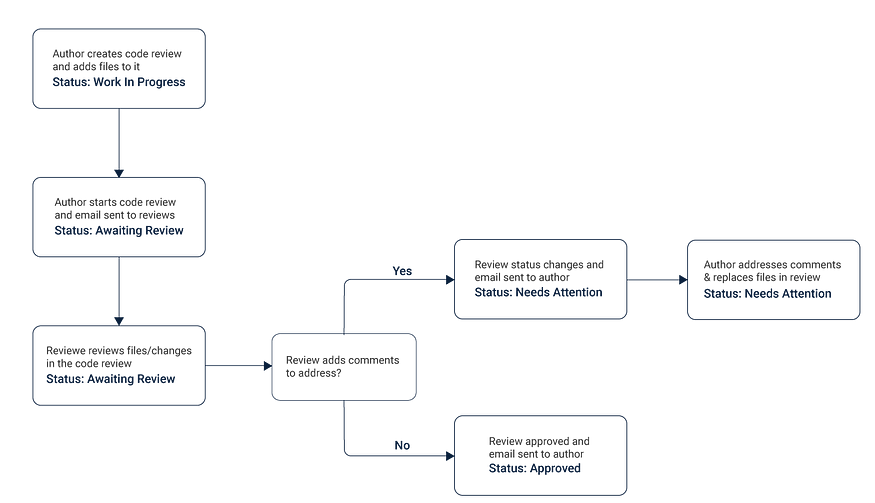
**CODE REVIEW PROCESS EVALUTION**

**VERIFACTS SERVICES PRIVATE LIMITED**

# Introduction:

Systems that work heavily on coding applications are often overwrought with lines and paragraphs of syntactical and contextual language that can be tough to understand to the average pair of eyes.

Even more cumbersome and tiring is the process of debugging and looking for errors in the source code whenever some pesky errors arrive, or the results don’t match the expected output.



As technology undergoes an upheaval in the techniques and ideas used to uphold the tenants of advancement, so do the methods that are used to ensure that there is no room for errors and other discrepancies.

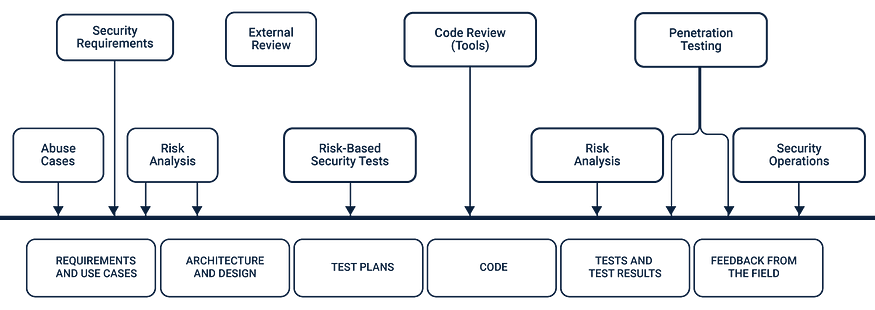
# The Roles Found In Code Review Process:

There are at least two roles that are always present in a code review:-

1. The author, who is responsible for creating the code being reviewed

2. The reviewer, who is the person responsible for examining the code and reporting the results to the author.

The code review process can occur over a large network of programmers and debuggers or can be a shared task among a small group of developers. Often there is a particular platform or methodology (such as Agile) that runs in the background to ensure that the process yields reasonable results.



# Examples of Lightweight techniques:

**Over-the-shoulder**

One developer looks over the author’s shoulder as the latter runs through the code and suggests changes to be made.

This method is more appropriate when dealing with short snippets, and extreme accuracy is required regarding the results.

**Email pass-around**

As the name suggests, the author or the SCM system creates a hierarchical system and emails code to reviewers.

**Pair Programming**

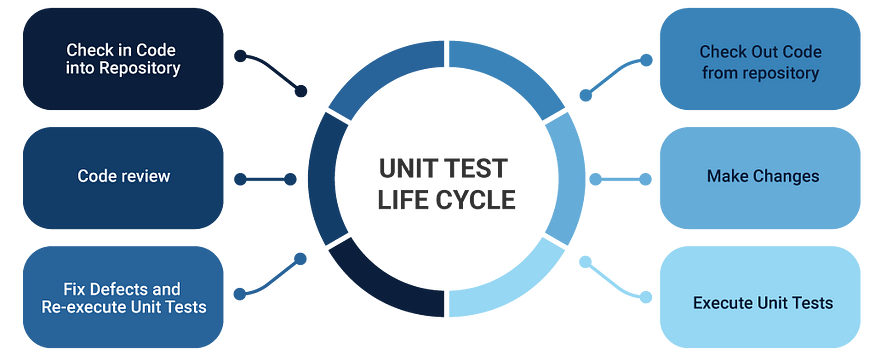
Two authors develop the source code together at the same workstation and reverse engineer the results with the assumption that they would arrive at the same base kernels.

**Tool-assisted**

Obvious to the name, this means that authors and reviewers use specialized tools designed for peer code review.

# Why The Code Review Process Matters:

It’s a rather common question to come across. Why bother reviewing code? The benefits or the goals of the code review process well range in the following reasons:-



**Debugging**

Finding bugs early before the deployment stage can save both time and money especially when they are at a stage when they are cheap to fix.

**Coding Standards Compliance**

Code review helps to maintain consistent coding style across all company applications and respond faster to errors when any situations arise.

**Proliferating Knowledge**

During the code review process, team members gain a better understanding of the code base and learn from each other while also correcting and edifying each other.

**Consistent Design and Implementation**

A peer review process conducted with the objective to debug code helps to maintain a level of consistency in software design and implementation.

**Higher Software Security**

Applications that require a high level of security are hugely benefited from targeted security reviews.

**Team Cohesion**

Review discussions also save team members from isolation and equip them with a greater knowledge of each other’s skill-sets and weaknesses, becoming more responsive to greater issues and discrepancies.

# The Basics of the Code Review Process:



**Review Less Than 400 Lines Of Code**

According to a SmartBear study of a Cisco Systems programming team, the developers should review no more than 200 to 400 lines of code (LOC) at a time citing that the brain can only effectively process so much information at a time.

Say if a team decides to go beyond the 400 LOC limit, the ability to find defects is highly affected.

Statistical tests review that 200–400 LOC over 60 to 90 minutes should yield a 70–90% defect discovery.

**Take your Time And Plan Accordingly**

It can be an intrusive thought to jump through the code and hope to find errors that perhaps someone could find as well in the next chain.

This does, however, produce a drop in defect density at rates faster than 500 LOC per hour. Code reviews in a reasonable quantity that occur at a slower pace for a limited amount of time results in the most effective code review.

**Keep a Time Limit of 60 Minutes**

Performance tends to suffer in a usual code review process after about 60 minutes. Studies show that taking breaks from a task over a period help maintain the flow of work without sacrificing much of it=s quality.

Conducting more frequent reviews should reduce the requirement to conduct a review of this length.

**Set Goals and Capture Metrics**

Decide on how you will measure the effectiveness of peer review and name a few tangible goals.

Use the SMART criteria and start with external metrics. This is done to give you a quantifiable picture of how your code is improving.

**Lookout for the Key Metrics**

Keep a close watch on internal process metrics such as inspection rate- the speed with which a review is performed, defect rate- the number of bugs identified per hour of review, defect density- the average number of bugs per line of code.

**Authors Should Annotate**

Even before the review is initiated, authors should annotate code showing which files to look at first and to justify every source code modification.

Annotations should be directed at other reviewers to simplify and provide more context to the process.

**Use Code Review Checklists**

Omissions can be the hardest defects to find because it´s difficult to review something that isn´t there.

Checklists help eliminate frequently made errors and deal with the challenges of omission finding.

Code review checklists also help members clear expectations for each type of review and can be helpful to find errors for reporting and process improvement purposes.

**Immediately Fix Defects Found**

It’s common for many teams to lack a systematic method for fixing the bugs that they´ve worked so hard to find.

Always use a collaborative code review tool that allows debuggers to log bugs, discuss them with the author, and make approved code changes.

**Exploit and Use Collective Consciousness**

The knowledge that others use to drive out errors and create better products is well contagious in the workplace.

Such an “ego effect” provokes developers to write cleaner code because their peers will certainly see it. Spot checking, for example, produced 20% to 33% of lower defect density with minimal time expenditure

**Practice Lightweight Code Reviews**

If the tools don’t fit the team, they won’t fit the results. Using a lightweight, the tool-assisted process is recommended.

Lightweight code reviews take less than 20% the time of formal reviews and can find as many bugs as a formal review process. Such rigid procedures can at best consolidate work to six participants and hours of meetings paging through detailed code printouts.

**Always Leave Comments**

Code review comments serve as the gateway or the language of communication between the programmer and the user. Even in a team of professionals proficient in coding, it is essential not to leave any second guesses of what the source code is and take some time to explain significant pieces.

This is especially useful when checking variables, values, function calls, loops and conditional statements.

**Decentralise All Information**

All stakeholders must be on the same page when it comes to the code review process and would benefit if there is a uniform understanding of the source code, possible through sharing it with everyone.

If however there may be objections due to the confidential nature of certain data, then keep a separate team trusted to such access that can work on the bits and pieces alone.

