



# Code improvement through peer code reviews

---

UA/DETI/TQS

Ilídio Oliveira ([ico@ua.pt](mailto:ico@ua.pt))

v2019-03-26.

# Motivation for a cleaner code

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>



# You can make the code easier to maintain

## Practices to find problems in the code

- ▶ Static code analysis (inspectors)
- ▶ Code reviews
- ▶ ...

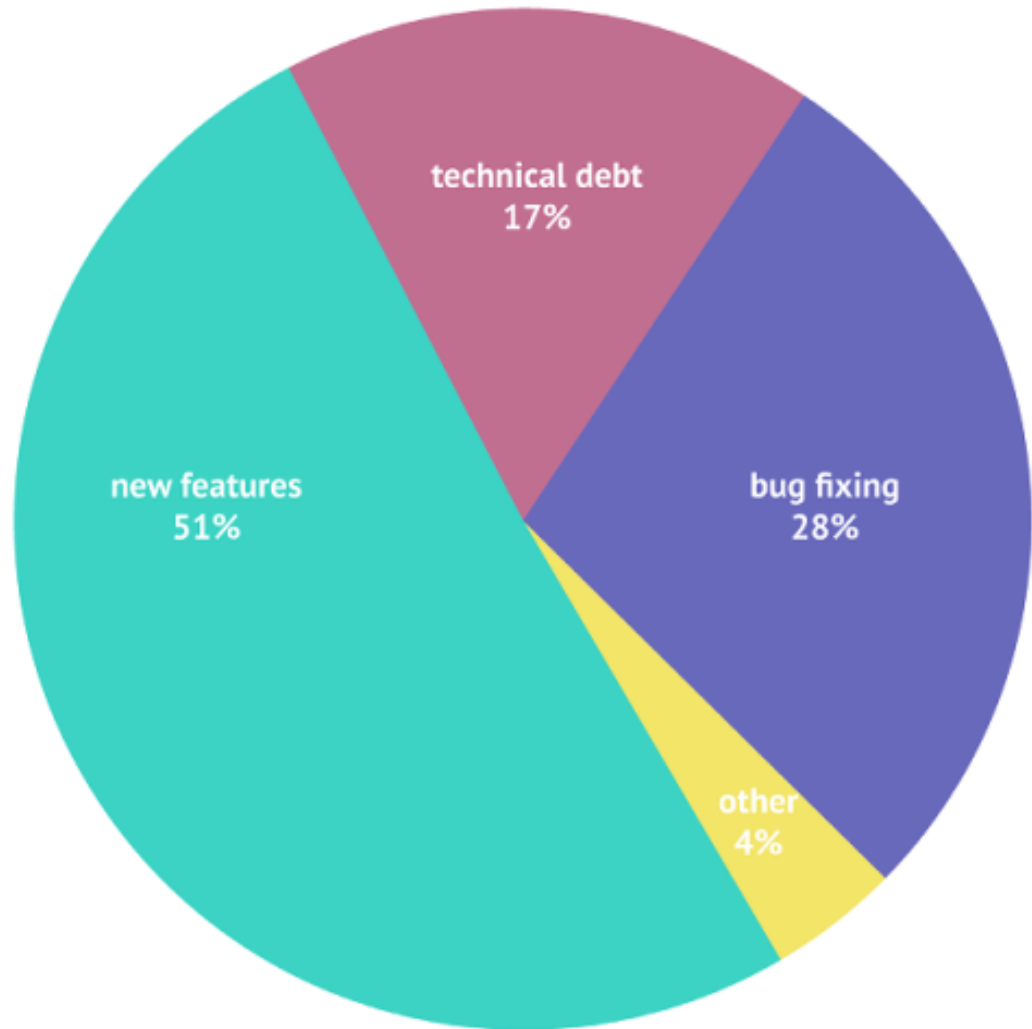
## Culture for clean/readable code

- ▶ Developer/style guidelines
- ▶ Patterns
- ▶ ...



## Time spent on

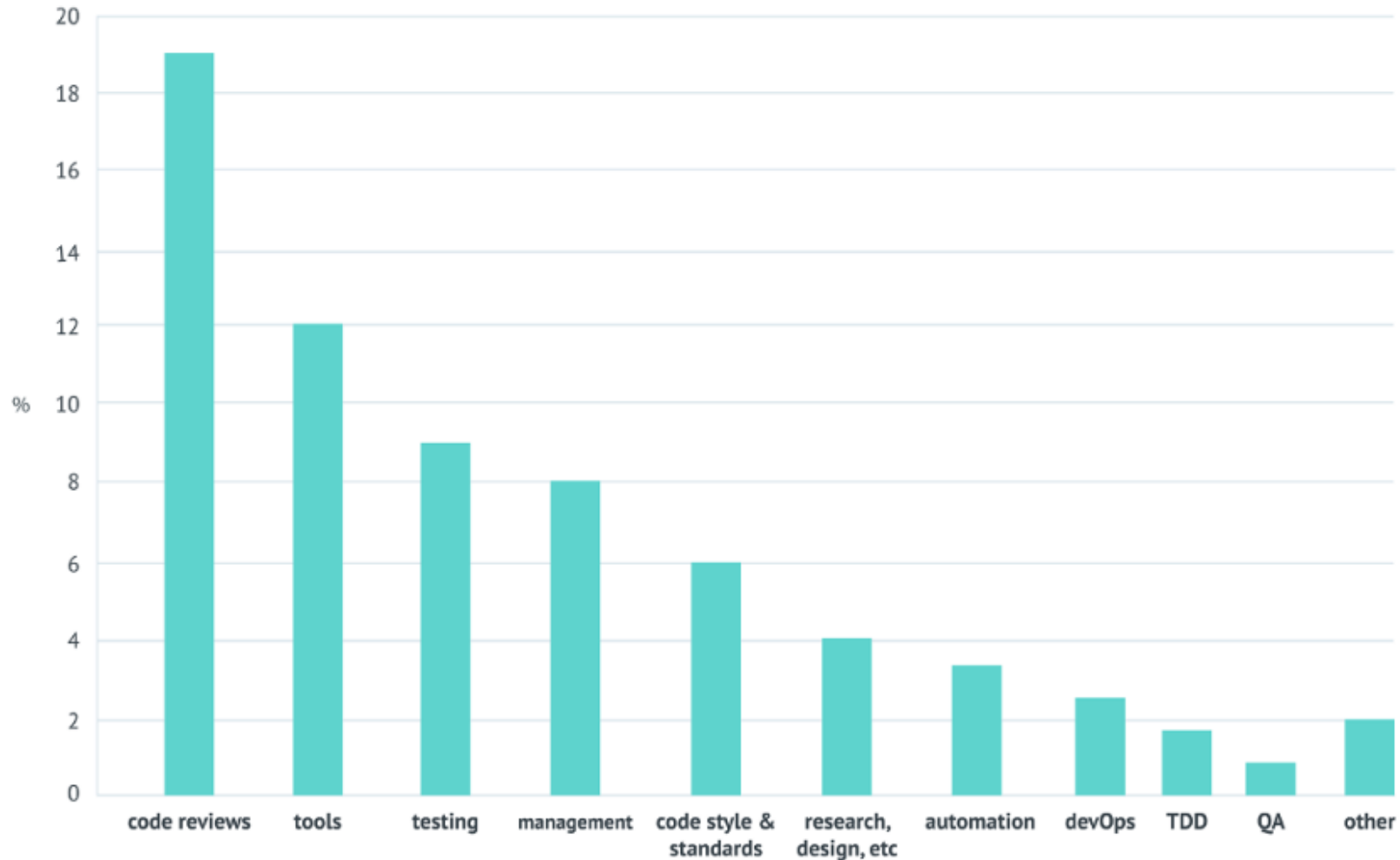
- new features
- technical debt
- bug fixing
- other



<https://www.codacy.com/ebooks/guide-to-code-reviews>



## What change in your development process had the biggest impact to code quality?\*

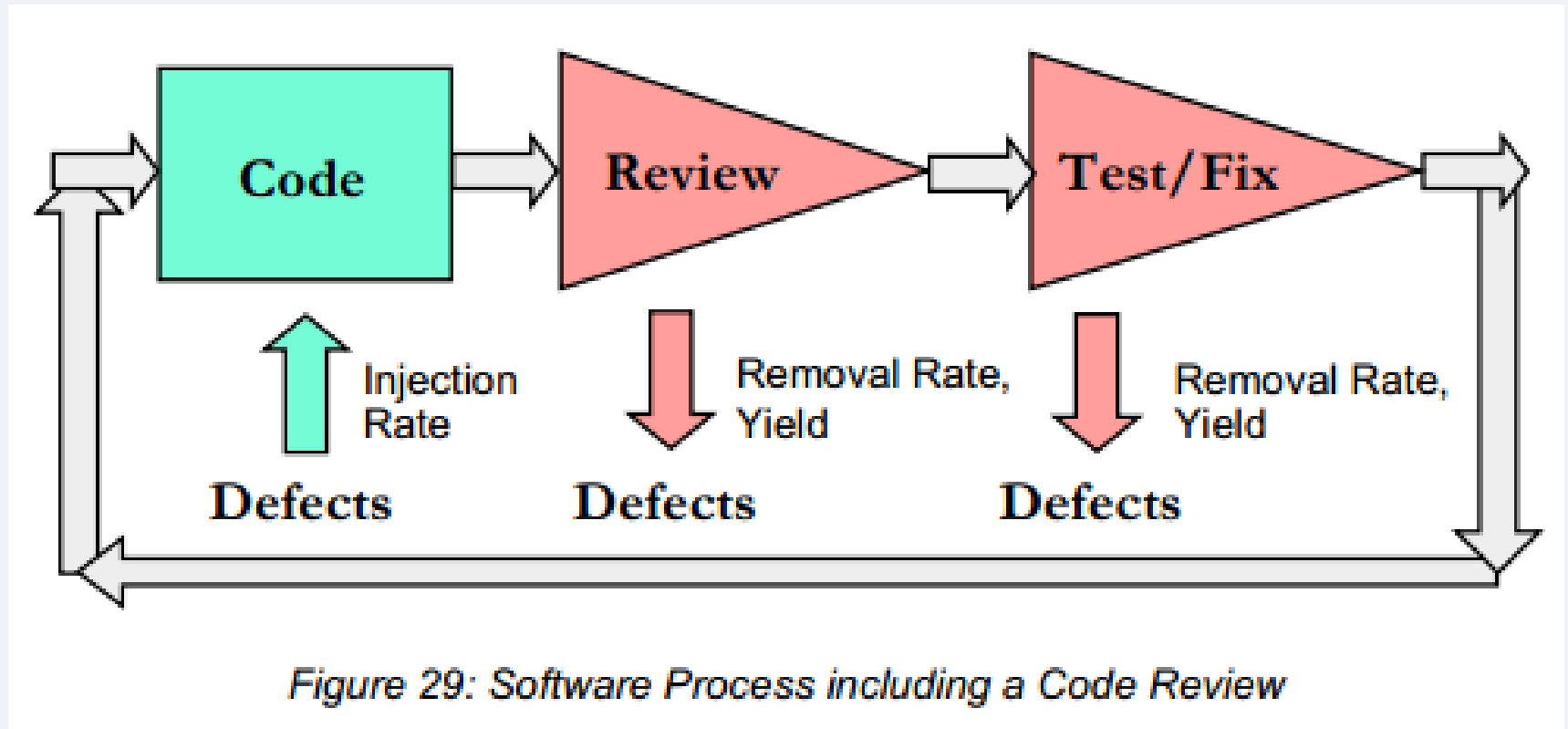


change in development process

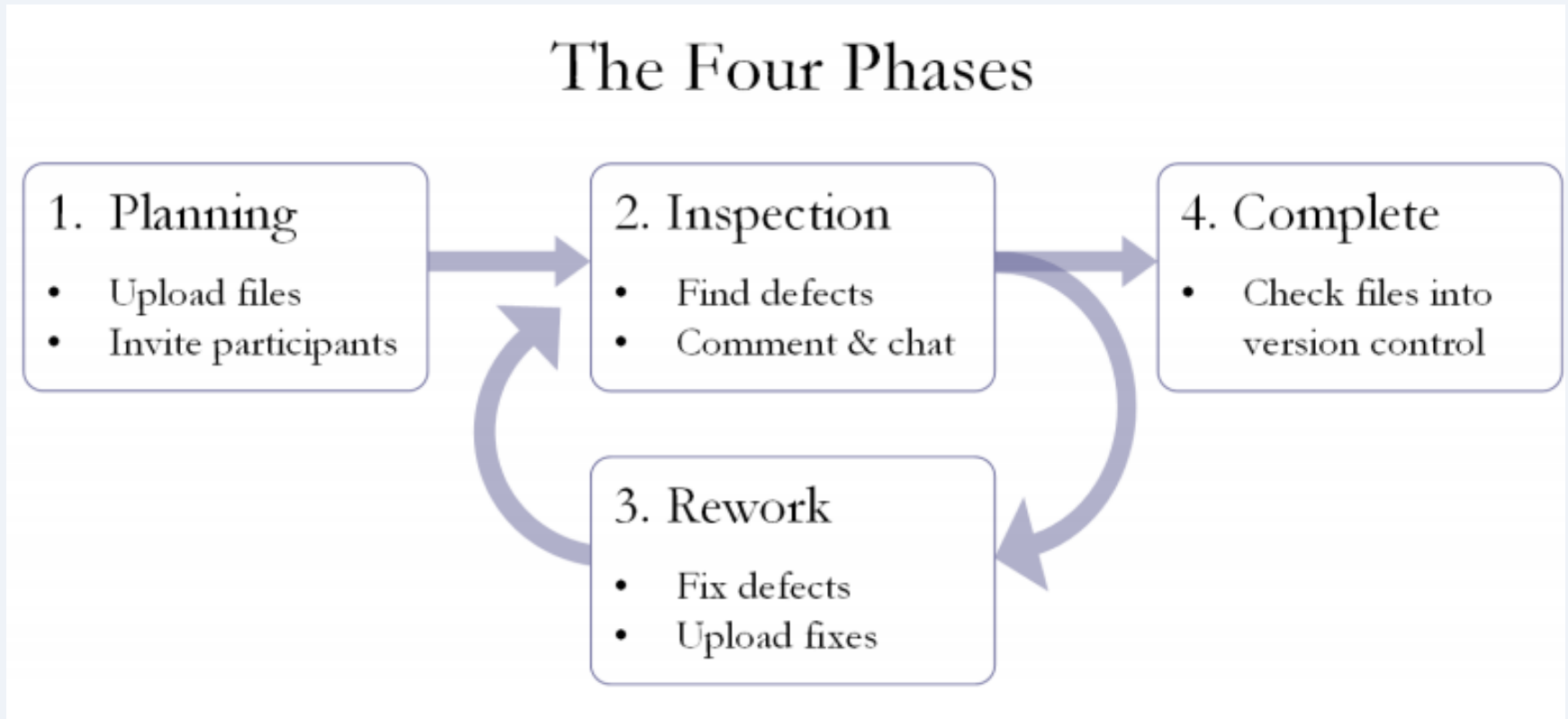
<https://www.codacy.com/ebooks/guide-to-code-reviews>

\*The question was open-ended to avoid leading the respondents into specific answers.

# Code review in the software development process



The code review lifecycle has four main stages



See also: [Cohen's book on code review](#)



# Defects most likely to find in a code review

## Deviations from standards

- ▶ either internally defined and managed or regulatory/legally defined

## Requirements defects

- ▶ e.g.: the requirements are ambiguous, or there are missing elements.

## Design defects

- ▶ e.g.: too much coupling; fail to use [known patterns](#)

## Insufficient maintainability

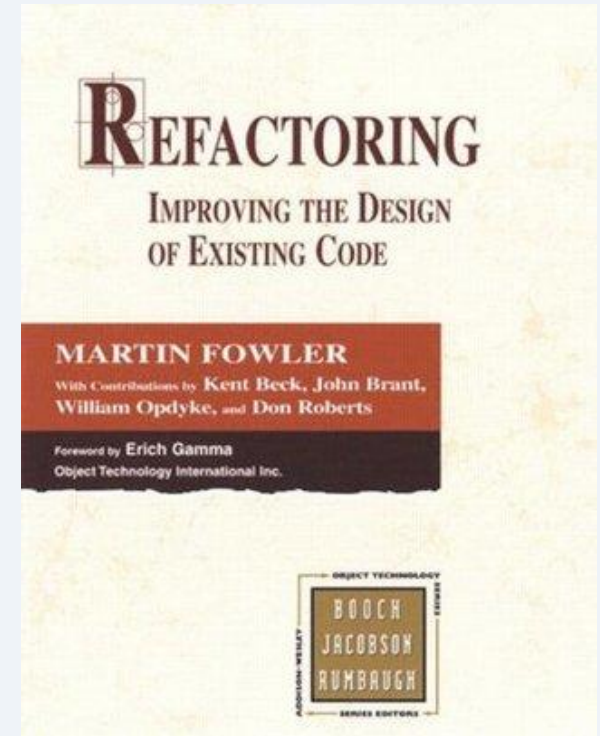
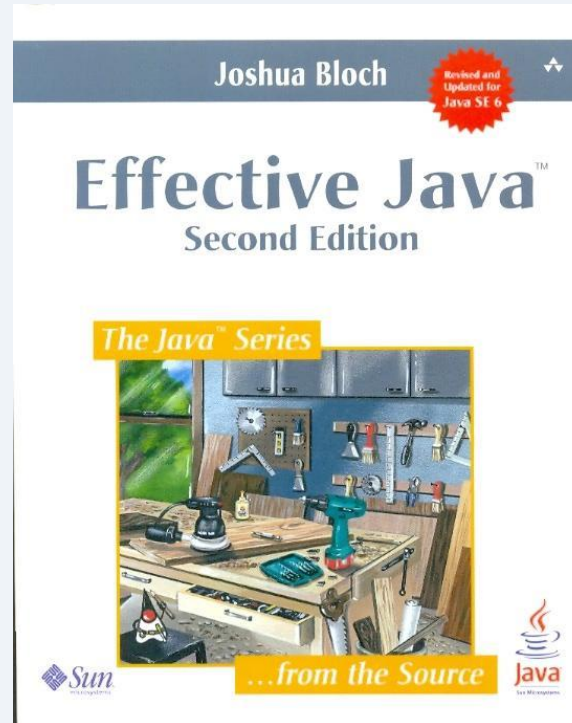
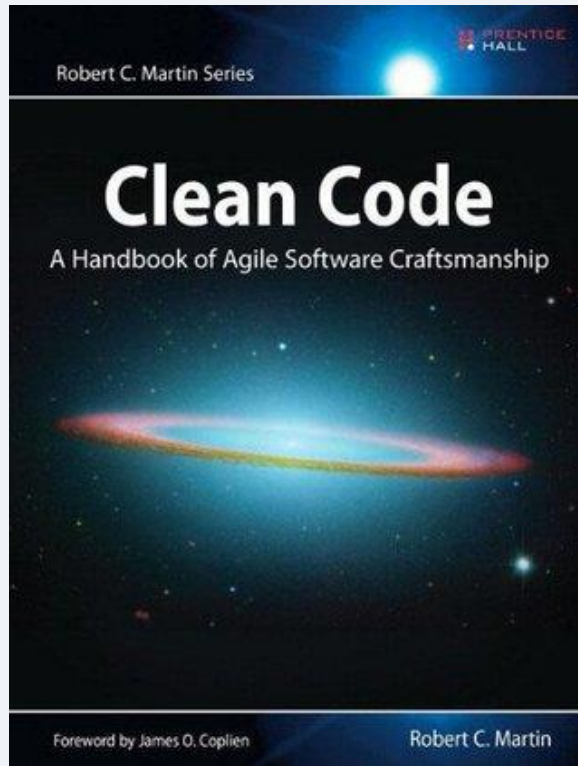
- ▶ e.g.: the code is too complex to maintain

## Incorrect interface specifications





# What to look for in code reviews ?



The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

The name `d` reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Choosing names that reveal intent can make it much easier to understand code. What is the purpose of this code?

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

## Chapter 2: Meaningful Names

### Introduction

### Use Intention-Revealing Names

#### Avoid Disinformation

#### Make Meaningful Distinctions

#### Use Pronounceable Names

#### Use Searchable Names

#### ► Avoid Encodings

#### Avoid Mental Mapping

#### Class Names

#### Method Names

#### Don't Be Cute

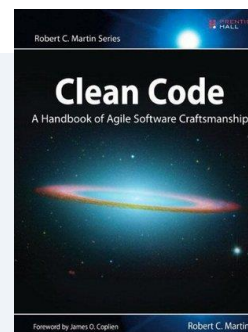
#### Pick One Word per Concept

#### Don't Pun

#### Use Solution Domain Names

#### Use Problem Domain Names

#### Add Meaningful Context



# Explain Yourself in Code

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Or this?

```
if (employee.isEligibleForFullBenefits())
```

It takes only a few seconds of thought to explain most of your i cases it's simply a matter of creating a function that says the same you want to write.

## Chapter 4: Comments

### Comments Do Not Make Up for Bad Code

#### Explain Yourself in Code

#### ▼ Good Comments

Legal Comments

Informative Comments

Explanation of Intent

Clarification

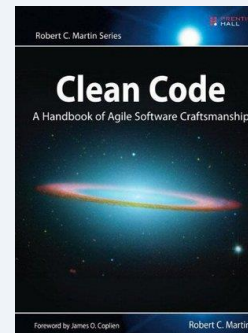
Warning of Consequences

TODO Comments

Amplification

Javadocs in Public APIs

#### ► Bad Comments



# Effective Java

## **2 Creating and Destroying Objects. . . . .5**

- Item 1: Consider static factory methods instead of constructors . . . 5
- Item 2: Consider a builder when faced with many constructor parameters . . . . . 11
- Item 3: Enforce the singleton property with a private constructor or an enum type . . . . . 17
- Item 4: Enforce noninstantiability with a private constructor . . . 19
- Item 5: Avoid creating unnecessary objects . . . . . 20
- Item 6: Eliminate obsolete object references . . . . . 24
- Item 7: Avoid finalizers . . . . . 27

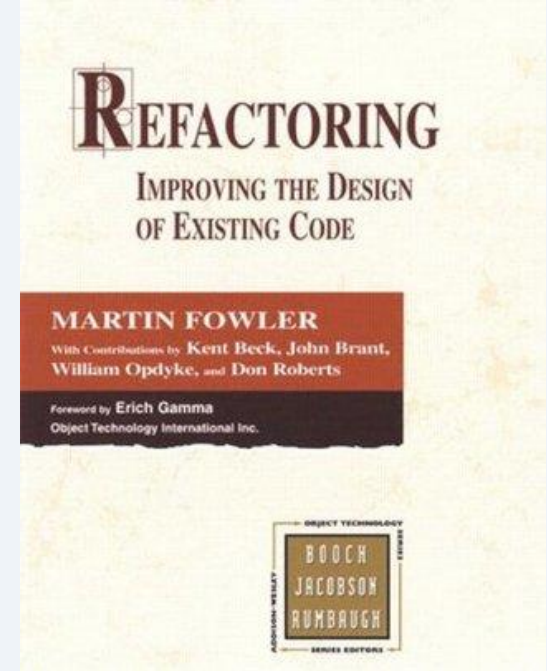
**Item 1: Consider static factory methods instead of constructors**

**Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors**

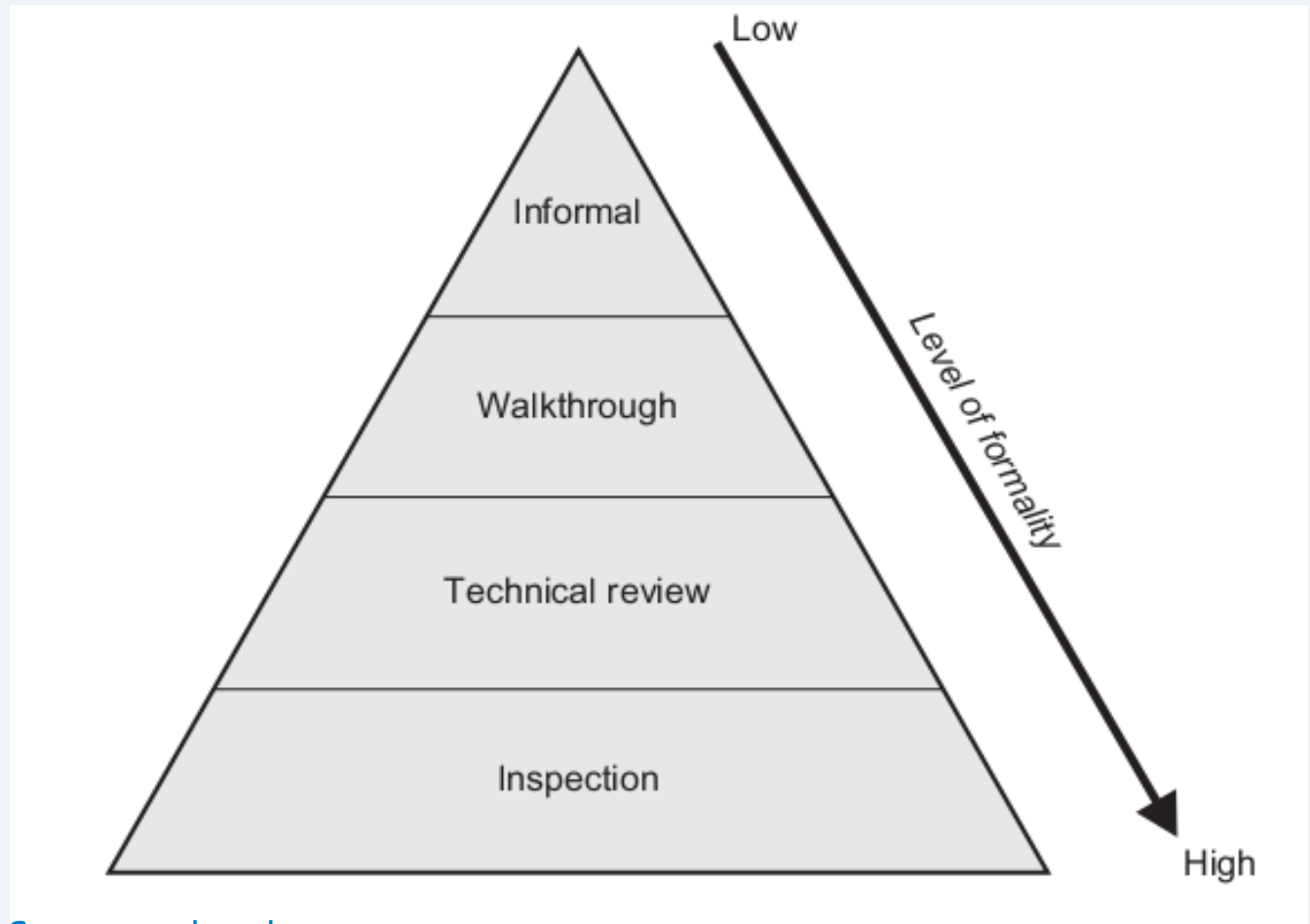


## "Bed smells" (M. Fowler)

- ▶ Duplicate code
- ▶ Very long methods
- ▶ Large class
- ▶ Long parameter list
- ▶ Feature envy
- ▶ Primitive obsession
- ▶ "When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous."
- ▶ ...



A review process can have very different levels of formality (Informal to FTR)

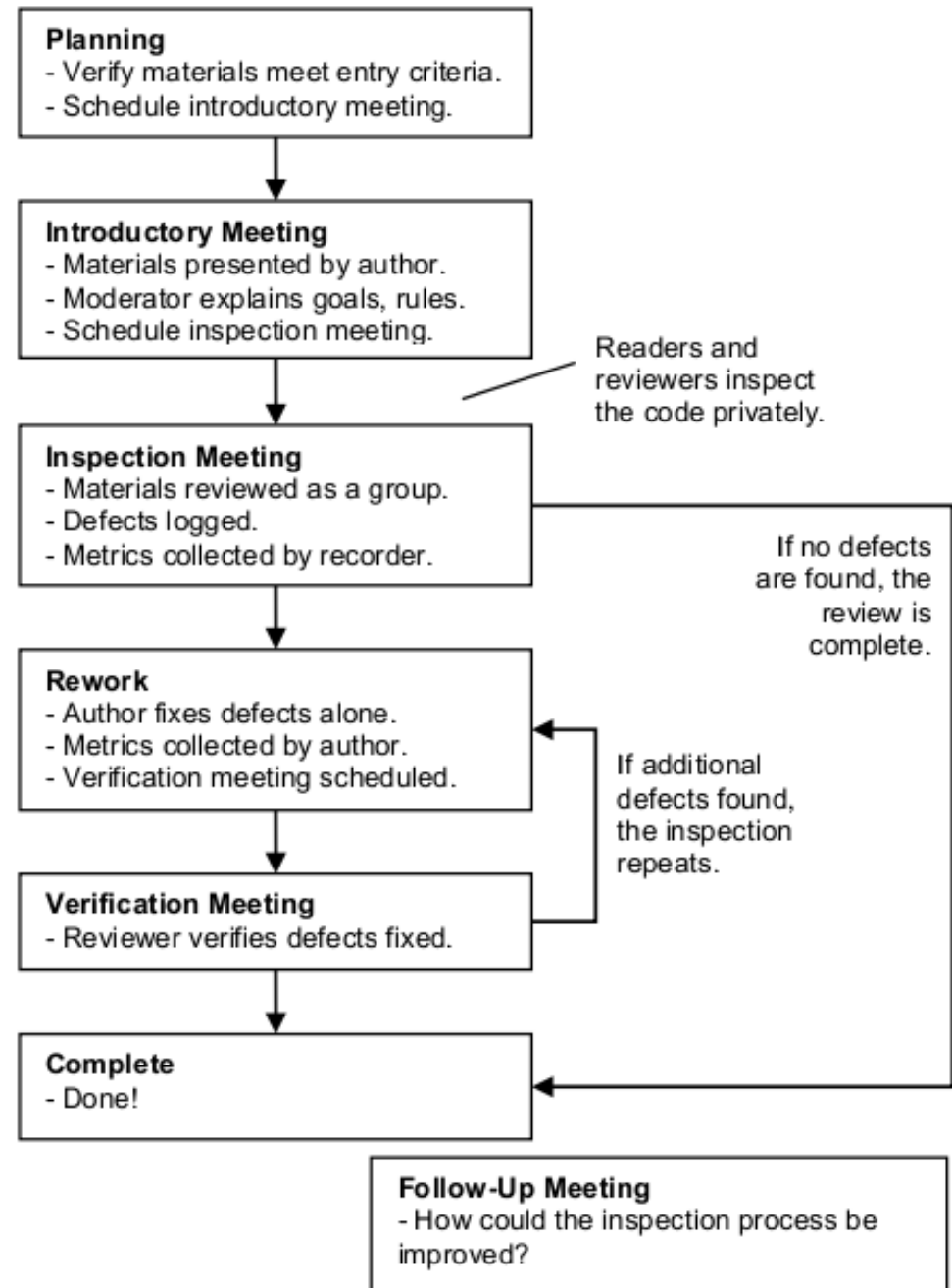


More info:

Wiegers, K. E. (2002). Seven truths about peer reviews. Cutter IT Journal, 15(7), 31-37.



## A Typical Formal Inspection Process



In: P. Farrell-Vinay, Manage Software Testing. Taylor & Francis, 2008.



# Formal technical review [R. Pressman]

## Objectives

- ▶ to uncover errors in function, logic, or implementation for any representation of the software;
- ▶ to verify that the software under review meets its requirements;
- ▶ to ensure that the software has been represented according to predefined standards;
- ▶ to achieve software that is developed in a uniform manner;
- ▶ to make projects more manageable.

## FTR serves as a training ground

- ▶ junior engineers → observe different approaches to software analysis, design, and implementation.







+ OSMA

+ Front Office

+ MSD

+ SARD

+ RAD

+ Centers

+ Return to OSMA Home  
Page

**Software Assurance  
Home**

+ Contacts

+ Documents

+ Research

+ Training

+ Working Groups

+ Complex Electronics ▶

+ Links

+ Questions?



## Documentation

Links to the current releases of our software assurance documents are provided below. When new documents are created, or existing documents are updated, the list of links will be revised accordingly.

[NASA Software Assurance Standard \(NASA-STD-8739.8\)](#)

[NASA Software Safety Standard \(NASA-STD-8719.13C\)](#)

[NASA Software Safety Guidebook \(PDF, large file, >6MB\)](#)

[Complex Electronics Handbook for Assurance Professionals \(NASA-HDBK-8739.23\)](#)

[NASA Software Formal Inspections Standard \(NASA-STD- 8739.9\)](#)

<https://standards.nasa.gov/standard/nasa/nasa-std-87399>

# Lightweight techniques for code review

## Over-the-shoulder

- ▶ a developer stands over the author's shoulder as the latter walks through the code changes.

## Email pass-around

- ▶ The author (or SCM system) emails code to reviewers

## Pair Programming

- ▶ Two authors develop code together at the same workstation.

## Tool-assisted reviews

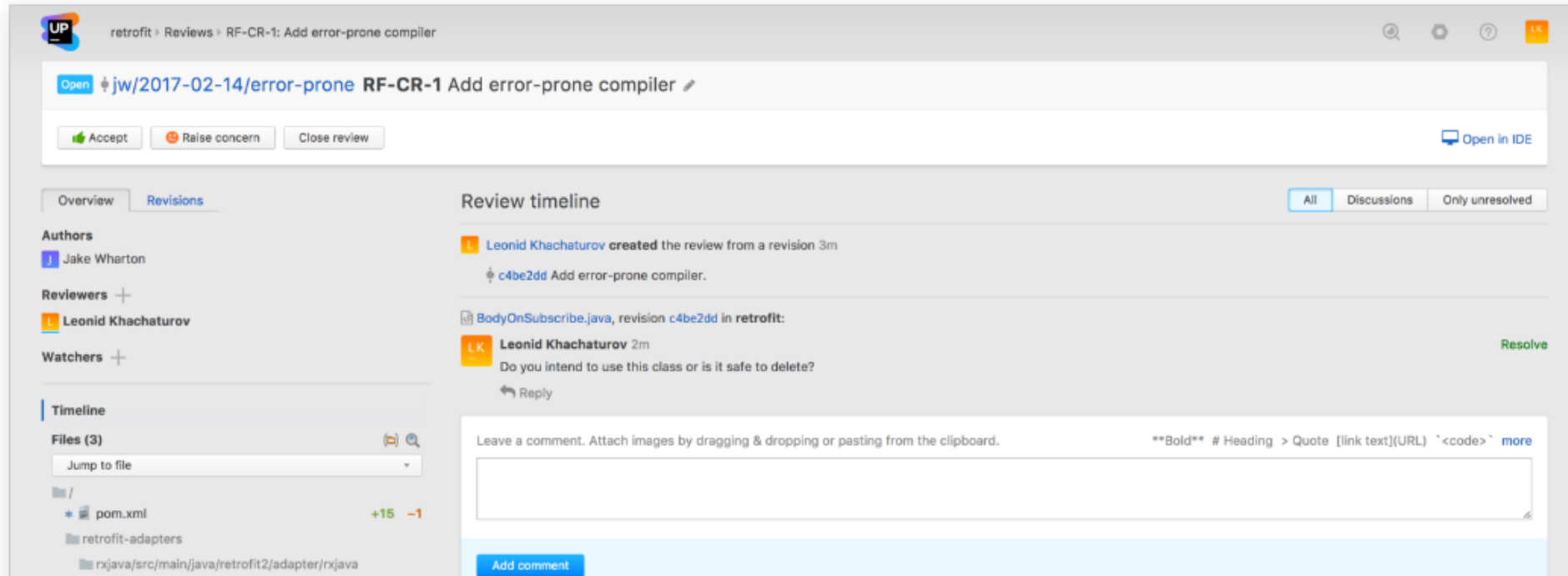
- ▶ Authors and reviewers use specialized tools designed for peer code review.
- ▶ Collect changes, support discussions, visualize diffs,...
- ▶ E.g.: [Collaborator](#), Guerrit



# Efficient Code Review

Performing ad-hoc code reviews provides an opportunity to improve code quality, enhance team collaboration, and learn from each other.

As Upsource does not impose any strict workflow, you can fit it into your preferred process: create a code review for a recent commit, for an entire branch, or review a GitHub pull request.



The screenshot displays the Upsource interface for a code review titled "RF-CR-1 Add error-prone compiler". The interface includes a sidebar with navigation tabs for "Overview" and "Revisions". The "Overview" tab is active, showing the review details. The review was created by Leonid Khachaturov 3m ago. The review timeline shows a comment from Leonid Khachaturov asking "Do you intend to use this class or is it safe to delete?". The interface also includes a "Files (3)" section on the left, a "Timeline" section, and a "Comments" section at the bottom.

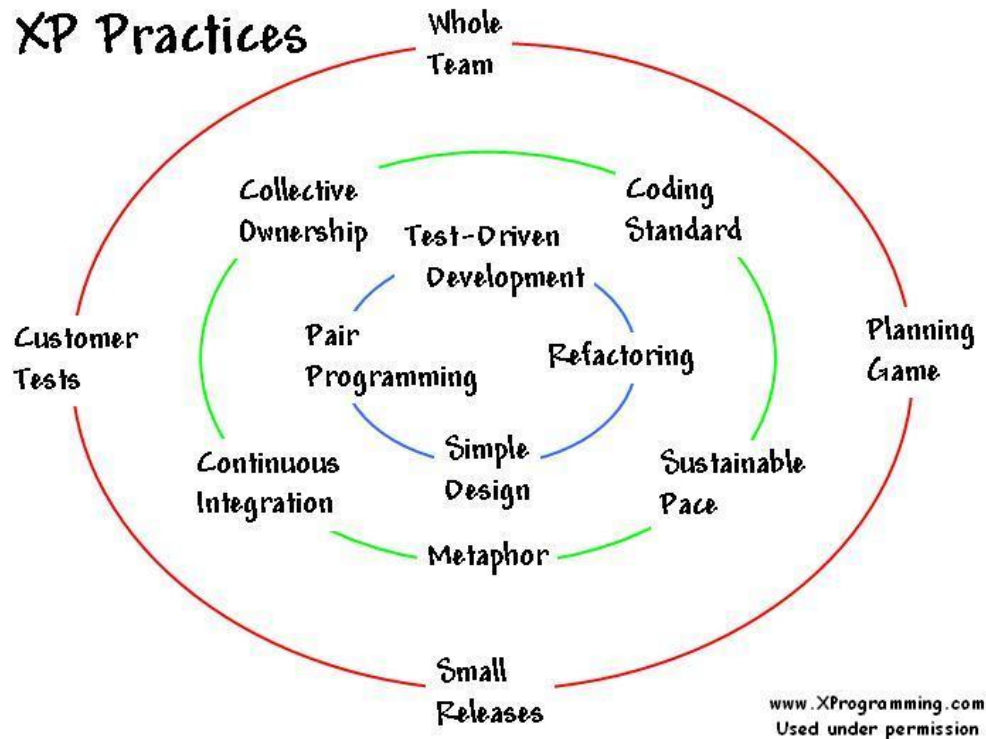
<http://www.jetbrains.com/upsource/>

<https://www.atlassian.com/software/crucible>

<https://smartbear.com/product/collaborator/overview/>



# Pair programming does code review all the time



- ▶ all production code is written by pairs of programmers
- ▶ Each pair works together at a single workstation
- ▶ The code is the product of both brains, not just one (co-authors)
- ▶ A form of “continuous code review”

XP @Eclipse process framework



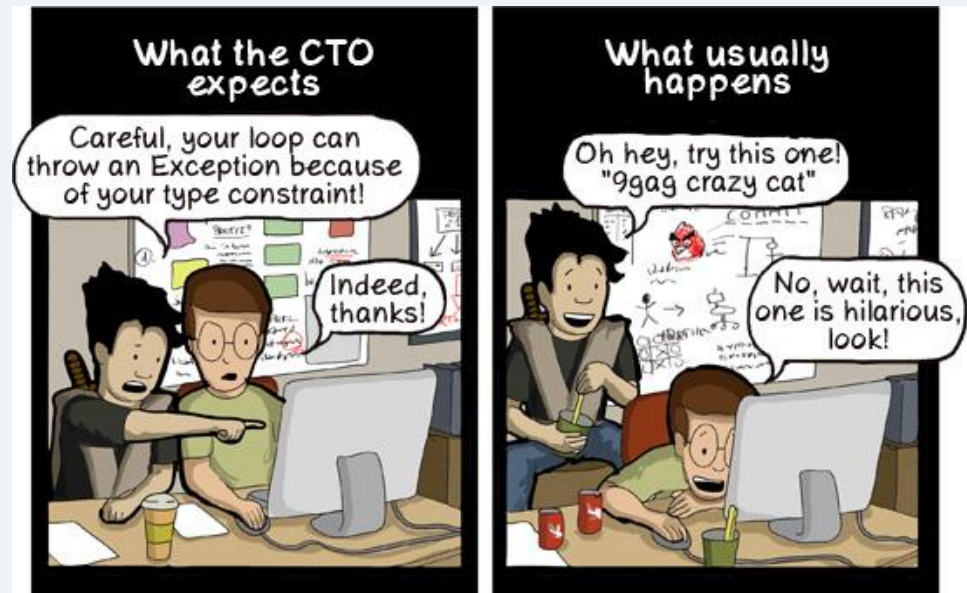
# Pair programming must be done right to be effective and productive

- ▶ Pairs are short-lived
- ▶ Half of the time, one is working on his own tasks (and then swap)
- ▶ You can't check in production code that you have written on your own.
- ▶ Excellent way to train a new team member in the existing code
- ▶ Newbies should pair most often with team members with more seniority...

<https://developer.atlassian.com/blog/2015/05/try-pair-programming/>



I Oliveira (2018)



CommitStrip

# Styles of code reviews

## Pre-commit review

- ▶ E.g.: discuss the changes with email, authorized maintainers will commit
- ▶ Not integrated in the history; only one author for a feature/patch

## Post-commit review

<https://youtu.be/6qKpbWyb6tg?t=1036>

- ▶ Diffs (added and removed lines)
- ▶ Review a single commit or a group of commits

## Gerrit-style

<https://www.gerritcodereview.com/>

- ▶ Specific workflow
- ▶ Fetch, push to staging branch, vote (score)

## Pull request (in Git)

- ▶ Review an unmerged branch before merge
- ▶ Different merge strategies
- ▶ Check whether build passes



# Integrate code review in the workflow with pull requests

All merge requests [...], whether written by a team member or a volunteer contributor, must go through a code review process to ensure the code is effective, understandable, and maintainable.

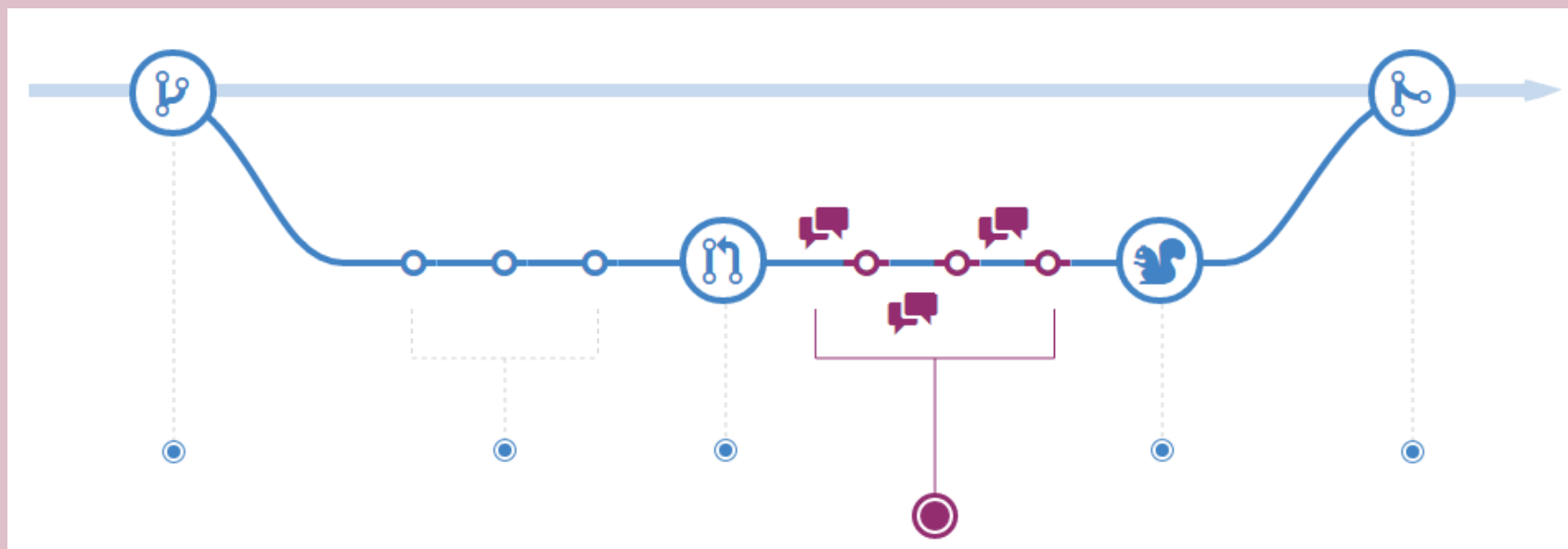




# Understanding the GitHub Flow

🕒 5 minute read

📄 Download PDF version



<https://guides.github.com/introduction/flow/index.html>

In detail: <http://scottchacon.com/2011/08/31/github-flow.html>





# Feature branch workflow

## Git Protocol

A guide for programming within version control.

## Maintain a Repo

- Avoid including files in source control that are specific to your development machine or process.
- Delete local and remote feature branches after merging.
- Perform work in a feature branch.
- Rebase frequently to incorporate upstream changes.
- Use a [pull request](#) for code reviews.

## Write a Feature

Create a local feature branch based off master.

```
git checkout master
git pull
git checkout -b <branch-name>
```

<https://github.com/thoughtbot/guides/tree/master/protocol/git>

Rebase frequently to incorporate upstream changes.



## **RULE 1**

Do the code reviews before deployment. Your team will end up, on average, spending 7 percentage points% more of its time on building new features compared with those who do after, and 10 percentage points% more than those who don't do code reviews at all.

## **RULE 2**

Make sure all your developers get to review code. This will improve the feeling of empowerment, facilitate knowledge transfer, and improve developer satisfaction and productivity.

## **RULE 3**

The optimal amount of time to spend on code reviews is between 0.5 to 1 day per week per developer.

## **RULE 4**

Make code reviews blocking, that is, don't deploy before they have been carried out.

## **RULE 5**

Be strict and thorough when reviewing code. Your code quality and velocity will thank you.



# 11 Best Practices for Peer Code Review

> Open

## A SmartBear White Paper

1. Review fewer than 200-400 lines of code at a time.....
2. Aim for your inspection rate of less than 300-500 LOC/hour.....
3. Take enough time for a proper, slow review, but not more than 60-90 minutes.....
4. Authors should annotate source code before the review begins. ....
5. Establish quantifiable goals for code review and capture metrics so you can improve your processes..
6. Checklists substantially improve results for both authors and reviewers.....
7. Verify that defects are actually fixed!.....
8. Managers must foster a good code review culture in which finding defects is viewed positively.....
9. Beware the “Big Brother” effect.....
10. The Ego Effect: Do at least some code review, even if you don’t have time to review it all .....
11. Lightweight-style code reviews are efficient, practical, and effective at finding bugs.....

# Guidelines from real projects

[https://gitlab.com/help/development/code\\_review.md](https://gitlab.com/help/development/code_review.md)



I Oliveira(2018)

≡ Help



## Code Review Guidelines

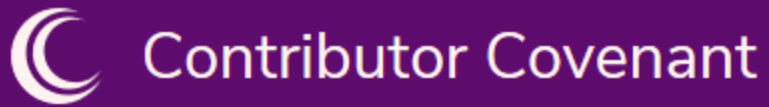
This guide contains advice and best practices for performing code review, and having your code reviewed. All merge requests for GitLab CE and EE, whether written by a GitLab team member or a volunteer contributor, must go through a code review process to ensure the code is effective, understandable, and maintainable.

Any developer can, and is encouraged to, perform code review on merge requests of colleagues and contributors. However, the final decision to accept a merge request is up to one the project's maintainers, denoted on the [team page](#).

### Everyone

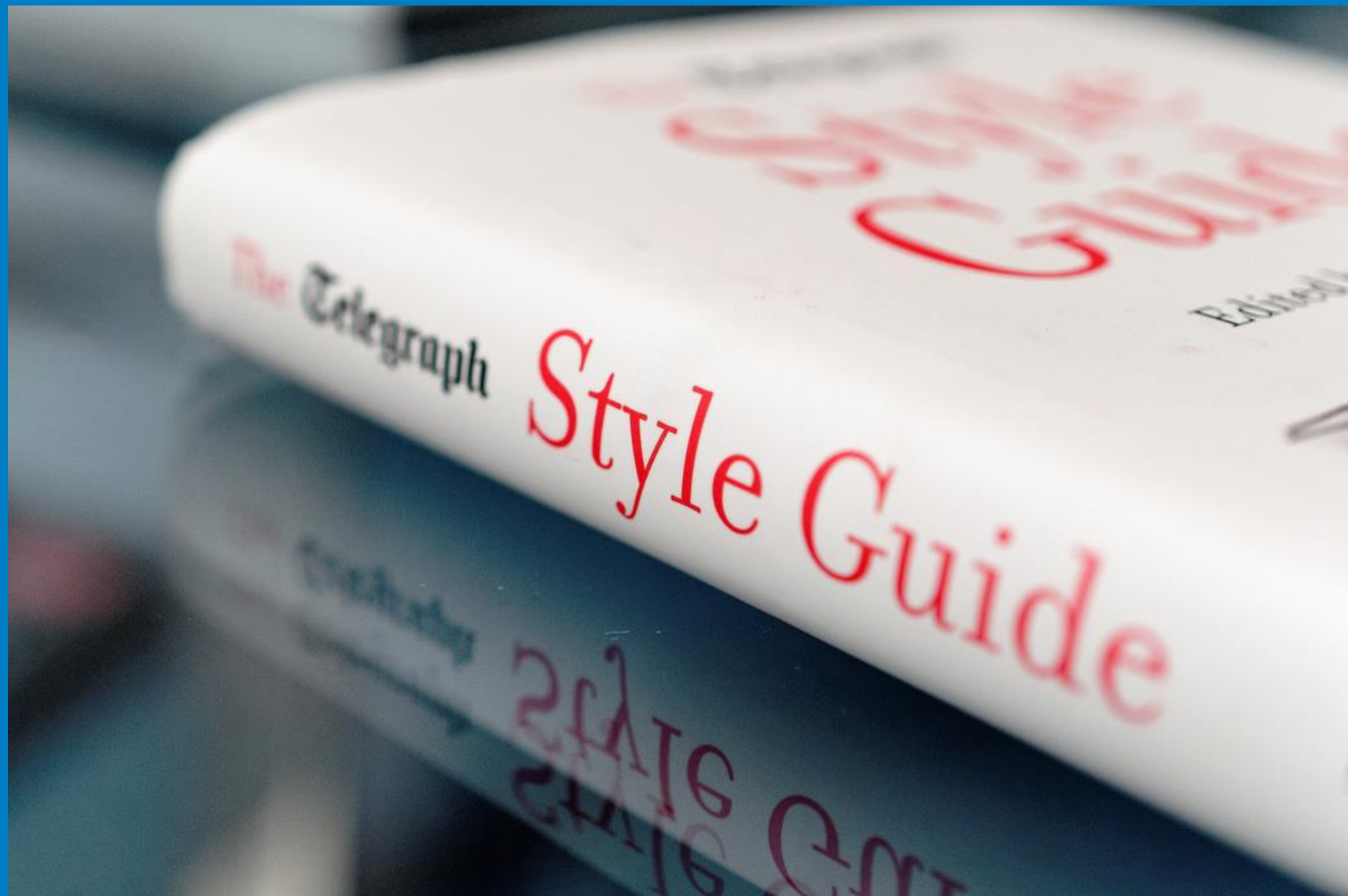
- Accept that many programming decisions are opinions. Discuss tradeoffs, which you prefer, and reach a resolution quickly.
- Ask questions; don't make demands. ("What do you think about naming this `:user_id` ?")
- Ask for clarification. ("I didn't understand. Can you clarify?")
- Avoid selective ownership of code. ("mine", "not mine", "yours")
- Avoid using terms that could be seen as referring to personal traits. ("dumb", "stupid"). Assume everyone is attractive, intelligent, and well-meaning.
- Be explicit. Remember people don't always understand your intentions online.
- Be humble. ("I'm not sure - let's look it up.")
- Don't use hyperbole. ("always", "never", "endlessly", "nothing")
- Be careful about the use of sarcasm. Everything we do is public; what seems like good-natured ribbing to you and a long-time colleague might come off as mean and unwelcoming to a person new to the project.
- Consider one-on-one chats or video calls if there are too many "I didn't understand" or "Alternative solution:" comments. Post a follow-up comment summarizing one-on-one discussion.

# Mind the code of conduct



<https://www.contributor-covenant.org/>





# Code style improves readability

## Major references

- ▶ [Google](#) coding style
- ▶ [Mozilla](#) Coding Style

## Code style for projects:

- ▶ [Android open-source project](#)  
(Good source for Java developers)
- ▶ Code style for [Chromium open source](#) (after [Google C++ style](#))

## Java

- ▶ Original [conventions for Java](#)

## Linux

- ▶ Kernel coding style ([txt](#))

### ASCIIVille

<http://www.asciiville.com>  
Copyright 2008. Todd Presta. All rights reserved.



The dark side of pair programming.



# References

- ▶ Cohen. 2012. Best Kept Secrets of Code Review, SmartBear contributed book .
- ▶ Bloch, Joshua. 2008. Effective Java. 2nd ed. Addison-Wesley Professional. <http://books.google.pt/books?id=ka2VUBqHiWkC>.
- ▶ Fowler, Martin. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. <http://books.google.com/books?id=1MsETFPD3I0C&pgis=1>
- ▶ Martin, Robert C. 2008. Clean Code: A Handbook of Agile Software Craftsmanship (Google eBook). Pearson Education. <http://books.google.com/books?id=i6bDeoCQzsC&pgis=1>.
- ▶ R. Pressman, "Software Engineering: A Practitioner's Approach," Jan. 2009.

