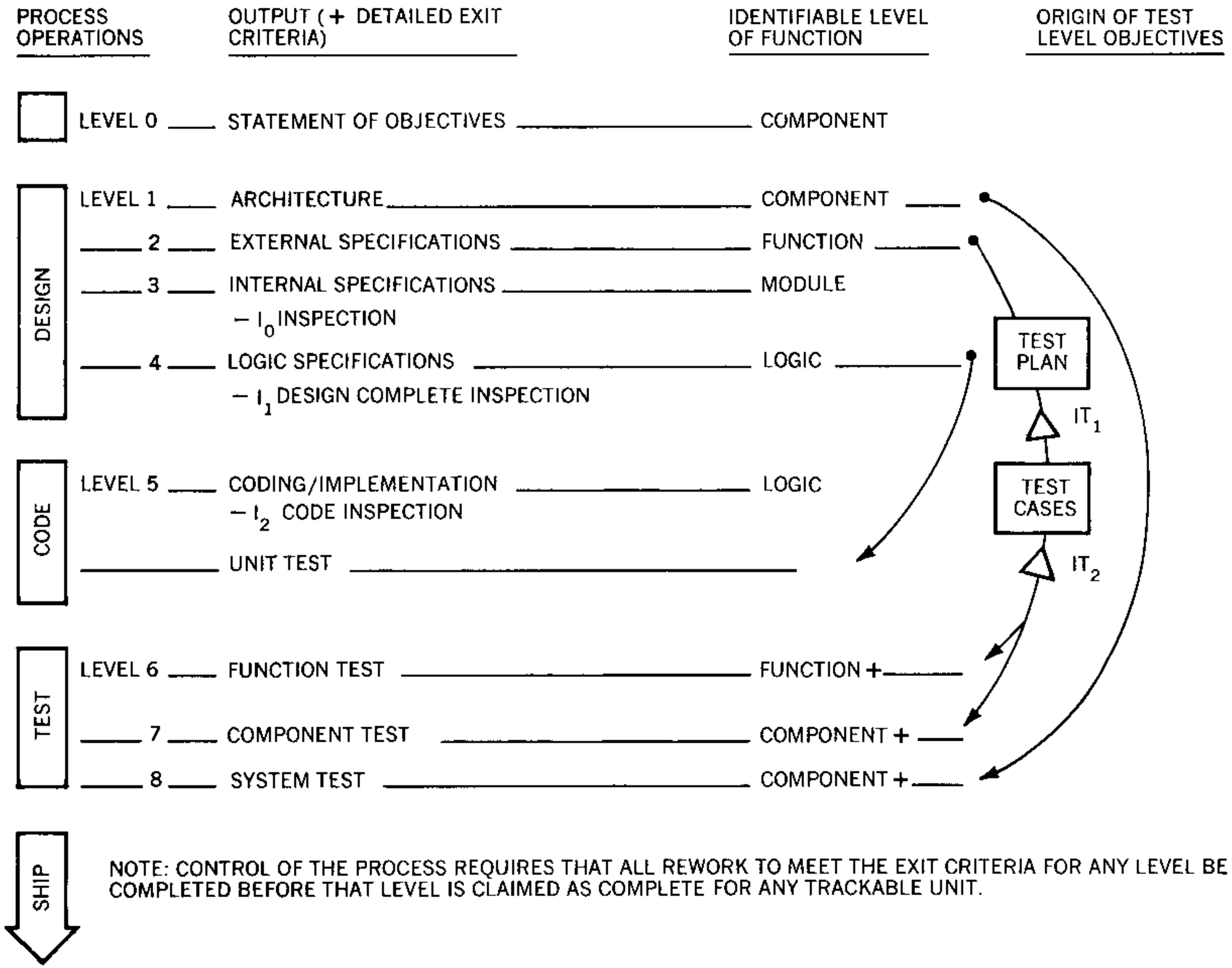
https://link.springer.com/chapter/10.1007/978-3-642-59412-0_35

Glass, R. (1999). Inspections - Some Surprising Findings. *Communications of the ACM, 42*(4), 17-19.

https://dl-acm-org.ezproxy.aut.ac.nz/doi/pdf/10.1145/299157.299161

Fagan Inspections

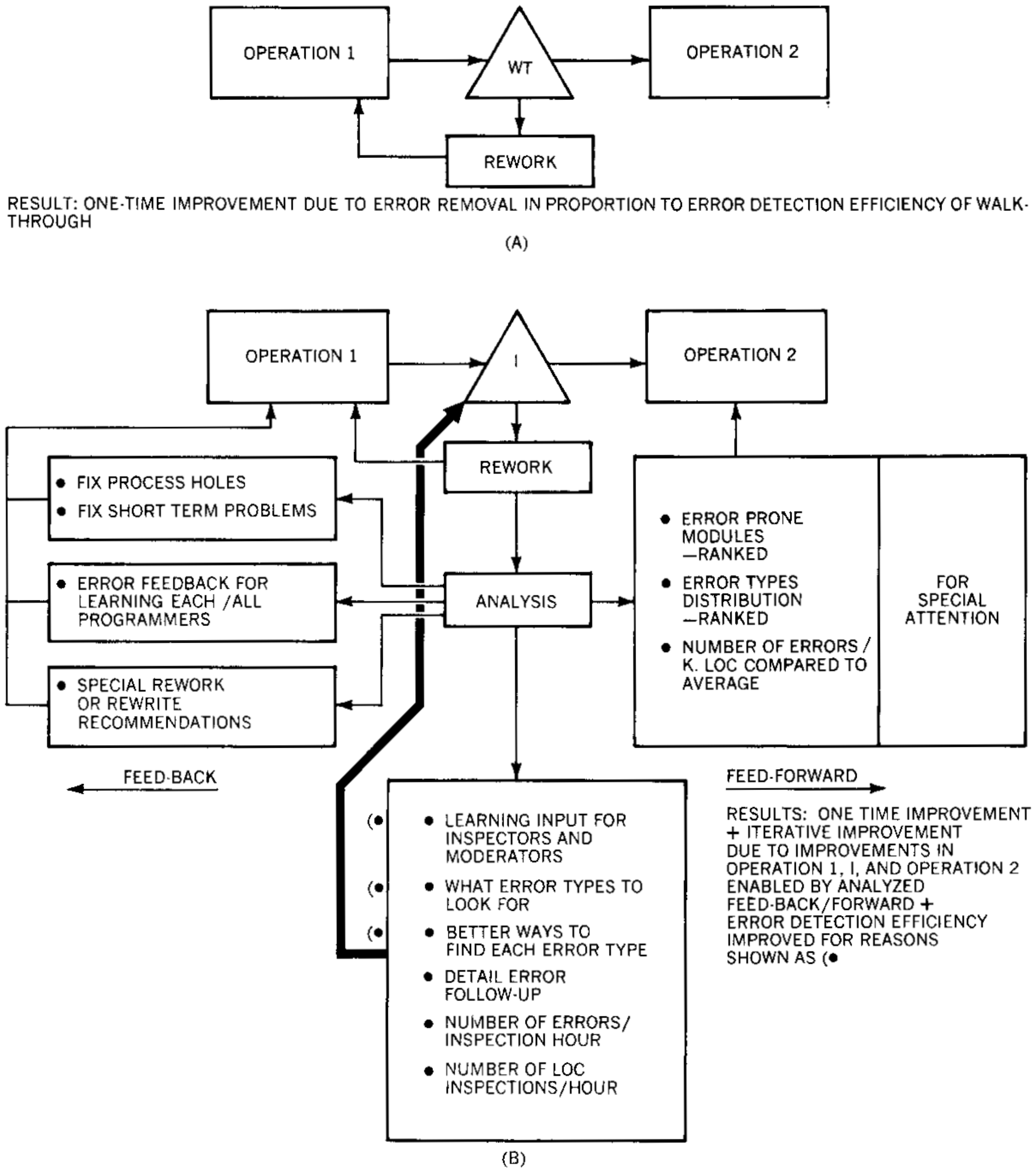
Figure 1 Programming process



NOTE: CONTROL OF THE PROCESS REQUIRES THAT ALL REWORK TO MEET THE EXIT CRITERIA FOR ANY LEVEL BE COMPLETED BEFORE THAT LEVEL IS CLAIMED AS COMPLETE FOR ANY TRACKABLE UNIT.

Fagan Inspections vs. Walkthroughs

Figure 10 (A) Walk-through process, (B) Inspection process



RESULT: ONE-TIME IMPROVEMENT DUE TO ERROR REMOVAL IN PROPORTION TO ERROR DETECTION EFFICIENCY OF WALK-THROUGH

(A)

RESULTS: ONE TIME IMPROVEMENT + ITERATIVE IMPROVEMENT DUE TO IMPROVEMENTS IN OPERATION 1, I, AND OPERATION 2 ENABLED BY ANALYZED FEED-BACK/FORWARD + ERROR DETECTION EFFICIENCY IMPROVED FOR REASONS SHOWN AS (●

(B)

OPERATION 1
WT
OPERATION 2
REWORK
OPERATION 1
I
OPERATION 2
REWORK
ANALYSIS
● FIX PROCESS HOLES
● FIX SHORT TERM PROBLEMS
● ERROR FEEDBACK FOR LEARNING EACH /ALL PROGRAMMERS
● SPECIAL REWORK OR REWRITE RECOMMENDATIONS
FEED-BACK
● ERROR PRONE MODULES —RANKED
● ERROR TYPES DISTRIBUTION —RANKED
● NUMBER OF ERRORS / K. LOC COMPARED TO AVERAGE
FOR SPECIAL ATTENTION
FEED-FORWARD
(● ● LEARNING INPUT FOR INSPECTORS AND MODERATORS
(● ● WHAT ERROR TYPES TO LOOK FOR
(● ● BETTER WAYS TO FIND EACH ERROR TYPE
● DETAIL ERROR FOLLOW-UP
● NUMBER OF ERRORS/ INSPECTION HOUR
● NUMBER OF LOC INSPECTIONS/HOUR

Why it is important to have well-crafted clean code?

Quality software is developed in teams

Other people will need to read and understand how your code works to extend it, debug it, change it or remove it.

You may need to do the same a day later, two weeks later, 6 months later

THINK ABOUT WHO WILL COME NEXT!
BE A GOOD TEAM MATE!

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. " — Martin Golding

So how can I craft my code so it is easier for me and others to understand how it works?

Pull Requests and Documentation

https://betterprogramming.pub/why-every-git-commit-message-must-include-its-commit-context-1171c0b2f710

https://dev.to/helderburato/patterns-for-writing-better-git-commit-messages-4ba0

Integration of Code - workflow

What are the steps during a day for a developer working in a team of developers with a shared code base in GitHub to work on code integrate their code

What does “Continuous” integration mean?

Pull latest version of working branch to local repo

Work on it writing test and functional code, running tests

Commit frequently with informative commit messages to local repo

Push code to working branch in GitHub and merge after checking all tests pass locally

Run integration tests on GitHub for merged code

Pull request merge with Develop or Main branches

Collaborator reviews, discusses, runs integration tests, if passes – merge to Develop/Master

Reflective Practice and Reviews

The role of reflective practice in increasing your professional effectiveness – putting reviews in context

https://youtu.be/M9hyWVEG2x0?si=VAAT65bmY5rPzCjB

Forms of action and reflection – not just doing, but thinking on what and how you are doing

https://www.youtube.com/watch?v=x2MfNE91jLk

Schön, D. (1987). *Educating the Reflective Practitioner*. Jossey Bass.

Automating Continuous Integration (and Deployment) Embedding Review

Based on some **trigger** (e.g. Pull request to merge into Main or Develop)

Take some steps **automatically** to check the code will integrate with the code to be merged into

Build the code (compile?)

Check the code meets some rules (linter)

Run all the unit tests for the entire code branch as if it is merged

Do the Merge

CD
If the merge is to the Main – deploy – release it- to the users

CI Servers

They will follow watch for the trigger specified

Follow the commands to run automatically

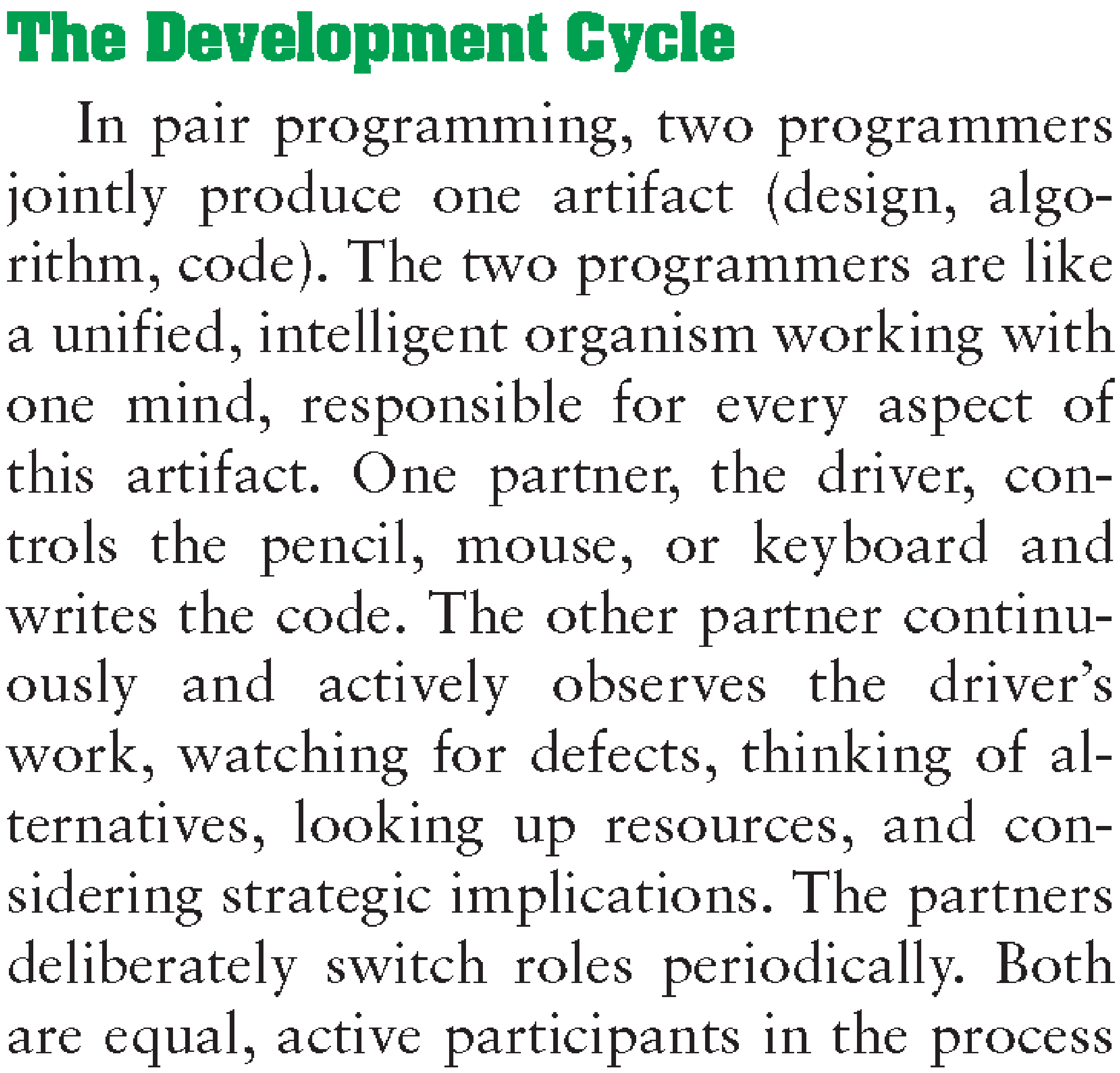
Usually stored in a YAML file

Github actions workflow

https://github.com/actions/starter-workflows/blob/main/automation/manual.yml

4 ways we use GitHub Actions to build GitHub - The GitHub Blog
https://images.app.goo.gl/QVxMe427dviFDncX7

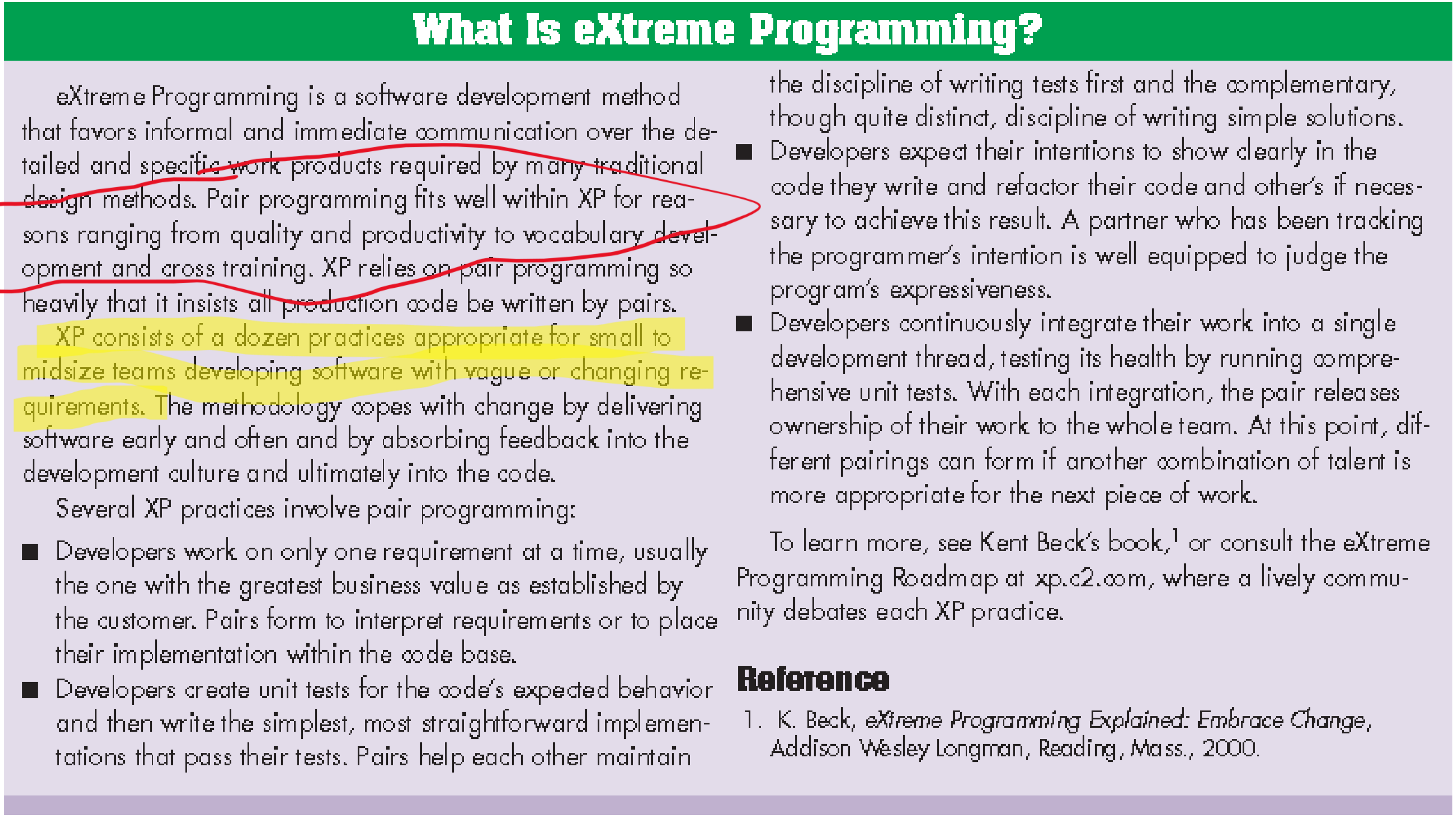
Collaborative Programming Practices



The Development Cycle

In pair programming, two programmers jointly produce one artifact (design, algorithm, code). The two programmers are like a unified, intelligent organism working with one mind, responsible for every aspect of this artifact. One partner, the driver, controls the pencil, mouse, or keyboard and writes the code. The other partner continuously and actively observes the driver's work, watching for defects, thinking of alternatives, looking up resources, and considering strategic implications. The partners deliberately switch roles periodically. Both are equal, active participants in the process

L. Williams, R. R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the case for pair programming," in *IEEE Software*, vol. 17, no. 4, pp. 19-25, July-Aug. 2000, doi: 10.1109/52.854064



What Is eXtreme Programming?

eXtreme Programming is a software development method that favors informal and immediate communication over the detailed and specific work products required by many traditional design methods. Pair programming fits well within XP for reasons ranging from quality and productivity to vocabulary development and cross training. XP relies on pair programming so heavily that it insists all production code be written by pairs.

XP consists of a dozen practices appropriate for small to midsize teams developing software with vague or changing requirements. The methodology copes with change by delivering software early and often and by absorbing feedback into the development culture and ultimately into the code.

Several XP practices involve pair programming:

- Developers work on only one requirement at a time, usually the one with the greatest business value as established by the customer. Pairs form to interpret requirements or to place their implementation within the code base.
- Developers create unit tests for the code's expected behavior and then write the simplest, most straightforward implementations that pass their tests. Pairs help each other maintain the discipline of writing tests first and the complementary, though quite distinct, discipline of writing simple solutions.
- Developers expect their intentions to show clearly in the code they write and refactor their code and other's if necessary to achieve this result. A partner who has been tracking the programmer's intention is well equipped to judge the program's expressiveness.
- Developers continuously integrate their work into a single development thread, testing its health by running comprehensive unit tests. With each integration, the pair releases ownership of their work to the whole team. At this point, different pairings can form if another combination of talent is more appropriate for the next piece of work.

To learn more, see Kent Beck's book,[1] or consult the eXtreme Programming Roadmap at xp.c2.com, where a lively community debates each XP practice.

Reference

1. K. Beck, *eXtreme Programming Explained: Embrace Change*, Addison Wesley Longman, Reading, Mass., 2000.

L. Williams, R. R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the case for pair programming," in *IEEE Software*, vol. 17, no. 4, pp. 19-25, July-Aug. 2000, doi: 10.1109/52.854064

Mob Programming

First coined in the Extreme Programming (XP) community in **2003** by Moses Hohman to describe their practice of code refactoring in a group of more than two.

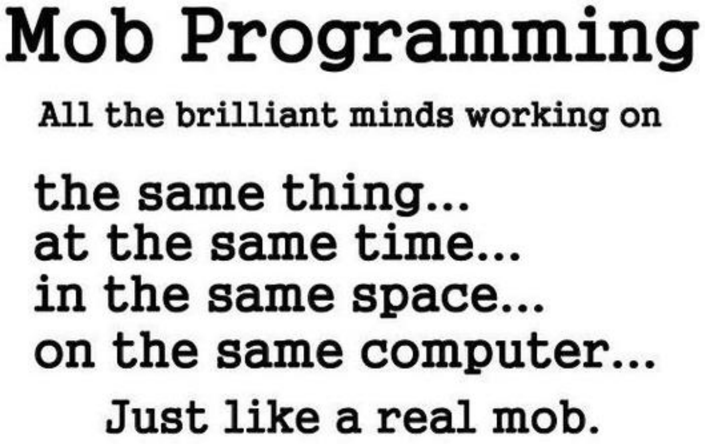
The team took "Mob Programming" to the next level.

Mobbing just extends the benefits of pair programming with more people working on a coding problem?

Agile leadership in mob programm

Mob Programming

Woody Zuill began popularizing it again from **2013**

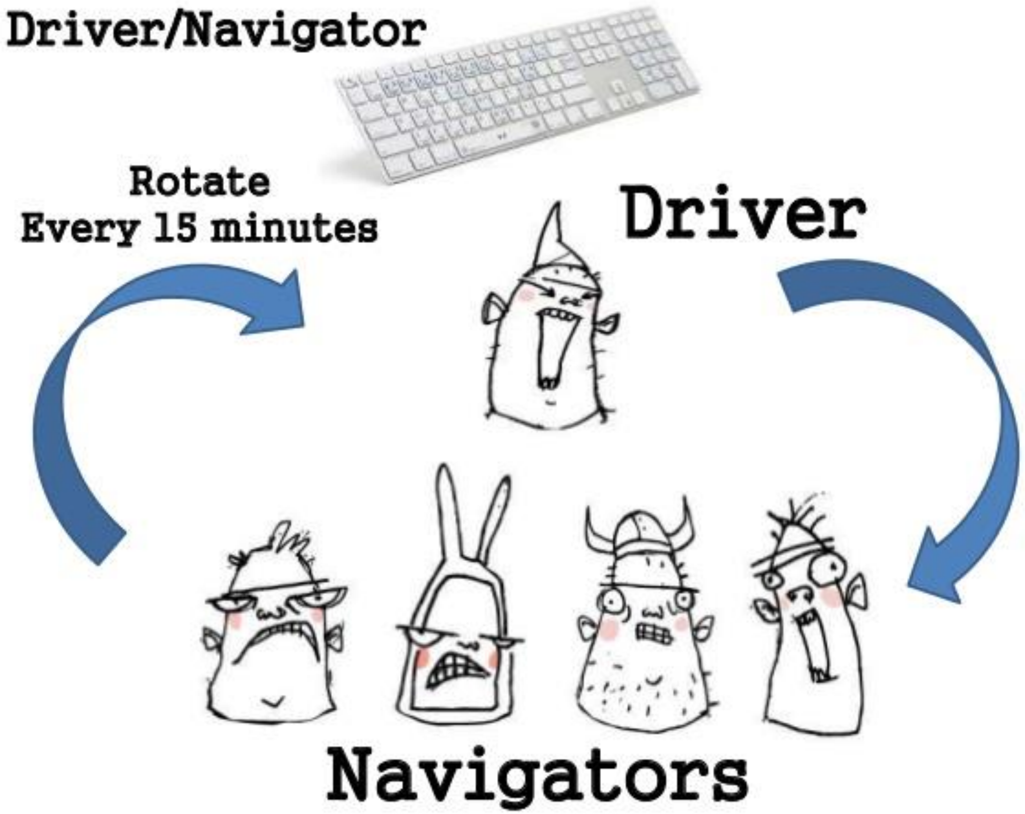


Mob Programming

All the brilliant minds working on

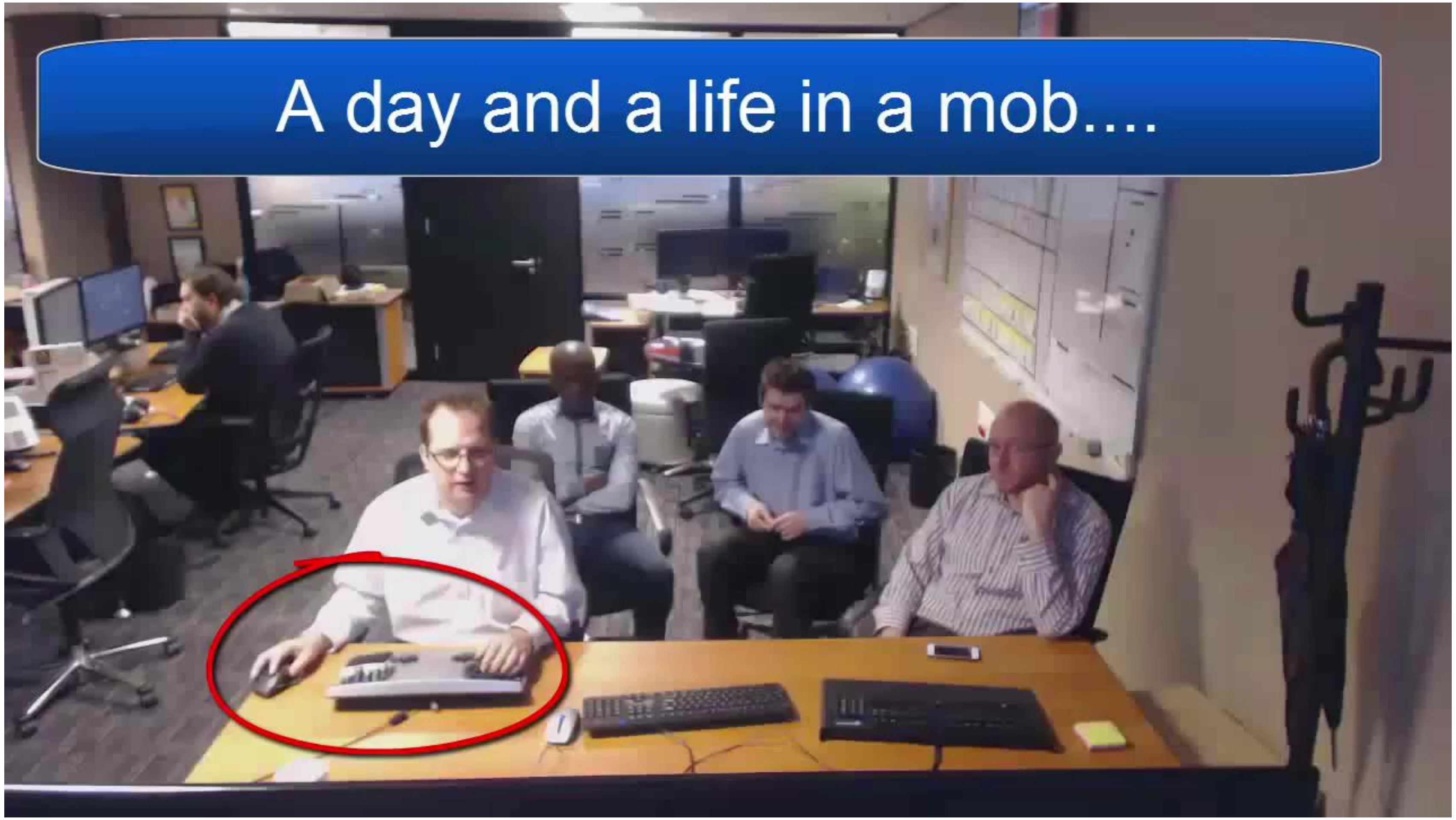
the same thing...
at the same time...
in the same space...
on the same computer...

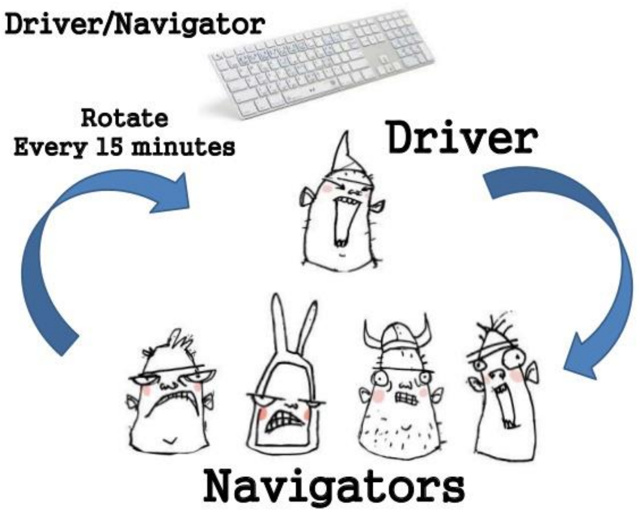
Just like a real mob.



Wider adoption in recent years– many benefits claimed……

Mob Programming





Seemed to have lots of side benefits and became the usual way of working for some teams

The Observed Benefits

Team code ownership emerged naturally

Individual have broader knowledge of the code base and front-end/back-end

- work shared more easily
- design decisions better informed
- critical knowledge loss by absence of individual less likely–

Increased confidence in quality of code

- improved team morale

The Observed Benefits

Consistent use of tools used

Onboarding new team member quicker

Higher confidence in the predictability of work effort

Increased productivity longer term

Reduction in multi-tasking

A higher level of code craft

Reduction of work in progress

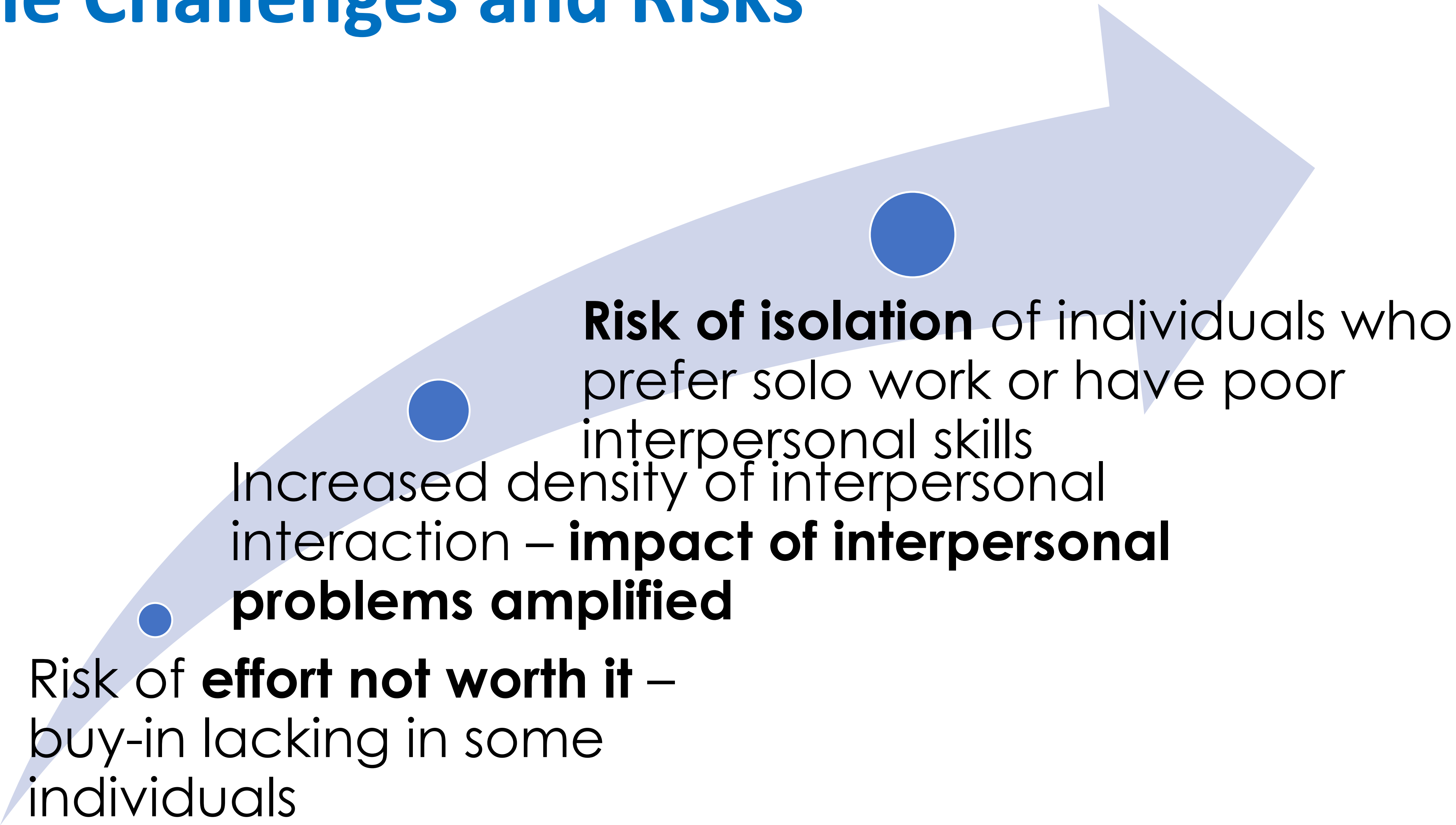
Increased productivity longer term

Less technical debt

Fewer interruptions

Fewer delays because of unavailable information

The Challenges and Risks



Risk of isolation of individuals who prefer solo work or have poor interpersonal skills

Increased density of interpersonal interaction – **impact of interpersonal problems amplified**

Risk of **effort not worth it** – buy-in lacking in some individuals

The Challenges and Risks

Suitable Workplace

Finding a suitable consistent work place with a a dedicated machine was a challenge

Role of tester in mob

Understanding and stabilizing the role of the QA in mobbing was challenging – QA morale low initially

Slower pace of coding

Pace of code generation slowed – can interrupt mobbing if time pressure dominates short term

Retrospectives

The high-level purpose is to keep learning as a team by reflecting on the last sprint

What can we learn from this that suggests something new to try for the next sprint

Need a process! There are many – find what suits the team

Happiness histogram
Sailboat
Answer questions

Need team members to

All participate

Be honest/candid

FOCUS ON MAKING THE TEAM STRONGER

http://scrummastertoolbox.libsyn.com/bonus-the-top-3-challenges-to-better-retrospectives-with-david-horowitz

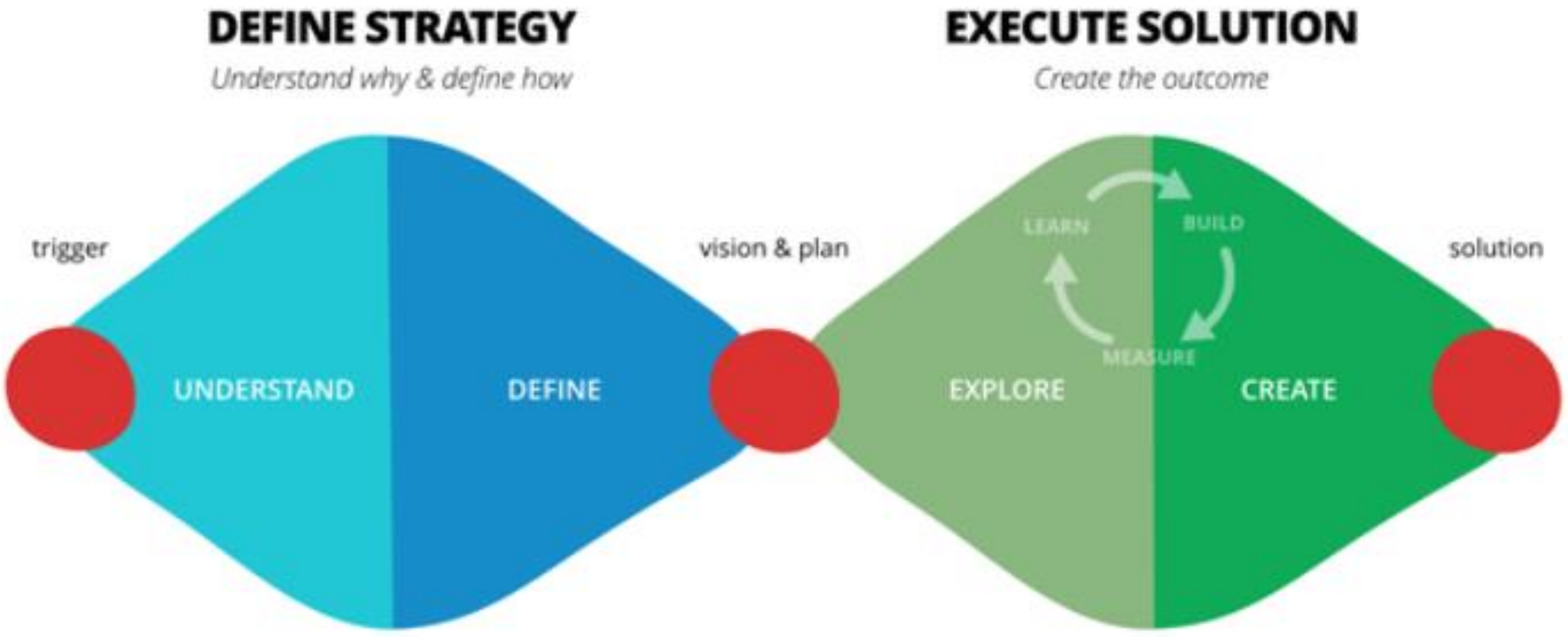
http://scrummastertoolbox.libsyn.com/how-to-find-what-agile-retrospective-format-works-for-your-team-justin-chapman

https://www.ponolabs.com/labs/wp-content/uploads/2019/03/Team_Canvas_v5.pdf

Retrospectives

Double diamond decision making

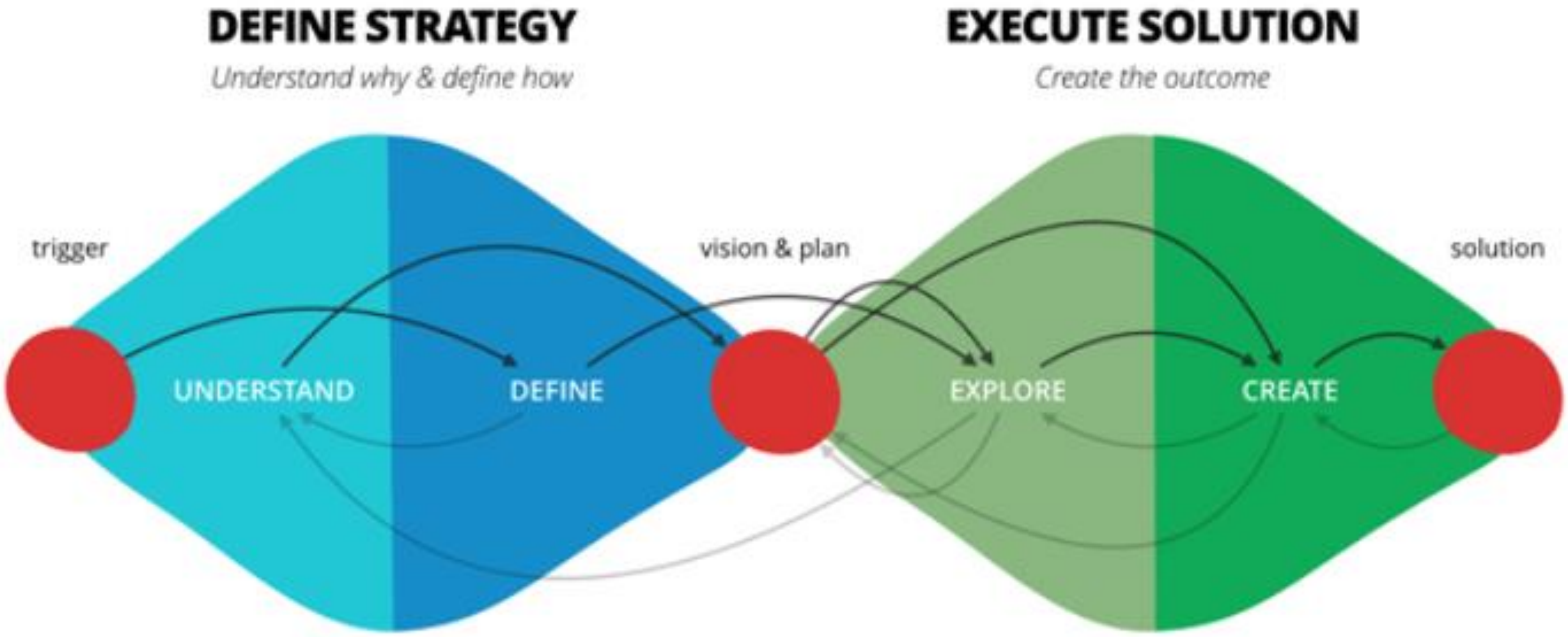
https://www.thoughtworks.com/insights/blog/double-diamond#:~:text=The%20'Double%20Diamond'%20process%20maps,of%20thinking%20that%20designers%20use.&text=It%20describes%20significant%20up%2Dfront,to%20produce%20a%20final%20solution



Double Diamond Thinking

Create

As we gain confidence in the solution, exploration gives way to engineering. Now we're creating and optimising working software. The opportunity here is two-fold. First, a working solution delivered to market. Second, we gather real market feedback. As a result, our understanding deepens, and new discoveries influence an ever-changing strategy. Software engineering is not merely execution of a plan, it also defines strategy.



Retrospectives

Divergent thinking

Groan Zone

Convergent thinking

What can we learn (impediment)

Some high value learning

ICE analysis (Impactful Confidence Effort)

Rank based on this – T-shirt sizes

Lean Coffee
(dot vote)

What are some options to try

One or more things to change for the next sprint

https://miro.com/templates/mad-sad-glad-retrospective/

Retrospectives

Top Issue
To get the team to converge on practical and impactful action items from the retro

Practical actions that team will take ownership for and implement in the next sprint and will make a difference

A retro that leads to no change is worse than a waste of time
No improvement and no value to the retro meeting

The retro needs follow through -> people get engaged->solves problems!

https://www.mountaingoatsoftware.com/blog/a-simple-way-to-run-a-sprint-retrospective

Tools to help online Retro

padlet

retrotool.io

https://retrotool.io/

retrium

https://www.infoq.com/articles/remote-retrospective-engage/?utm_source=notification_email&utm_campaign=notifications&utm_medium=link&utm_content=content_in_followed_topic&utm_term=daily

Software Process Improvement

Fundamental differences between traditional and agile software development regarding SPI[5].

First,

while SPI in the plan driven perspective **prescribes norms f**or how the individual, team and organization should operate,

agile software development address the improvement and management of software development **practices within individual teams** [2].

In agile development, **processes are not products,**

but rather practices that evolve dynamically with the team as it adapts to the particular circumstances [21].

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

Software Process Improvement (2)

Second,

plan-driven methods, such as the waterfall model, usually adopt a top-down approach for improving the software development process [5],

while the agile view has a **bottom-up approach**.

Third, SPI in plan-driven development often emphasizes the continuous improvement of the organizational software process for future projects,

while the principles of agile software development focus on **iterative adaption and improvement in the on-going projects**.

Short development cycles provide continuous and rapid loops to iterative learning, to enhance the process and to pilot the improvement.

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

Software Process Improvement (3)

When doing agile development, there are typically two meetings where the team focuses on improving the process.

1) Daily meetings. In the daily meeting the team members are supposed to coordinate their work and focuses on solving problems that stop the team from working effectively.

   In Scrum, the Scrum-master is supposed to facilitate this meeting and making sure impediments to the process are removed

1) Retrospective [22]. At the end of each iteration, a retrospective is held. In this meeting the team focuses on what was working well and what needs to be improved. Measures are then taken.

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

SPPI – Diagnosis & Planning

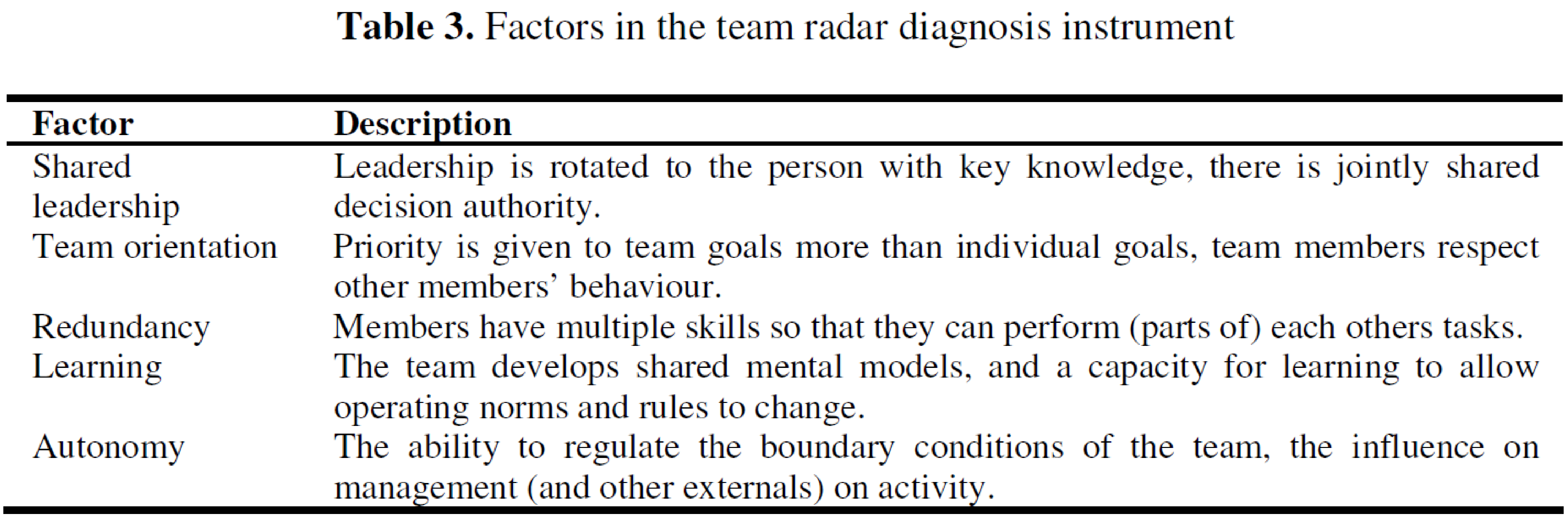


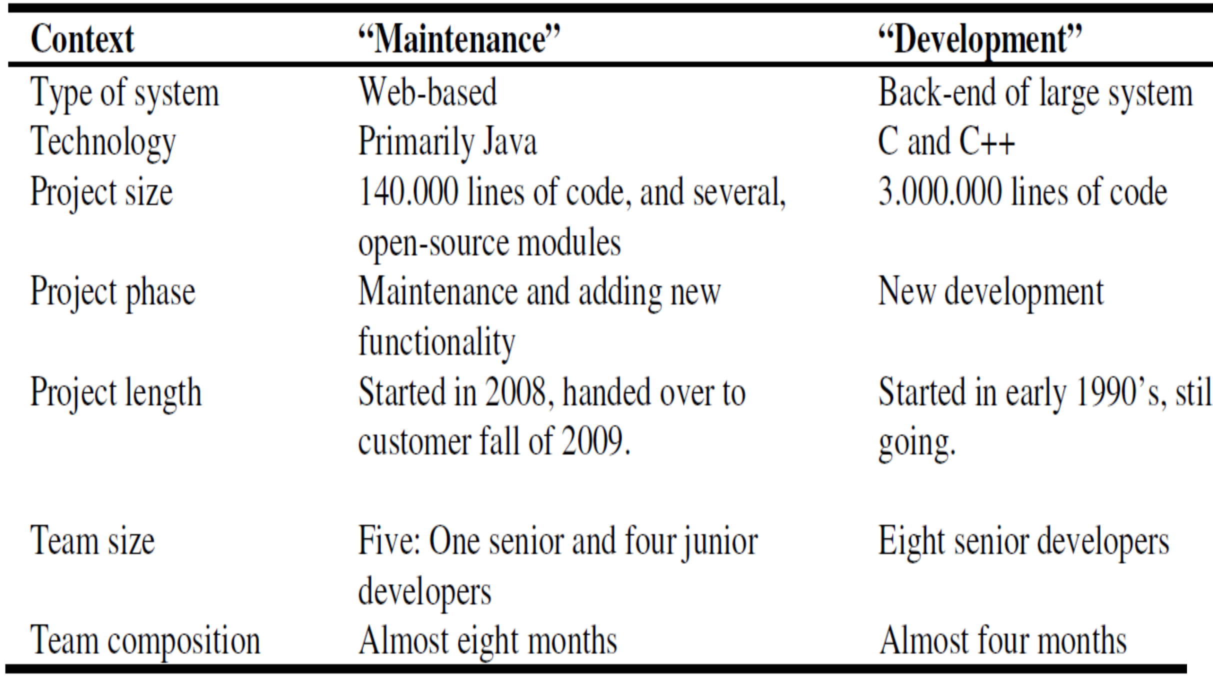
Table 3. Factors in the team radar diagnosis instrument

Factor	Description
Shared leadership	Leadership is rotated to the person with key knowledge, there is jointly shared decision authority.
Team orientation	Priority is given to team goals more than individual goals, team members respect other members' behaviour.
Redundancy	Members have multiple skills so that they can perform (parts of) each others tasks.
Learning	The team develops shared mental models, and a capacity for learning to allow operating norms and rules to change.
Autonomy	The ability to regulate the boundary conditions of the team, the influence on management (and other externals) on activity.

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

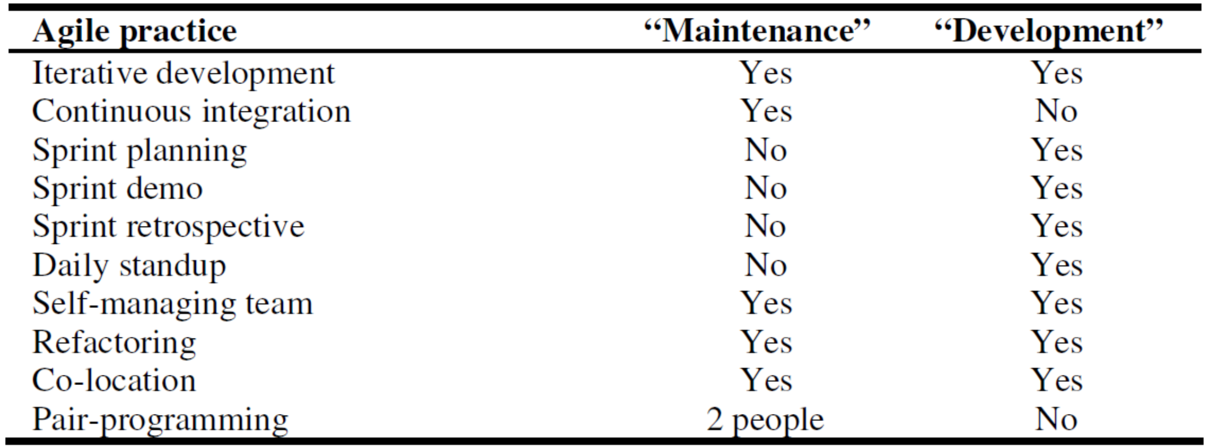
SPPI – Study Context

Table 1. Properties of the maintenance and development team



Context	"Maintenance"	"Development"
Type of system	Web-based	Back-end of large system
Technology	Primarily Java	C and C++
Project size	140.000 lines of code, and several, open-source modules	3.000.000 lines of code
Project phase	Maintenance and adding new functionality	New development
Project length	Started in 2008, handed over to customer fall of 2009.	Started in early 1990's, still on-going.
Team size	Five: One senior and four junior developers	Eight senior developers
Team composition	Almost eight months	Almost four months

Table 2. Agile practices in the two teams



Agile practice	"Maintenance"	"Development"
Iterative development	Yes	Yes
Continuous integration	Yes	No
Sprint planning	No	Yes
Sprint demo	No	Yes
Sprint retrospective	No	Yes
Daily standup	No	Yes
Self-managing team	Yes	Yes
Refactoring	Yes	Yes
Co-location	Yes	Yes
Pair-programming	2 people	No

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

SPPI – Diagnosis & Planning

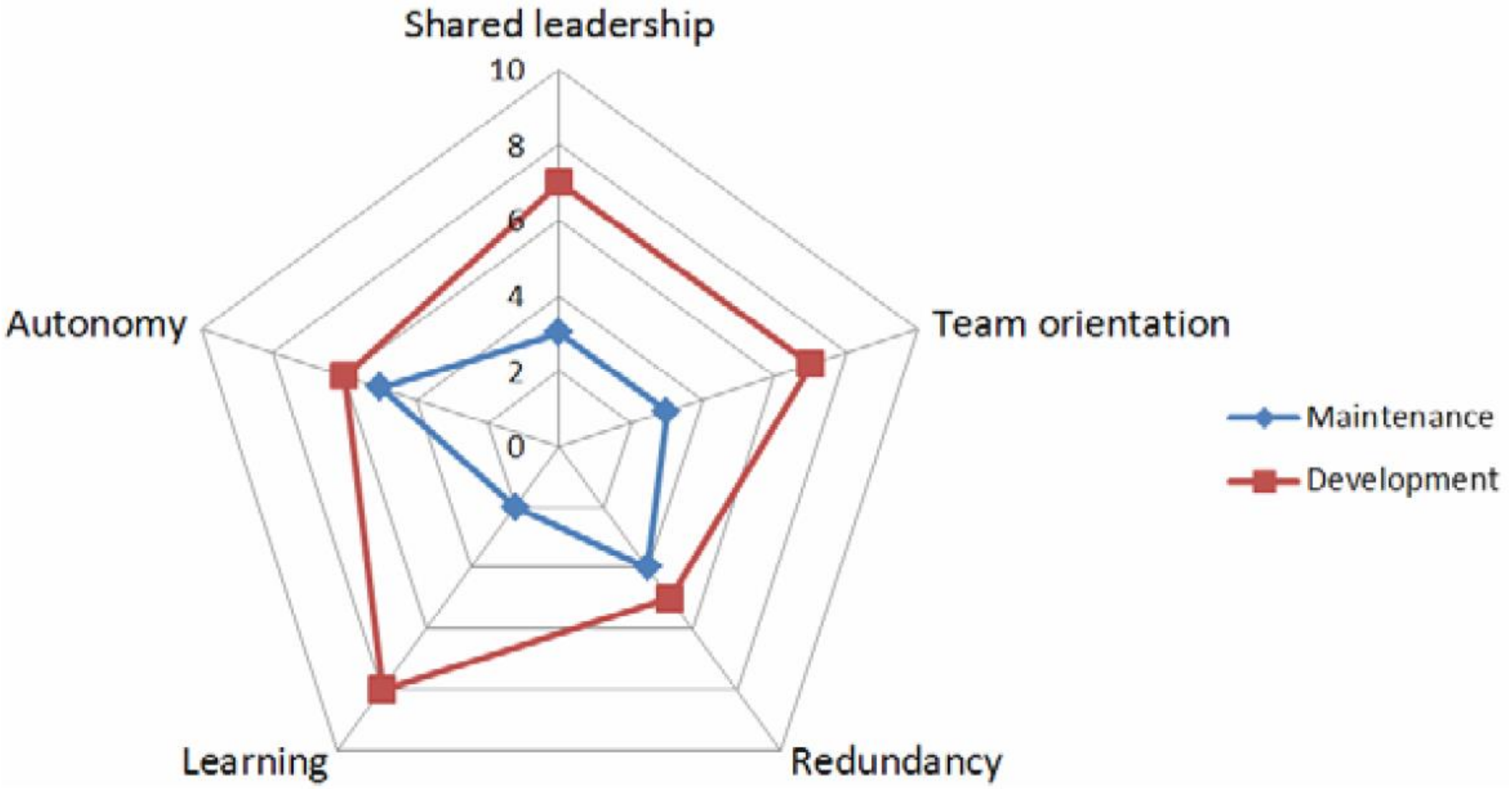


Fig. 1. A plot of teamwork characteristics of the two teams

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

SPPI –Planning: Mtce Team

To improve teamwork in the two teams,

- presented results of diagnosis phase,
- discussed priority on teamwork factors together with the teams.

As a result concrete measures to improve development processes and teamwork were suggested.

For the maintenance team we observed challenges related to *shared leadership, team orientation, and learning*.

- As for leadership, team dominated by junior developers, little involvement of the team in leadership and little process in place.
- Team heavily specialized, - team members working on independent modules, again lowered team orientation.
- Finally, team had no arenas for learning except for being in the same room, but observation showed little discussion and feedback on the actual work tasks the team members were involved in.

SPPI –Planning: Mtce Team

In a workshop, we presented the scores, problems and consequences to the team.

Team decided to reintroduce important agile practices they had stopped doing. In prioritized order:

□ Sprint retrospective to improve learning. Team members would be able to give feedback and improve both the development process as well as the product.

□ Daily stand-up meetings to improve coordination of tasks, team communicating, and solve problems daily. The meeting was expected to have an effect on shared leadership, team orientation and learning.

□ Code review to improve software quality, learning and increase redundancy.

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

SPPI –Planning: Dev Team

The development team got higher scores on all factors compared to the maintenance team. The team prioritized to improve the problems with the highest potential for the team: *inefficient sprint planning, variable ownership to project goals, and not solving process related problems in the retrospective.* The following actions were suggested:

□ Open space sprint planning, to conduct sprint planning more efficiently. The sprint planning meetings in the team were dominated by specialists and long lasting. Using the open space process, the team members would suggest topics to discuss and then several discussions could happen in parallel in the same room. Team members are encouraged to walk between discussions. This action was expected to improve shared leadership and team orientation.

□ Pair programming to improve team orientation. Making people to work closely together constantly giving feedback could also improve shared decision-making and improve learning.

□ Collocating the team in the same room, would improve communication and oversight, and improving team orientation.

SPPI in Agile – A Technique for Diagnosis & Planning?

Now we return to our research question, *"how to efficiently improve teamwork in agile software development?"*

We have shown results from using diagnosis with the team radar and action planning in a small and immature team and in a large and more mature team.

Both the teams perceived the diagnosing and the outcome as something they learned from, because it illuminated issues they had seen individually but not discussed within the team.

It is not enough to do retrospectives if the team is not able to discuss the cause of the problems they are experiencing.

SPPI in Agile – A Technique for Diagnosis & Planning?

This study indicates that

process improvement, *although a central concept in agile development* is still hard to achieve.

The main implication for practice is that

this study with two teams reveals that *process improvement does not happen by itself even in agile methods*, there needs to be effort invested to actively experiment with solutions.

Ringstad, M. A., Dingsøyr, T., & Moe, N. B. (2011). Agile process improvement: diagnosis and planning to improve teamwork. In *European Conference on Software Process Improvement* (pp. 167-178): Springer.

Questions and Comments….