Week 10

Reviews, Best Programming Practices and Tools

Introduction

For many years, most of us in the programming community worked under the assumptions that programs are written solely for machine execution, and are not intended for people to read, and that the only way to test a program is to execute it on a machine. This attitude began to change in the early 1970s through the efforts of program developers who first saw the value in reading code as part of a comprehensive testing and debugging regimen.

Today, not all testers of software applications read code, but the concept of studying program code as part of a testing effort certainly is widely accepted. Several factors may affect the likelihood that a given testing and debugging effort will include people actually reading program code: the size or complexity of the application, the size of the development team, the timeline for application development (whether the schedule is relaxed or intense, for example), and, of course, the background and culture of the programming team.

For these reasons, we will discuss the process of noncomputer-based testing (''human testing''). Human testing techniques are quite effective in finding errors—so much so that every programming project should use one or more of these techniques. You can either apply these methods between the time the program is coded and when computer- based testing begins or at earlier stages in the programming process (such as at the end of each design stage).

Before we begin the discussion of human testing techniques, take note of this important point: Because the involvement of humans results in less formal methods than mathematical proofs conducted by a computer, you may feel sceptical that something so simple and informal can be useful. Just the opposite is true. These informal techniques do not get in the way of successful testing; rather, they contribute substantially to productivity and reliability in two major ways.

First, it is generally recognized that the earlier errors are found, the lower the costs of correcting the errors and the higher the probability of correcting them correctly. Second, programmers seem to experience a psychological shift when computer-based testing commences. Internally induced pressures seem to build rapidly and there is a tendency to want to ''fix this darn bug as soon as possible.'' Because of these pressures, programmers tend to make more mistakes when correcting an error found during computer-based testing than they make when correcting an error found earlier. The three primary human testing methods are code inspections, walkthroughs, and user (or usability) testing. We cover the first two of these, which are code-oriented methods, in this chapter. These methods can be used at virtually any stage of software development, after an application is deemed to be complete or as each module or unit is complete.

Inspections and Walkthroughs

Inspections and walkthroughs involve a team of people reading or visually inspecting a program. With either method, participants must conduct some preparatory work. The climax is a ‘‘meeting of the minds,’’ at a participant conference. The objective of the meeting is to find errors but not to find solutions to the errors - that is, to test, not debug.

In a walkthrough, a group of developers - with three or four being an optimal number - performs the review. Only one of the participants is the author of the program. Therefore, the majority of program testing is conducted by people other than the author, which follows testing principle which states that an individual is usually ineffective in testing his or her own program.

An inspection or walkthrough is an improvement over the older desk - checking process (whereby a programmer reads his or her own program before testing it). Inspections and walkthroughs are more effective, again because people other than the program’s author are involved in the process.

Another advantage of walkthroughs, resulting in lower debugging (error-correction) costs, is the fact that when an error is found it usually is located precisely in the code as opposed to black box testing where you only receive an unexpected result. Moreover, this process frequently exposes a batch of errors, allowing the errors to be corrected later in a group together. Computer-based testing, on the other hand, normally exposes only a symptom of the error (e.g., the program does not terminate, or the program prints a meaningless result), and errors are usually detected and corrected one by one.

These human testing methods generally are effective in finding from 30 to 70 percent of the logic-design and coding errors in typical programs. They are not effective, however, in detecting high-level design errors, such as errors made in the requirements analysis process. Of course, a possible criticism of these statistics is that the human processes find only the ‘‘easy’’ errors (those that would be trivial to find with computer-based testing) and that the difficult, obscure, or tricky errors can be found only by computer-based testing. However, some testers using these techniques have found that the human processes tend to be more effective than the computer-based testing processes in finding certain types of errors, while the opposite is true for other types of errors (e.g., uninitialized variables versus divide by zero errors). The implication is that inspections/walkthroughs and computer-based testing are complementary; error-detection efficiency will suffer if one or the other is not present.

Finally, although these processes are invaluable for testing new programs, they are of equal, or even higher, value in testing modifications to programs. In my experience, modifying an existing program is a process that is more error prone (in terms of errors per statement written) than writing a new programme. Therefore, programme modifications also should be subjected to these testing processes.

- Inspection: It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria [1].
- Walkthrough: It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios [2].

Code Reviews

Code review is a software quality assurance activity in which one or several people check a program mainly by viewing and reading parts of its source code (as discussed above), and they do so after implementation or as an interruption of implementation. The persons performing the checking, excluding the author, are called "reviewers".

Although direct discovery of quality problems is often the main goal, code reviews are usually performed to reach a combination of goals. Some of these are as under:

Better code quality: Improving the internal code quality following different quality attributes (some of which we mentioned in week 1 notes) and keeping code maintainable for future deliverables and extensions.

Finding defects: That is the purpose of conducting this activity. Mainly, it is about improving the code quality regarding external aspects, especially correctness, but also finding performance problems, security vulnerabilities, injected malware etc.

Learning/Knowledge transfer: help in transferring knowledge about the codebase, solution approaches, expectations regarding quality, etc; both to the reviewers as well as to the author and the maintenance teams.

Increase sense of mutual responsibility: It creates a sense of collective ownership, one of the main ingredients of modern open source projects and reviews.

Finding better solutions: Code review gives an opportunity to understand code thoroughly and this generates ideas for new and better solutions and ideas that transcend the specific code at hand collectively.

Complying to QA guidelines: Different companies and institutes have a QA standard guideline, so a code review activity helps in finding if the standards are followed. Also, it is critical to follow standards in certain contexts - where a defect can cause millions of dollars of loss such as air traffic software. Code reviews are mandatory in some contexts, e.g., air traffic software.

Are Code Reviews Peer Reviews?

In software development, peer review is a type of software review in which a work product (document, code, or other) is examined by one or more colleagues (who is an expert in the specified field), in order to evaluate its technical content and quality. In CMM peer review is defined as a disciplined engineering practice for detecting and correcting defects in software artefacts

Peer Code Review (Pair programming) is one of the integral parts of eXtreme Programming and mentioned by Kent Beck. Kent Beck describes pair programming as involving "two people looking at one machine, with one keyboard and one mouse". The person at the keyboard concentrates on the here and now, while their partner is reviewing both the code and thinking strategically about how the code fits in with the larger system. According to Beck, pairing should happen dynamically within the team, rather than being a long-term partnership between two programmers.

Roles in Code Review

All the people involved in the review process are informed of the group review meeting schedule two or three days before the meeting. They are also given a copy of the work package for their perusal. Reviews are conducted in bursts of 1 - 2 hours. Longer meetings are less and less productive because of the limited attention span of human beings. The rate of code review is restricted to about 125 lines of code (in a high-level language) per hour. Reviewing complex code at a higher rate will result in just glossing over the code, thereby defeating the fundamental purpose of code review. The composition of the review group involves a number of people with different roles. These roles are explained as follows:

- **Moderator**: A review meeting is chaired by the moderator. The moderator is a trained individual who guides the pace of the review process. The moderator selects the reviewers and schedules the review meetings. Myers suggests that the moderator be a member of a group from an unrelated project to preserve objectivity [4].
- **Author**: This is the person who has written the code to be reviewed.
- **Presenter**: A presenter is someone other than the author of the code. The presenter reads the code beforehand to understand it. It is the presenter who presents the author's code in the review meeting for the following reasons: (i) an additional software developer will understand the work within the software organization; (ii) if the original programmer leaves the company with a short notice, at least one other programmer in the company knows what is being done; and (iii) the original programmer will have a good feeling about his or her work, if someone else appreciates their work. Usually, the presenter appreciates the author's work.
- **Recordkeeper**: The recordkeeper documents the problems found during the review process and the follow-up actions suggested. The person should be different than the author and the moderator.
- **Reviewers**: These are experts in the subject area of the code under review. The group size depends on the content of the material under review. As a rule of thumb, the group size is between 3 and 7. Usually this group does not have manager to whom the author reports. This is because it is the author's ongoing work that is under review, and neither a completed work nor the author himself is being reviewed.
- **Observers**: These are people who want to learn about the code under review. These people do not participate in the review process but are simply passive observers.

How Code Reviews are Conducted?

Regardless of whether a review is called an inspection or a walkthrough, it is a systematic approach to examining source code in detail. The goal of such an exercise is to assess the quality of the software in question, not the quality of the process used to develop the product [3]. Reviews of this type are characterized by significant preparation by groups of designers and programmers with varying degree of interest in the software development project. Code examination can be time consuming. Moreover, no examination process is perfect. Examiners may take shortcuts, may not have adequate understanding of the product, and may accept a product which should not be accepted. Nonetheless, a well-designed code review process can find faults that may be missed by execution-based testing. The key to the success of code review is to divide and conquer, that is, having an examiner inspect small parts of the unit in isolation, while making sure of the following:

i. nothing is overlooked and
ii. the correctness of all examined parts of the module implies the correctness of the whole module.

The decomposition of the review into discrete steps must assure that each step is simple enough that it can be carried out without detailed knowledge of the others.

The objective of code review is to review the code, not to evaluate the author of the code. A clash may occur between the author of the code and the reviewers, and this may make the meetings unproductive. Therefore, code review must be planned and managed in a professional manner. There is a need for mutual respect, openness, trust, and sharing of expertise in the group. The general guidelines for performing code review consists of six steps as outlined in Figure 3.1: readiness, preparation, examination, rework, validation, and exit. The input to the readiness step is the criteria that must be satisfied before the start of the code review process, and the process produces two types of documents, a change request (CR) and a report. These steps and documents are explained in the following.

Step 1: Readiness

The author of the unit ensures that the unit under test is ready for review. A unit is said to be ready if it satisfies the following criteria:

- **Completeness**: All the code relating to the unit to be reviewed must be available. This is because the reviewers are going to read the code and try to understand it. It is unproductive to review partially written code or code that is going to be significantly modified by the programmer.
- **Minimal Functionality**: The code must compile and link. Moreover, the code must have been tested to some extent to make sure that it performs its basic functionalities.
- **Readability**: Since code review involves actual reading of code by other programmers, it is essential that the code is highly readable. Some code characteristics that enhance readability are proper formatting, using meaningful identifier names, straightforward use of programming language constructs, and an appropriate level of abstraction using function calls. In the absence of readability, the reviewers are likely to be discouraged from performing the task effectively.
- **Complexity**: There is no need to schedule a group meeting to review straightforward code which can be easily reviewed by the programmer. The code to be reviewed must be of sufficient complexity to warrant group review. Here, complexity is a composite term referring to the number of conditional statements in the code, the number of input data elements of the unit, the number of output data elements produced by the unit, real-time processing of the code, and the number of other units with which the code communicates.

- **Requirements and Design Documents**: The latest approved version of the low-level design specification or other appropriate descriptions of program requirements should be available. These documents help the reviewers in verifying whether or not the code under review implements the expected functionalities. If the low-level design document is available, it helps the reviewers in assessing whether or not the code appropriately implements the design.

Step 2: Preparation

Before the meeting, each reviewer carefully reviews the work package. It is expected that the reviewers read the code and under- stand its organization and operation before the review meeting. Each reviewer develops the following:

- **List of Questions:** A reviewer prepares a list of questions to be asked, if needed, of the author to clarify issues arising from his or her reading. A general guideline of what to examine while reading the code is outlined in….
- **Potential CR:** A reviewer may make a formal request to make a change. These are called change requests rather than defect reports. At this stage, since the programmer has not yet made the code public, it is more appropriate to make suggestions to the author to make changes, rather than report a defect. Though CRs focus on defects in the code, these reports are not included in defect statistics related to the product.
- **Suggested Improvement Opportunities:** The reviewers may suggest how to fix the problems, if there are any, in the code under review. Since reviewers are experts in the subject area of the code, it is not unusual for them to make suggestions for improvements.

Step 3: Examination

The examination process consists of the following activities:

- The author makes a presentation of the procedural logic used in the code, the paths denoting major computations, and the dependency of the unit under review on other units.
- The presenter reads the code line by line. The reviewers may raise questions if the code is seen to have defects. However, problems are not resolved in the meeting. The reviewers may make general suggestions on how to fix the defects, but it is up to the author of the code to take corrective measures after the meeting ends.
- The recordkeeper documents the change requests and the suggestions for fixing the problems, if there are any. A CR includes the following details:
    - Give a brief description of the issue or action item.
    - Assign a priority level (major or minor) to a CR.
    - Assign a person to follow up the issue. Since a CR documents a potential problem, there is a need for interaction between the author of the code and one of the reviewers, possibly the reviewer who made the CR.
    - Set a deadline for addressing a CR.
- The moderator ensures that the meeting remains focused on the review process. The moderator makes sure that the meeting makes progress at a certain rate so that the objective of the meeting is achieved.
- At the end of the meeting, a decision is taken regarding whether or not to call another meeting to further review the code. If the review process leads to extensive rework of the code or

critical issues are identified in the process, then another meeting is generally convened. Otherwise, a second meeting is not scheduled, and the author is given the responsibility of fixing the CRs.

Step 4: Rework

At the end of the meeting, the recordkeeper produces a summary of the meeting that includes the following information:

- A list of all the CRs, the dates by which those will be fixed, and the names of the persons responsible for validating the CRs
- A list of improvement opportunities
- The minutes of the meeting (optional)

A copy of the report is distributed to all the members of the review group. After the meeting, the author works on the CRs to fix the problems. The author documents the improvements made to the code in the CRs. The author makes an attempt to address the issues within the agreed-upon time frame using the prevailing coding conventions [5].

Step 5: Validation

The CRs are independently validated by the moderator or another person designated for this purpose. The validation process involves checking the modified code as documented in the CRs and ensuring that the suggested improvements have been implemented correctly. The revised and final version of the outcome of the review meeting is distributed to all the group members.

Step 6: Exit

Summarizing the review process, it is said to be complete if all of the following actions have been taken:

- Every line of code in the unit has been inspected.
- If too many defects are found in a module, the module is once again reviewed after corrections are applied by the author. As a rule of thumb, if more than 5% of the total lines of code are thought to be contentious, then a second review is scheduled.
- The author and the reviewers reach a consensus that when corrections have been applied the code will be potentially free of defects.
- All the CRs are documented and validated by the moderator or someone else. The author's follow-up actions are documented.
- A summary report of the meeting including the CRs is distributed to all the members of the review group.

The effectiveness of human reviews is limited by the ability of a reviewer to find defects in code by visual means. However, if occurrences of defects depend on some actual values of variables, then it is a difficult task to identify those defects by visual means. Therefore, a unit must be executed to observe its behaviours in response to a variety of inputs. Finally, whatever may be the effectiveness of static tests, one cannot feel confident without actually running the code.

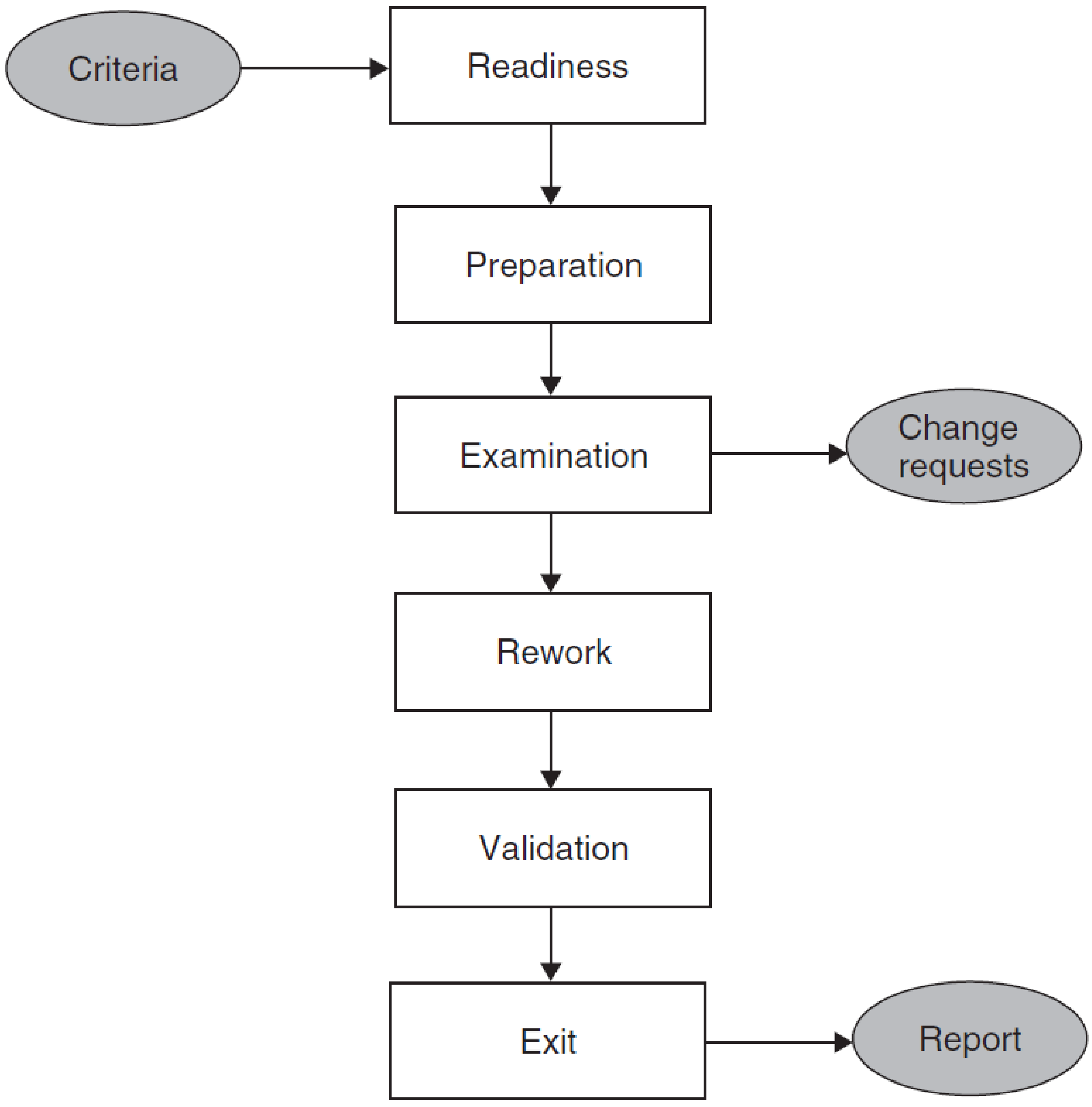

Figure 1: Code Review Steps.

A Sample Code Review Checklist

1. Does the code do what has been specified in the design specification?
2. Does the procedure used in the module solve the problem correctly?
3. Does a software module duplicate another existing module which could be reused?
4. If library modules are being used, are the right libraries and the right versions of the libraries being used?
5. Does each module have a single-entry point and a single exit point? Multiple exit and entry point programs are harder to test.
6. Is the cyclomatic complexity of the module more than 10? If yes, then it is extremely difficult to adequately test the module.
7. Can each atomic function be reviewed and understood in 10 - 15 minutes? If not, it is considered to be too complex.

8. Have naming conventions been followed for all identifiers, such as pointers, indices, variables, arrays, and constants? It is important to adhere to coding standards to ease the introduction of a new contributor (programmer) to the development of a system.
9. Has the code been adequately commented upon?
10. Have all the variables and constants been correctly initialized? Have correct types and scopes been checked?
11. Are the global or shared variables, if there are any, carefully controlled?
12. Are there data values hard coded in the program? Rather, these should be declared as variables.
13. Are the pointers being used correctly?
14. Are the dynamically acquired memory blocks deallocated after use?
15. Does the module terminate abnormally? Will the module eventually terminate?
16. Is there a possibility of an infinite loop, a loop that never executes, or a loop with a premature exit?
17. Have all the files been opened for use and closed at termination?
18. Are there computations using variables with inconsistent data types? Is overflow or underflow a possibility?
19. Are error codes and condition messages produced by accessing a common table of messages? Each error code should have a meaning, and all of the meanings should be available at one place in a table rather than scattered all over the program code.
20. Is the code portable? The source code is likely to execute on multiple processor architectures and on different operating systems over its lifetime. It must be implemented in a manner that does not preclude this kind of a variety of execution environments.
21. Is the code efficient? In general, clarity, readability, or correctness should not be sacrificed for efficiency. Code review is intended to detect implementation choices that have adverse effects on system performance.

This is just a sample list of code review items. The checklist items may vary based on the code, language, specifications and the environment.

WTFPM Code Quality Measure

Cars have MPH (Miles per Hour) or KPH (Kilometre per hour) that measures the speed that they travel. The better the car the faster the MPH or KPH. Developers have WTFPM (WTF per Minute) that measures the number of ‘Works That Frustrate’ that the developer can read per minute, aka code quality. And just like cars, the better the developer the more WTFPM they can attain. Here is where they differ, however. With a car, the BETTER the road the more MPH or KPH it can attain. With a developer, the WORSE the code the more WTFPM can be obtained.

Code Reviews allow for a controlled WTFPM increase while also addressing the offending code. Due to the small amount of code being reviewed at any given time the WTFPM should be relatively small and controllable.

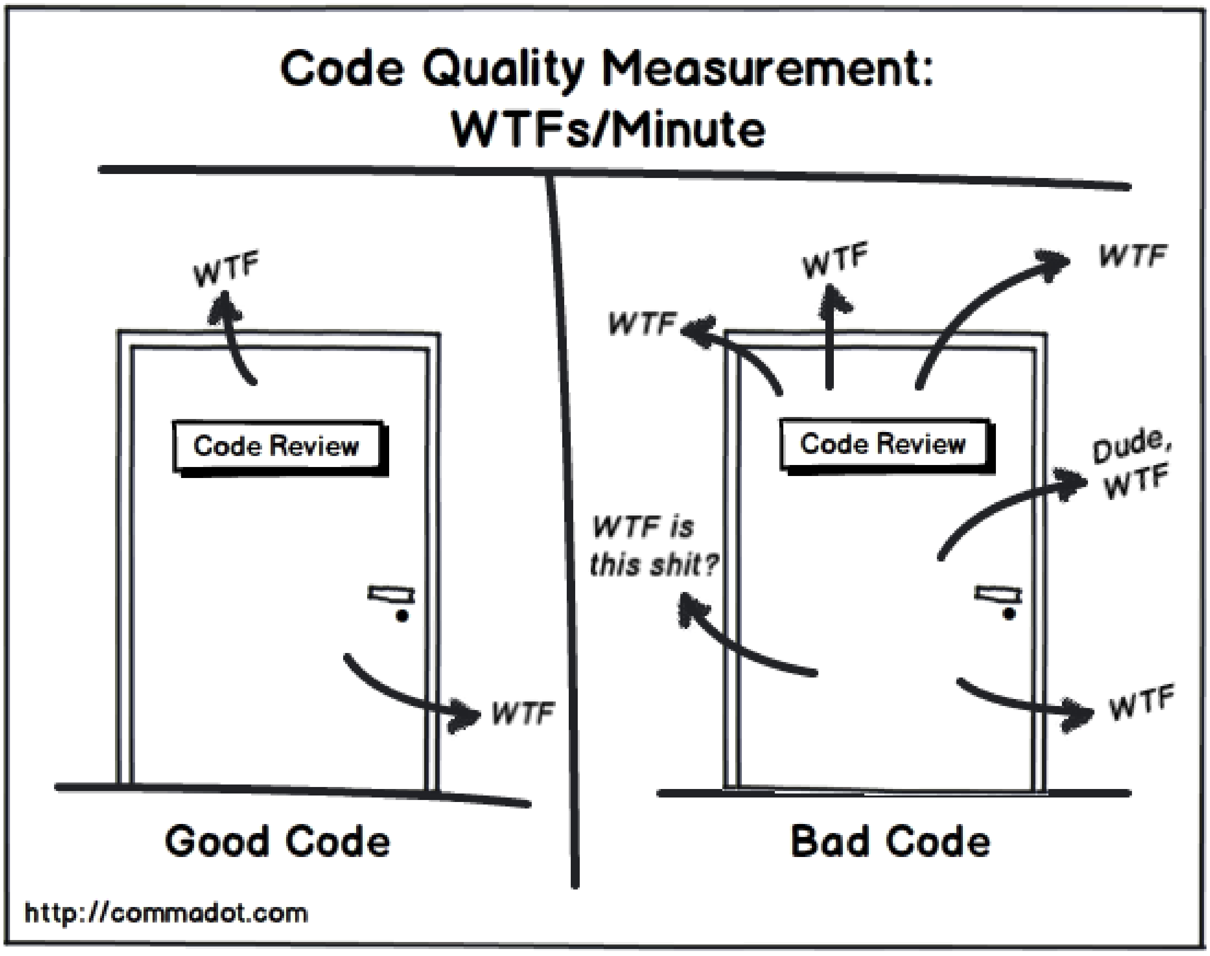

Figure 2: The only valid measurement of code quality is WTF/minutes

Modern Code Review

Software code review is a well-established software quality practice. Recently, Modern Code Review (MCR) has been widely adopted in both open source and proprietary projects. MCR is a lightweight process involving fewer formal requirements, a shorter time to a finished review, and often review tool support. Many tools such as Git, Gerrit etc are used these days in industry to quality check the code as the majority of the development is continuous.

From a black box perspective on Figure 3, a review is a process that takes as input original source code (i.e., the first unreviewed change attempt, Step 1), and outputs accepted source code (2). The author is the person responsible for the implementation of the assigned task as source code. The reviewer(s) assure the implementation meets the quality standards. The original source code is a work that stemmed solely from the author, whereas in the accepted
source code the author incorporated the reviewers' suggestions so that everybody is satisfied with the result. The grey area in Figure 3 reveals the inner workings of the review process from a white box perspective: Once source code is submitted for review; the reviewers decide whether they accept it (3) or not (4). Their decision is normally based on the project's quality acceptance criteria, reviewing checklists, and guidelines. If they do not accept the code, reviewers annotate it with their suggestions (4) and send the reviewed source code back to the author. Addressing the reviewers' suggestions, the author makes alterations to the code and sends it back for further review (5). A review round

comprises the two steps 'source code submitted for review' (1 or 5) and its actual, technical review (3 or 4). Therefore, a piece of code can minimally have one review round, if (3) is executed directly, or potentially infinitely many rounds, if the reviewers are never satisfied.

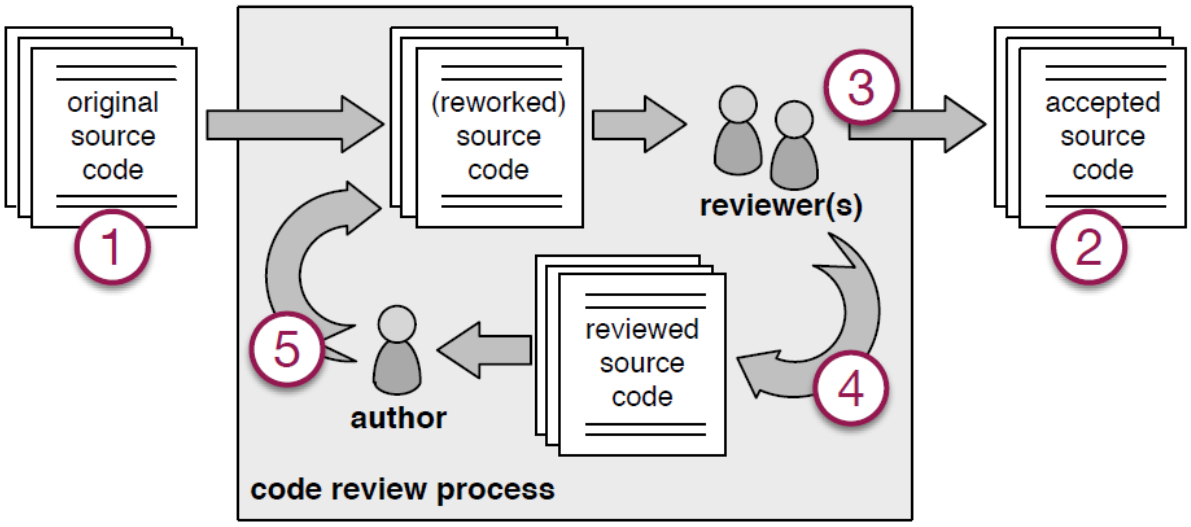

Figure 3: The review process.

How Google uses code review is explained in the research paper uploaded in the reading section of week 10 on Moodle. Do read.

Code Review Guidelines for the Authors

As an author of the code or work being reviewed, you will need to be able to be open-minded and be receptive to all types of recommendations and feedback, including negative ones. Most importantly, you should set your own expectations to an appropriate level, so that you can get something positive out of the review process.

- Be humble
  - Nobody is perfect. Mistakes and bugs will eventually happen even if you pay your utmost care and attention when developing code or other deliverables.
  - If a mistake happens, accept it and use it as a learning opportunity to make yourself better as an IT professional, or even as a person.
  - There are always many ways to do something and all of them could be perfectly valid. The code you write might be the best to you but be mindful that others may feel the total opposite.
  - The process of learning and self-improvement should never stop, even if you are the best at what you do. You should treat the review as an opportunity to exchange knowledge, as well as to upskill yourself and stay up to date with the best practices currently utilised in the industry.

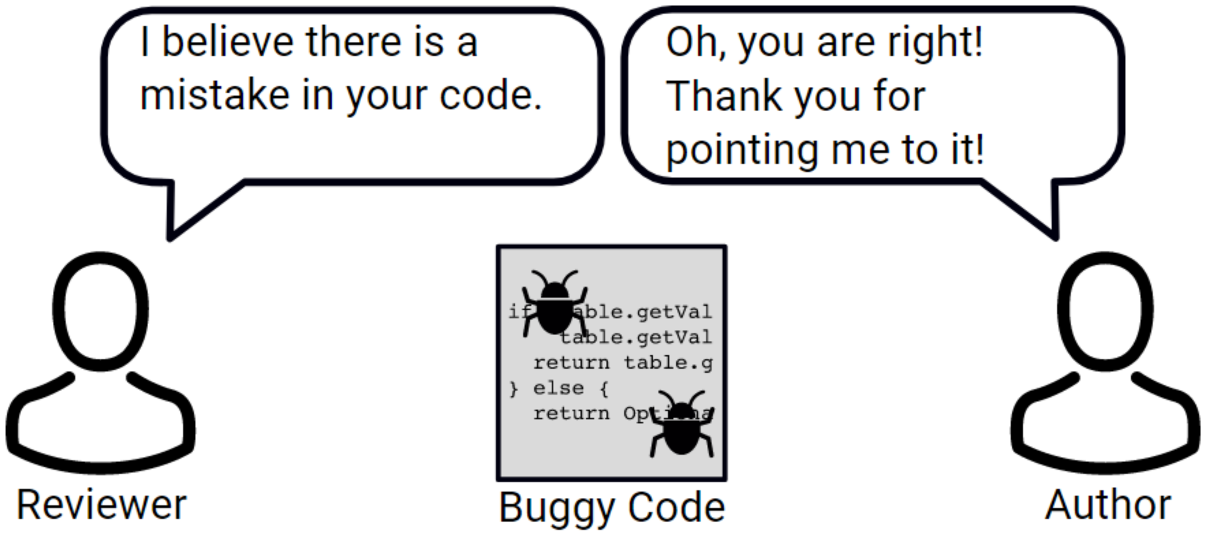
I believe there is a mistake in your code.
Oh, you are right! Thank you for pointing me to it!
Reviewer
Buggy Code
Author

Figure 4: Making mistakes is accepted and admitting them is desired.

- Your code does not represent you as a person
    - Flaws in your code are not translatable to what, how, or who you are as a person.
    - Feedback about your code is essentially just that – it is not about you! Comments and recommendations about your code should never be treated as a personal attack.
    - If you receive poor feedback regarding the code you wrote, you should not lose confidence in yourself or your abilities.

- The reviewer and you are friends
    - Usually, the reviewer will also be a stakeholder in the project you are developing the code or deliverable for.
    - Hence, there is no reason for the reviewer to make your job difficult purposely and unnecessarily as this also impacts them negatively!
    - Reviewers are out there to help you achieve the common goal that the both of you (and the entire team) share, which is to deliver good quality software which meets the project's budget, timeframe and requirements.

- Beware of cognitive biases (IKEA effect)
    - When you write a piece of code, you will usually value your own code more highly compared to equivalent code written by someone else - this is known as the IKEA effect, and have some degree of emotional attachment to it. This makes it harder for you to accept criticism or suggestions on how you could improve your code.
    - It might be harder for us to accept changes or removal of code that we have created. It's important to be aware of this bias when we receive feedback because we might be influenced by the IKEA effect.

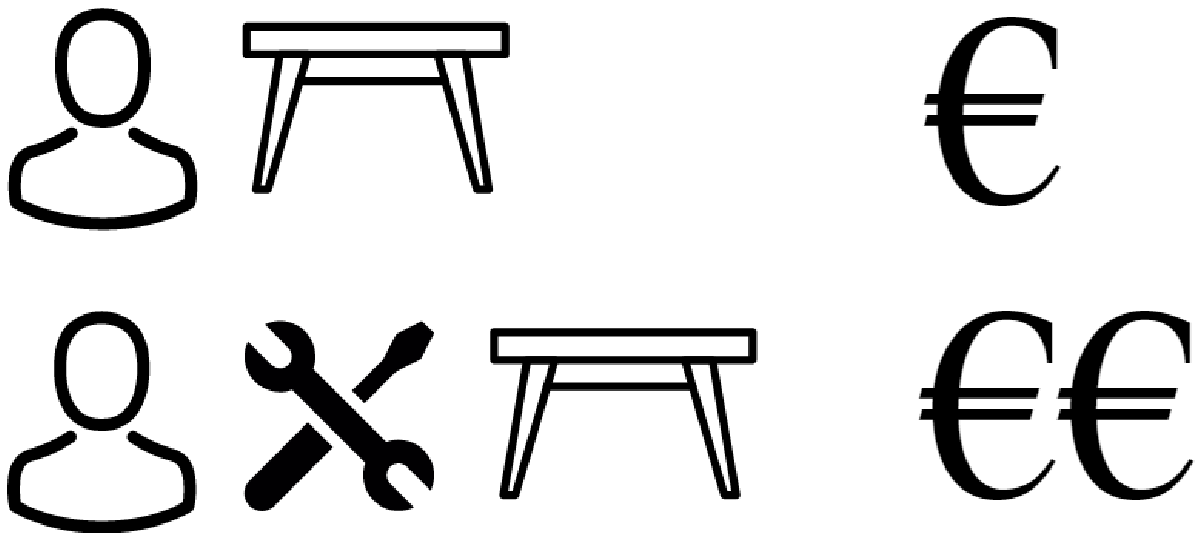
Figure 5: The IKEA effect let us place more value on things (furniture, code) we have created by ourself.

- Bandwagon Effect
  - Try to independently assess each of the suggestions and feedback given to you by the reviewer – do not fall into the trap of the bandwagon effect/groupthink by blindly opposing or agreeing to everything that was raised during the review without discussing them further with the reviewer.
- Be open to other perspectives
  - Everyone sees things in a different way, as not everyone has the same skill levels, experiences, or thought processes as you. Other people might know a few tips and tricks which they might dismiss as being trivial or common knowledge, but it might be something that you have never heard of and could also be the last piece of the puzzle you are missing for a long-standing problem you are facing.
  - The code you have written will usually be clear to you as its author (although not forever!), but it is not always the case that it will be immediately obvious to others. It is perfectly normal if you cannot see any problems with your code – this is the exact reason why we have code reviews!
  - This is also why you need to be explicit and intentional in what you are trying to achieve with your code – nobody else in this world has the same viewpoint and perspective as you. The more barriers to understanding you remove, the less opportunities there are for bugs to be introduced or something getting missed out.

Code Review Guidelines for the Reviewer

When you take on the role of a reviewer, you need to be mindful of the way you articulate and present your feedback to the person whose code or work you are reviewing. Good and constructive feedback is essential for a productive review process.

- Focus on reviewing only code, not the author or their personal traits
    - The purpose of a code review, like what its name implies, is to objectively review the code and not subjectively review the coder!
    - Instead of saying
        - *"You have written code that is quite poor and unmaintainable"*, say something like
        - *"The code here seems to be unmaintainable and could have been written in a better way"*.
    - If a personal aspect or behaviour of the author is pulling down the quality of their work and you need to address it, refer to their behaviour or actions instead. Never ever resort to personal (ad hominem) attacks. Instead of saying
        - *"This code would not have any problems if it was written by a more intelligent person"*, say this instead:
        - *"I think you should look at getting more coding practice in your free time, so that you can write better code*".
    - Most of the time, it is not necessary to talk about the author at all in a review!
    - This way, you will avoid making the author become resistant and defensive towards you, and potentially turn the review into a back and forth shouting match which does not benefit anyone.
    - A key reason why you should only focus on the code and not the coder is that parts of the code being reviewed might have been written by someone else before the author took over responsibility of the code.

- Express your feedback as I-messages
    - Instead of saying
        - "You have written something that is not understandable", try saying
        - "I have tried reading your code, but I am still unable to understand it".
    - By shifting the focus to what you think or feel about the piece of work under review, it encourages discussion as you are giving an opinion and not a definitive statement.
    - Most importantly, you will more easily avoid arguing and accidentally making personal attacks, and the author does feel a need to keep justifying themselves.

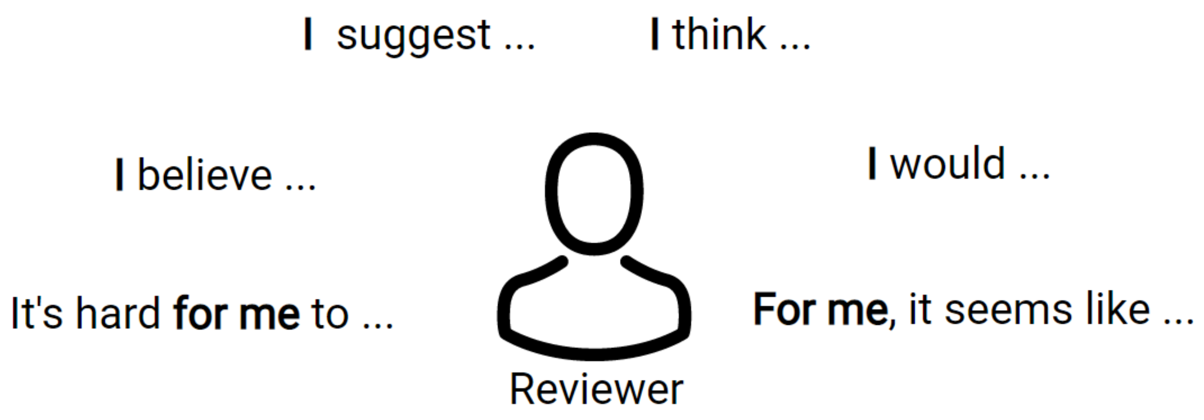

Figure 6: Increase the acceptance of your feedback by using I-messages.

- Prefer asking questions instead of making statements
  - When you ask questions as a reviewer, it provides you with an opportunity to clarify your understanding and remove any existing doubts, as well as initiating a thought process in the author's mind.
  - This can lead to a more effective review as authors will often discover mistakes that they could not have found on their own, and they might potentially come up with much better solutions on the spot.
  - Again, asking questions means that you do not issue definitive statements or remarks which the author might feel uncomfortable with.
  - For example, say
    - "*Can you tell me why this class is named as Customer and not User?*", instead of
    - "*This class should have been named as User*".

- Give structured, effective and feedback
  - You need to make sure that the feedback that you provide is constructive and allows the recipient to learn from their mistakes.
  - Often, it is not enough to point out a mistake and stop there. You should also suggest actionable ways to improve or avoid those mistakes in the future.
  - A good way to make constructive feedback is to follow the **OIR framework (Observation, Impact, Request)** and make sure you use I-messages.
    - *Observation*

      "This class has a lot of attributes with obscure names such as 'x', 'y' and 'z' ..."

    - *Impact*

      "... which resulted in **me** not being able to understand why we need this class in the system's design."

        - *Request*

          “**I** would like to suggest renaming them to something more explicit, such as ‘x’ as ‘user_count’ and ‘y’ as ‘user_id’ instead.”

    - Each time you give feedback, ask yourself whether that piece of feedback is true, necessary and kind.
        - *True*
            - Make sure you avoid making a definitive statement for something where there is no right or wrong answer. That is, avoid saying that something is correct, wrong, or dictate that the author “should” do something.
            - Be mindful that an opinion is not a fact.
        - *Necessary*
            - Do not criticise and nitpick every small detail such as minor formatting issues, insignificant typos in comments etc.
            - Focus on things which are more important such as functional and algorithmic correctness, code smells, etc. instead.
            - Constantly making unnecessary comments will irritate the author and make them less receptive to your feedback.
        - *Kind*
            - Do not shame, belittle, insult, patronise, or put someone down in your feedback. If you do not have anything nice to say, don’t say it.
            - Always ask yourself how you would feel if someone else gave you the exact same comment you are planning to give out.

- Be aware that multiple valid solutions can exist
    - As mentioned in the guidelines for authors, acknowledging the fact that there is no single right way to implement a solution to something is also applicable to reviewers.
    - Always be objective and be ready to compromise. Do not impose your personal preferences onto somebody else. If the requirements are being met, it should not be a problem at all if the author is using a package which you dislike solely due to personal reasons and interests.
    - There is a big difference between following good programming practices and following personal preferences.

- Don’t Jump in Front of Every Train
    - Don’t be a pedant.
    - Don’t criticize every single line of code. Again, this would annoy the author and reduce their openness to further feedback and harm your relationship. And if

your interpersonal relationship is destroyed, you have a much bigger problem than some not perfectly named variables. Instead, choose wisely the battles you are going to fight.

- Focus on the flaws and code smells that are most important to you.

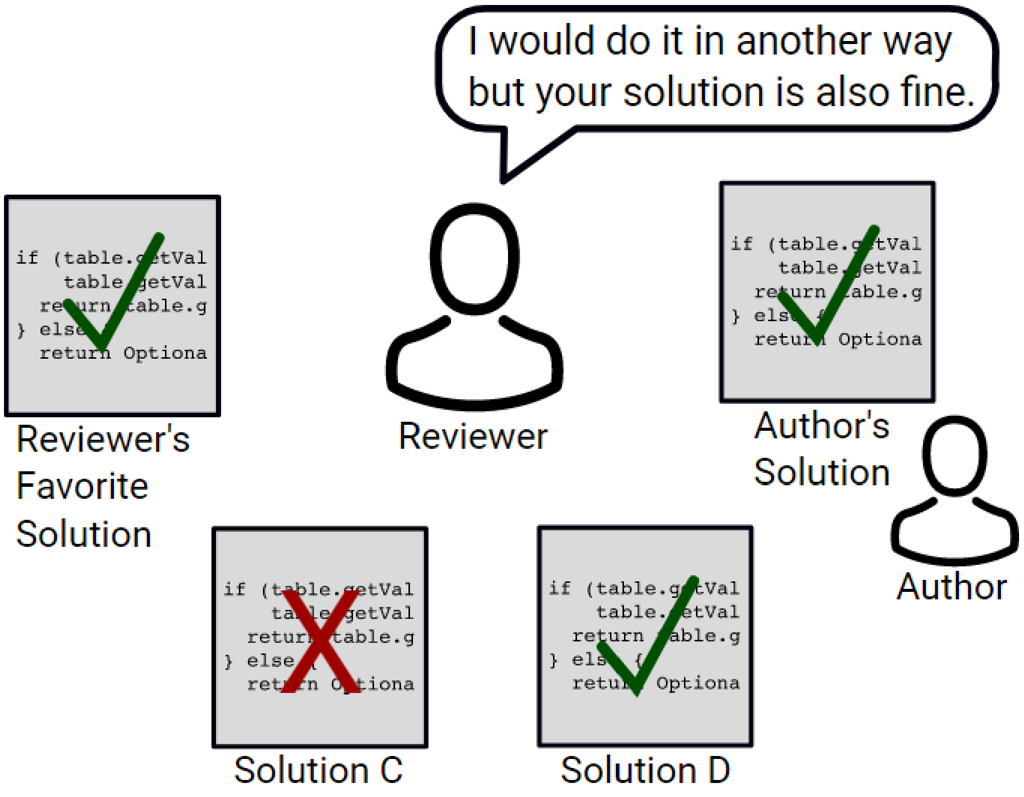

Figure 7: Be aware of your personal taste, accept other solutions and make compromises.

- Give praise when it is deserved
  - When the code you have reviewed is great and the author has done a good job, be sure to make it known to them. Make sure you elaborate on why and how something was done well, too.
  - Giving praise is free and motivates people to do an even better job, which results in an increasingly positive environment for everyone moving forward.
  - Use different sentences and avoid sandwiching
  - It is perfectly valid to have a review where the outcome is that no changes to the code are required – you do not have to find a fault with the code when there isn't one.
  - Always start with the positives before you talk about the downsides. Praise should not be combined with criticism in the same sentence, so that the user can clearly distinguish between praise and criticism. This minimises the chance of misinterpretation occurring.

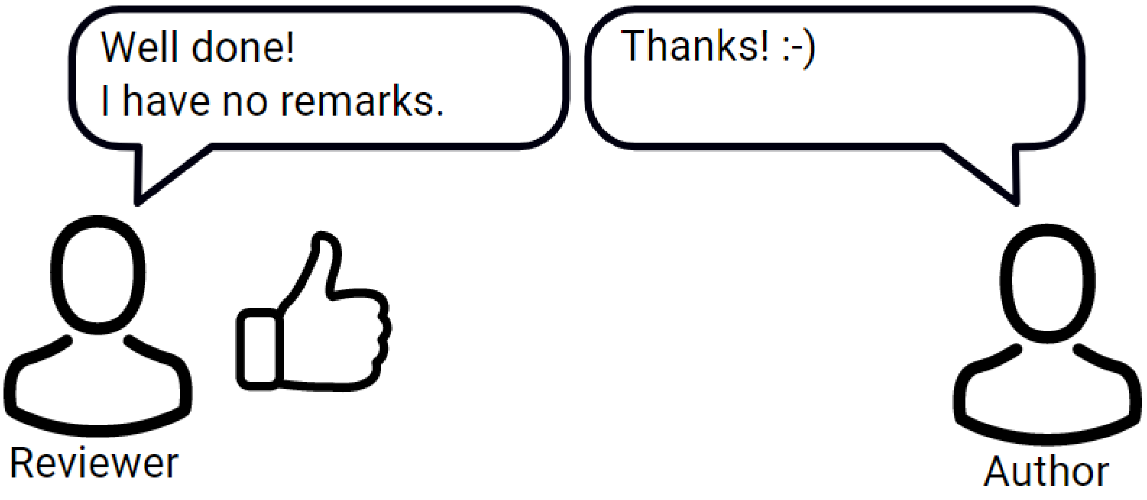

Figure 8: Don't forget to praise.

Good Programming Practices

An old saying:

> *"The day you write your code, you and God understand how it works. Six months later, only God does."*

Code is often worked on by many people (in teams) and has a very long-life span. Therefore, it should be written in a way that everyone in the team as well as those who are going to use it in future understands it thoroughly. So how to make life easier for these people? We follow certain practices that are industry norms now and must be followed.

Practice 1: Meaningful names/ naming conventions

Use of proper naming conventions is considered good practice. Sometimes programmers tend to use X1, Y1, etc. as variables and forget to replace them with meaningful ones, causing confusion. It is usually considered good practice to use descriptive names.

Example: A variable for taking in weight as a parameter for a truck can be named `TrkWeight` or `TruckWeightKilograms`, with `TruckWeightKilograms` being the preferable one, since it is instantly recognisable.

Naming conventions make programs more understandable by making them easier to read. Python has a PEP8 style guide[1] whereas java uses its own conventions[2], which are different from Python. I recommend you all to follow the proper conventions or style guide for the language you are programming in.

[1] https://www.python.org/dev/peps/pep-0008/
[2] https://google.github.io/styleguide/javaguide.html

Practice 2: Commenting and Documentation

All classes, methods etc should be properly commented as per the Python language conventions. IDE's (Integrated Development Environment) have come a long way in the past few years and have made commenting your code more useful than ever. Utilise IDE and other tools, for example, Javadocs for documenting code.

Practice 3: Documentation

It is important that the code artefact can be understood easily by the SQA team, maintainer, developers, architects etc therefore documenting the basic idea of what this class is created or what the purpose of this specific application is important.

Documenting the code using comments may including following information:

- Name of the module
- Purpose of the Module
- Description of the Module
- Original Author
- Modifications
- Authors who modified code with a description on why it was modified.

Practice 4: Code Indentation

There is no right or wrong indentation that everyone should follow. The best style is a consistent style. Python is strict about indentation and will not compile if the code is not properly indented. For some languages, such as Java, it is not mandatory to have a code properly indented. You can write a Java code in a single line and compile it. This causes serious readability issues. Therefore, proper indentation between methods, variables etc is really important.

Practice 5: Avoid Code Smells

Martin Fowler defined code smell as:

A code smell is a surface indication that usually corresponds to a deeper problem in the system.

Though there are many types of code smell that may exist in your code, we only discuss some common code smells and their solutions. Code Refactoring is applied to avoid code smells in a code. Martin Folwer has compiled an extensive list of instances, including code smells code smells and their rectification and where the refactoring is important and must be applied can be accessible at https://refactoring.com/catalog/.

Some Example of Code Smells

1. Code Duplication

When the same code appears in multiple places, it must be maintained in multiple places. If a bug is discovered in that code, every piece of duplicate or very similar code must be checked and fixed. If the requirements change and the code needs to be modified, it needs to be tracked down and changed everywhere. Pain?? Indeed, it is.

Apply *DRY Principl*e i.e. *DRY stands for Don't Repeat Yourself*. According to this principle abstract out common code and put it in a single location.

2. Excessive use of Constants

A classic code smell is seeing constants embed in the bodies of methods. This is bad for several reasons. First, if the value of the constant (a.k.a. literal) needs to change in future, you have to hunt for every place it occurs in the code and change it in all of them. There may also be subtle dependencies between pieces of code due to the value of the constant, even though the same value does not appear in both places. These also need to be found and fixed.

3. God Class/ Object

A God class is one that does way too much i.e., it has more than one main responsibility. It violates two important object-oriented design principles that are Separation of Concerns (SOP) and *Single Responsibility Principle (SRP)*. SOP is separating a programme into different sections such that each section addresses a separate concern. Whereas SRP is that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. Therefore, it is important to split the code into smaller classes and methods with each having their own responsibility.

4. Too Many Parameters

We can agree that a function with zero arguments is a good function, we do not have to worry too much about it. The perfect world will be if all functions have zero arguments, but we know that is not possible. Our functions should have the minimum number of arguments possible, if it has less than four argument, nice. Put a limit on the argument's number is not the ideal, we should strive to keep them as minimal as we can.

One of the problems that could emerge in our application if we have too many arguments is that the function is doing too many things, not respecting the *Single Responsibility Principle*. Maybe there are some parameters set that could be related and turned into its own object, using the *Parameter Object pattern*.

There are two techniques that can be used to reduce a functions' arguments. One of them is to refactor the function, making it smaller, consequently, reducing the arguments' number. The *Extract Method* technique can be used to achieve this goal.

Another technique is the pattern mentioned earlier, the Parameter Object pattern. We can aggregate arguments that are within same context, and then create a plain object containing those arguments.

Advantages:

- It makes the code more readable, because probably the functions are smaller and following the single responsibility principle.

- Makes the code easier to test. By making the function smaller, test problems individually are pretty simple. We can test the paths in the main function and have a collection of smaller tests for each individual function.

FIT2099 and FIT3077 are good units to learn code programming practices and object-oriented principles and patterns that are commonly used in industry now.

Code Review Tools

There are many tools available that are used for automating code reviews. I am providing a brief information of these tools.

GitLab: Gitlab has a great functionality to conduct code reviews. Unfortunately, on Git Infotech we cannot conduct code reviews as it is still using the older version of Gitlab.

Gerrit: is a free, web-based team code collaboration tool. Software developers in a team can review each other's modifications on their source code using a Web browser and approve or reject those changes.

Pylint is a Python static code analysis tool which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions. In addition to that PyCharm has a built-in code inspection tool that will automatically detect and correct anomalous code

SonarQube: is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on 20+ programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

Reference

[1] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, July 1976, pp. 182–211; reprinted 1999, pp. 258–287.
[2] E. Yourdon. Structured Walkthroughs. Prentice-Hall, Englewood Cliffs, NJ, 1979.
[3] D. Parnas and M. Lawford. The Role of Inspection in Software Quality Assurance. IEEE Transactions on Software Engineering, August 2003, pp. 674–676.
[4] G. Myers. A Controlled Experimentat in Program Testing and Code Walk-throughs/Inspections.Communications of the ACM, September 1978, pp. 760–768.
[5] H. Sutter and A. Alexandrescu. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley, Reading, MA, 2004.

Other References

- Baum, Tobias; Liskin, Olga; Niklas, Kai; Schneider, Kurt (2016). "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes". 2016 IEEE International

Conference on Software Quality, Reliability and Security (QRS). pp. 74–85. doi:10.1109/QRS.2016.19. ISBN 978-1-5090-4127-5.

- Kolawa, Adam; Huizinga, Dorota (2007). Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press. p. 260. ISBN 978-0-470-04212-0.
- Bacchelli, A; Bird, C (May 2013). "Expectations, outcomes, and challenges of modern code review" (PDF). Proceedings of the 35th IEEE/ACM International Conference On Software Engineering (ICSE 2013). Retrieved 2015-09-02.
- Baum, Tobias; Liskin, Olga; Niklas, Kai; Schneider, Kurt (2016). "Factors Influencing Code Review Processes in Industry". Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. pp. 85–96. doi:10.1145/2950290.2950323. ISBN 9781450342186.
- Berman, Jillian [1], "The IKEA Effect: Study Finds Consumers Over-Value Products They Build Themselves", September 2011
- https://phauer.com/2018/code-review-guidelines/