Best Practices for Peer Code Review

1.- Review fewer than 200-400 lines of code at a time

At this rate, with the review spread over no more than 60-90 minutes, you should get a 70-90% yield; in other words, if 10 defects existed, you would find 7-9 of them.

Defect density is a measure of “review effectiveness.” If two reviewers review the same code and one finds more bugs, we would consider her more effective.

The World’s Largest Code Review Study at Cisco Systems

Using lightweight code review techniques, developers can review code in 1/5th the time needed for full “formal” code reviews.

To test our conclusions about code review in general and lightweight review in particular, we conducted the world’s largest-ever published study on code review, encompassing 2500 code reviews, 50 programmers, and 3.2 million lines of code at Cisco Systems®.

At the start of the study, we set up some rules for the group:

- All code had to be reviewed before it was checked into the team’s Perforce version control software.
- Smart Bear’s Code Collaborator code review software tool would be used to expedite, organize, and facilitate all code review.
- In-person meetings for code review were not allowed.
- The review process would be enforced by tools.
- Metrics would be automatically collected by Code Collaborator, which provides review-level and summary-level reporting.

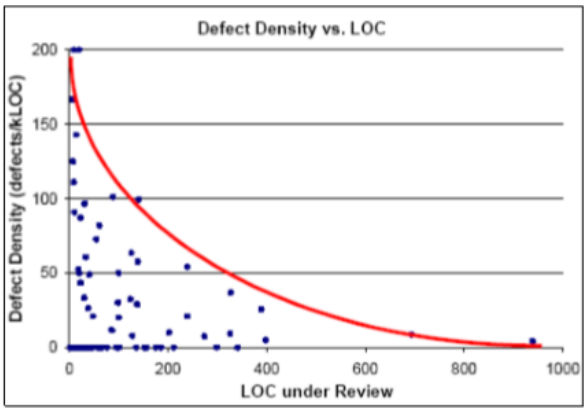

Figure 1: Defect density dramatically decreases when the number of lines of inspection goes above 200, and is almost zero after 400.

After ten months of monitoring, the study crystallized our theory: done properly, lightweight code reviews are just as effective as formal ones – but are substantially faster (and less annoying) to conduct! Our lightweight reviews took an average of 6.5 hours less time to conduct than formal reviews, but found just as many bugs.

2.- Aim for an inspection rate of less than 300-500 LOC/hour.

Take your time with code review. Faster is not better. Reviewing faster than 400-500 LOC/hour results in a severe drop-off in effectiveness. And at rates above 1000 LOC/hour, you can probably conclude that the reviewer isn't actually looking at the code at all.

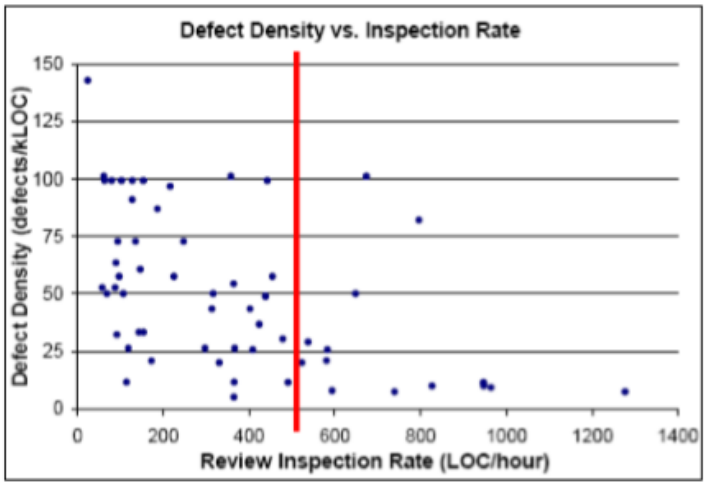

Figure 2: Inspection effectiveness falls off when greater than 500 lines of code are under review.

Important Definitions

Inspection Rate: How fast are we able to review code? Normally measured in kLOC (thousand Lines Of Code) per man-hour.

Defect Rate: How fast are we able to find defects? Normally measured in number of defects found per man-hour.

Defect Density: How many defects do we find in a given amount of code (not how many there are)? Normally measured in number of defects found per kLOC.

3.- Take enough time for a proper, slow review, but not more than 60-90 minutes.

You should never review code for more than 90 minutes at a stretch.

Given these human limitations, a reviewer will probably not be able to review more than 300-600 lines of code before his performance drops.

4.- Authors should annotate source code before the review begins.

Our theory was that because the author has to re-think and explain the changes during the annotation process, the author will himself uncover many of the defects before the review even begins, thus making the review itself more efficient. As such, the review process should y a lower defect density, since fewer bugs remain.

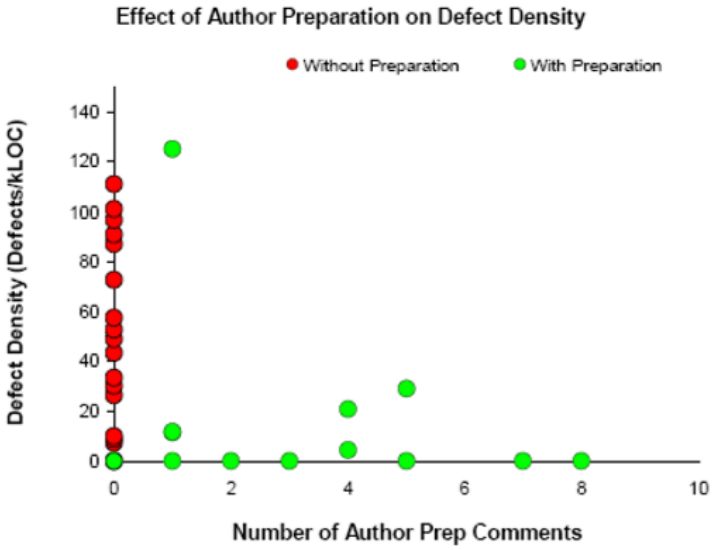

Figure 3: The striking effect of author preparation on defect density.

5.- Establish quantifiable goals for code review and capture metrics so you can improve your processes.

you should decide in advance on the goals of the code review process and how you will measure its effectiveness. Once you've defined specific goals, you will be able to judge whether peer review is really achieving the results you require.

To improve and refine your processes, collect your metrics and tweak your processes to see how changes affect your results. Pretty soon you'll know exactly what works best for your team

6.- Checklists substantially improve results for both authors and reviewers.

Checklists are a highly recommended way to find the things you forget to do, and are useful for both authors and reviewers. Another useful concept is the personal checklist. Each person typically makes the same 15-20 mistakes. As soon as you start recording your defects in a checklist, you will start making fewer of them.

7.- Verify that defects are actually fixed!

If you're going to go to the trouble of finding the bugs, make sure you've fixed them all!

8.- Managers must foster a good code review culture in which finding defects is viewed positively.

The point of software code review is to eliminate as many defects as possible – regardless of who "caused" the error. Reviews present opportunities for all developers to correct bad habits, learn new tricks and expand their capabilities.

To maintain a consistent message that finding bugs is good, management must promise that defect densities will never be used in performance reports.

9.- Beware the "Big Brother" effect.

Metrics should never be used to single out developers, particularly in front of their peers. This practice can seriously damage morale.

If metrics do help a manager uncover an issue, singling someone out is likely to cause more problems than it solves. We recommend that managers instead deal with any issues by addressing the group as a whole. It's best not to call a special meeting for this purpose, or developers may feel uneasy because it looks like there's a problem. Instead, they should just roll it into a weekly status meeting or other normal procedure.

Managers must continue to foster the idea that finding defects is good, not evil, and that defect density is not correlated with developer ability.

10.- The Ego Effect: Do at least some code review, even if you don't have time to review it all.

The "Ego Effect" drives developers to write better code because they know that others will be looking at their code and their metrics. And no one wants to be known as the guy who makes all those junior-level mistakes. The Ego Effect drives developers to review their own work carefully before passing it on to others.

Reviewing 20-33% of the code will probably give you maximal Ego Effect benefit with minimal time expenditure, and reviewing 20% of your code is certainly better than none!

11.- Lightweight-style code reviews are efficient, practical, and effective at finding bugs.

The average heavyweight inspection takes nine hours per 200 lines of code. While effective, this rigid process requires three to six participants and hours of painful meetings paging through code print-outs in exquisite detail.

the most effective reviews are conducted using a collaborative software tool to facilitate the review. A good lightweight code review tool integrates source code viewing with "chat room"

collaboration to free the developer from the tedium of associating comments with individual lines of code.

Software Inspections

Introduction

In the book I address both subjects using two primary processes:

1. Inspections to find defects earlier and at a lower cost

2. Defect Prevention to reduce the volume of defects injected

The lesson to be learned from these experiences is that methods and tools can be misapplied, treated as a failure, and then dismissed as a bad experience by users who were not enabled for success.

Why Inspections?

We need Inspections to remove software defects at reduced cost.

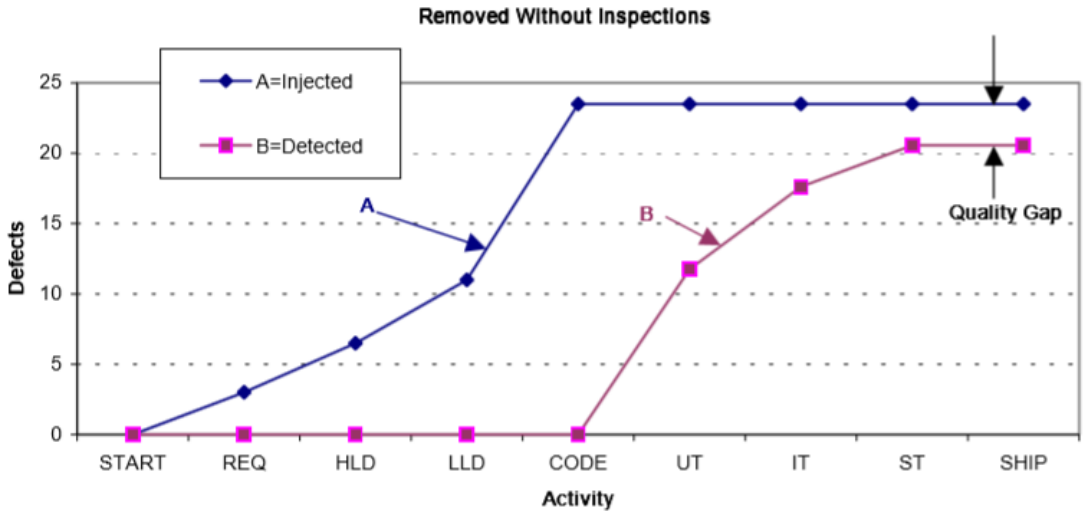

Figure 1. Example of Defect Injection and Defect Removal Curve

So far in this example we can see that:

• Defects are generated in each life cycle production activity

• Injected defects are removed in testing activities after code is completed

• Not all defects are removed at SHIP

Had we tried to deliver the product at the conclusion of coding, we would most probably not have had a working product due to the volume of defects.

Our objective with Inspections is to reduce the Cost of Quality by finding and removing defects earlier and at a lower cost.

While some testing will always be necessary, we can reduce the costs of test by reducing the volume of defects propagated to test.

So the problems that must addressed are:

• Defects are injected when software solutions are produced

• Removal of defects in test is costly

• Users are impacted by too many software defects in delivered products

Effectiveness is the percentage of defects removed from the base of all the defects entering the defect removal activity. As is seen in Figure 2, the number of latent defects in the final product is reduced with Inspections.

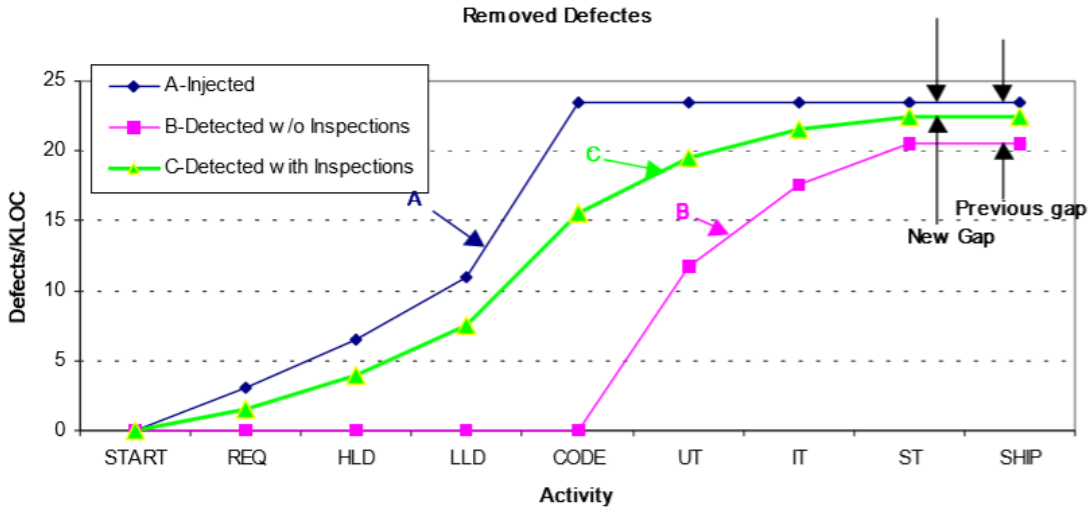

Figure 2. Example of Defect Removal With Inspections

In the scenario, with Inspections, the gap is smaller, due to the defects removed by Inspections. This reduced gap represents a quality improvement in the product delivered to the users.

Ok, so we have removed defects earlier, but why is it cheaper you may ask. It is cheaper primarily due to the following:

- Defects are not discovered all at once during test or by the users. There are likely to be a number of cycles of finding defects, fixing them, and integrating the fixes into each delivery to test.
- The increased labor hours required for fixing defects after the product is shipped is often due to loss of project team knowledge
- When fewer defects enter test, the productivity of test improves; i.e., the costs of test are lower and the time to complete test is reduced.

Besides the costs to the project, we should also note that there is a cost to the customer for downtime, lost opportunity, etc. These costs while often transparent to the project can have a negative effect on future sales or contracts with the customer.

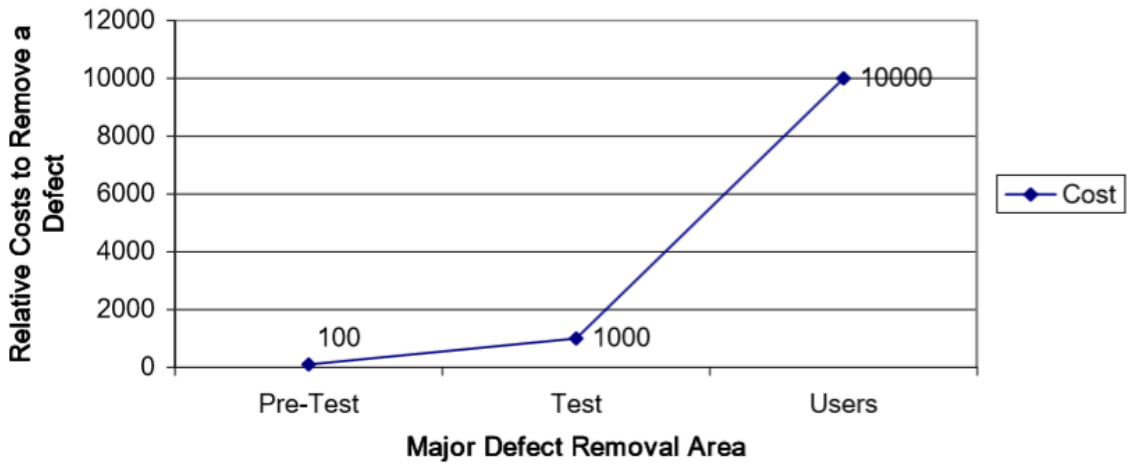

Figure 3 . Defect Cost Relationship

Inspections have clear value independent of any model or standard for software development.

So Why isn't Everyone Using Inspections

There are factors, including social and psychological, that we must consider. Management often views Inspections as an added cost, when in fact Inspections will reduce costs during a project.

Do Good Programmers Make Defects?

People and processes become more capable as an organization moves up the SW-CMM levels. An example is shown in Figure 4 for the Inspection process from organizations I have come across over the past twenty-eight years. This figure represents by example that as an organization moves up the SW-CMM ladder, the effectiveness; i.e., the percentage of defects removed with Inspections increases with each level attained. Improved effectiveness should also be seen in other processes as an organization's maturity grows, but for Inspections it is easier and faster to see.

SW-CMM LEVEL	EFFECTIVENESS
1	<50%
2	50-65%
3	65-75%
4	75-90%
5	90-100%

Figure 4. Inspection Effectiveness and Maturity Levels

Effectiveness and Efficiency

Let us now take a look at the distinction between effectiveness and efficiency. They are different and both have business; i.e., economic, value to organizations practicing Inspections.

Effectiveness

Effectiveness of Inspections is the percentage of defects removed by Inspections compared to the total sum of defects eventually found by Inspections, test, and in use by the customers.

Efficiency

Efficiency of Inspections is represented by various cost relationships; e.g., :

• $ spent / defect found in Inspections

• $ for Inspections / total project costs; this is a subset of Cost of Quality (COQ) measures

• $ ratio of cost of finding defects in Inspection to cost of defects found in test and by the customers

• hours spent / defect found in Inspections

1:1 Inspections

1:1 Inspections occur when there are only two participants in an Inspection, the Producer and the Inspector/Moderator. If you can perform a 1:! Inspection you can typically save about 50% compared to the traditional Inspection.

vv

Moderator Task	Required?	Who Performs
1. Inspection Scheduling	Yes	PL
1.1 Determine need for Overview	Yes	PL & I
1.2 Determine Inspection team	Yes	PL
1.3 Ensuring availability of materials	Yes	P
1.4 Assigning roles	NA	NA
1.5 Chunking materials	Yes	PL & P
1.6 Defining activities schedule	Yes	PL & P & I
1.6.1 Overview	Optional	PL & I
1.6.2 Preparation effort	Yes	PL & P & I
1.6.3 Inspection Meeting duration	Yes	PL & P & I
1.6.4 Analysis Meeting	Optional	PL & P & I
1.6.5 Logistics	Yes	PL & I
2. Overview	Optional	P & I
3. Preparation	Yes	P & I
4. Inspection Meeting	Yes	P & I
5. Data Recording	Yes	I
6. Analysis Meeting	Optional	P & I
7. Rework	Yes	P
8. Follow-up	Yes	P & I

Table 1. Tasks Assignments in 1:1 Inspections

Can Inspections Replace Test?

I previously discussed that the solution for a defect is usually evident when it is found during an Inspection, thus the costs for repair are minimized. Tests after unit test do not make readily identifiable the area requiring repair and thus the costs increase.

- As Inspections require a motivated team, testing first may lead to a view that the code is reasonably stable and the team will be less motivated to perform the best Inspection.
- With the investment of test the Producer may be less receptive to major rework on an “already-stable program image” which will also require retesting.

Ackerman found that the savings from defect detection costs in Inspections was 2.2 hours compared to 4.5 hours in unit test. A two to one savings is a good place to bank. In another organization he states a 1.4 to 8.5 staff hour relationship in finding defects with Inspections versus testing.

Weller [WEL93] states that there are disadvantages of inspecting after unit test:

- Unit test leads programmers to have false confidence that the product works, so why inspect
- It is a hard decision to inspect a large batch that has been unit tested and there may be the view that there is no longer time to inspect

He also gives reasons to perform Inspections first:

- You may actually be able to bypass unit test if the Inspection results are good
- You can recover earlier with lower cost to serious design defects found in Inspections versus unit test

Despite this (and other) clear evidence, to this day many will suggest that unit test should be done before Inspections. Always this has proven to be less efficient. When I run into people who are hard to convince.

Will Inspections Ever Become Obsolete?

There is no reason why programmers should be put into positions of generating 100's of defects to be found by Inspections.

Inspections should not be a requirement to achieve quality and they are not required for all work products.

Solo:Inspections

In the Solo:Inspections, the inspector logs on to his workstation and brings up the work product to be inspected. He has all required documents available through the organization's intranet for ancillary and reference documents he may need. He has access to the appropriate checklist, standards, and forms to record any defects found. He can open as many concurrent windows or views with required documents as he needs to perform the Inspection. His time is recorded based on log-on time. When he is done, he submits his Inspection documents to the Producer, Inspection Coordinator, and Project Lead. He is then available to the Producer if there should be any questions.

Another benefit is that the Inspection can be for longer than two hours, as the inspector can start and stop, chunking as he feels necessary. In Solo:Inspections the inspector makes a commitment to the Project Lead for a completion time or date and this is what he then can concurrently manage along with other work in his queue.

Organizations that are using Solo:Inspections move to them when their traditional Inspections have already proven high effectiveness. Since there is feedback and coaching to the inspectors about their effectiveness during this transition, the Solo:Inspection effectiveness remains high.

Another advantage is that the Producer and inspector can be in two different locations. The inspectors I have observed seemed fully comfortable with this remote approach. One might wonder that given the starts and stops whether the inspector lost continuity of thinking and if this affected effectiveness.

Moderator Task	Required?	Who Performs
1. Inspection Scheduling	Yes	PL
1.1 Determine need for Overview	Yes	PL & I
1.2 Determine Inspection team	Yes	PL
1.3 Ensuring availability of materials	Yes	P
1.4 Assigning roles	NA	NA
1.5 Chunking materials	Yes	I
1.6 Defining activities schedule	Yes	PL & I
1.6.1 Overview	Yes	PL & I
1.6.2 Preparation effort	Yes	PL & I
1.6.3 Inspection Meeting duration	Yes	PL & I
1.6.4 Analysis Meeting	Optional	PL & P & I
1.6.5 Logistics	Yes	PL & I
2. Overview	Optional	P & I
3 Preparation	Yes	I
4. Inspection Meeting	Yes	I
5. Data Recording	Yes	I
6. Analysis Meeting	Optional	P & I
7. Rework	Yes	P
8. Follow-up	Yes	P & I

Table 2. Solo:Inspection Activity Assignments

Solo:Inspections, to date, have been best applied for code and Low-level design work products, but the concept could work for other work product types. Some work products, such as requirements specifications, may never be able to use Solo: Inspections, but most effort on Inspections is not in these work products anyway.