Code Review Process

25 Aug 2017

OVERVIEW

This document outlines the code review process for multiple languages, and common code review practices. This document also provides a quick checklist for the code reviewers to use.

ADVANTAGES OF REVIEW

Code review process is not just to maintain code quality. There are tangible business advantages of a proper code review process. Below are some of the advantages of code review process.

- Reduced Bugs: Since the code review process happens earlier than testing, an iterative code review can help reduce a lot of possible bugs, saving testing time.
- Easier to support: Code review process include checks that can help identify issue with class designs, and even application architecture. Fixing these issues can make software much easier to support and modify.
- Reduced Cost: Code review can reduce testing requirements, and improve change management process, thus overall reducing costs.
- Less surprises: Code review can help identify potential issues in integration of modules with existing modules and systems, thus reducing the surprises in deployment stage.

STATIC CODE ANALYSIS

Static code analysis is the process of reviewing code without actual execution. This analysis can help identify issues like missing error handling code, improper object creation, chances of memory leak, code comments, naming schemes etc. This analysis can be done using automated tools, thus saving time of the reviewer, and providing a way to improve consistency across a breadth of applications across teams.

Below are some of the tools available, that can be used for static code analysis.

StyleCop	.Net (Integrates into Visual Studio)
NDepend	.Net (Integrates into Visual Studio)
CodeRush	Visual Studio Plugin
Codacy	Java, PHP, Python, Ruby, Javascript

GrammaTech	C, C++
Understand	C, C++, Java, PHP, Python, Ruby, C#, XML, ASP
CheckStyle	Java
FindBugs	Java
JSLint	JavaScript
Visual Expert	PL/SQL, T-SQL

BEST PRACTICES

Take It Slow

It is quite a common mistake to try and run through the code review quickly, which defeats the purpose of code review completely. Normally on an average 300-600 lines of code per hour is the maximum you should review. Going faster than this will cause misses, and defects missed in code review will end up as defects in testing, requiring even more testing time.

Take Breaks

When working for a long time, productivity goes down, and chances of mistakes increase. Also when reviewing code, same mistake might appear again and again, which can increase chances of missing out on mistakes even more. It is generally a good idea to take a break every hour from code review to do something else.

Annotate the code

This is a responsibility of the developer, not the reviewer. A properly annotated code can help reviewer understand the reasoning for a lot of choices made during development. This can improve efficiency of the code review, allowing for better chances of finding defects.

Use checklists

A properly defined checklist can help reviewer stay on the goal. A checklist can help to make sure that reviewer does not miss out on checking some common fallouts. For example, in some cases, reviewer might forget to check if null checks are defined properly.

Establish a review positive environment

It is quite easy for developers to feel irritated at bad reviews on their code. A positive review environment guarantees that developers consider bad review as an opportunity to learn and not as extra burden.

Use tools when possible

For simple, easily automatable parts of code review (e.g. proper alignment, variable names), using tools can save a lot of time. Taking advantage of the tools mean that reviewer can focus on dynamic assessment of the code.

Explain principle instead of defect

When a defect is found in the code review, try to explain which principle is invalidated by the defect. This will ensure that developers understand the reasoning behind a defect and will stop making same mistakes again.

Encourage Learning

The idea of code review is not just to review the code, but to help developers understand the mistake. Try to share links and tutorials with developers to help them learn, instead of bringing all the topics during review itself.

Document everything

This should go without saying for all project activities. But it is a common mistake that happens so regularly, that it should be emphasized. Foster an environment, where developers document everything they learnt, all defects found, how they are planning to reduce defects in future, and so on. A well documented project is easiest to maintain, even if the code itself is difficult to understand.

Prioritize requirements

During code review, while requirements are not to be tested against, it is a good idea to read about the same. If the reviewer has enough understanding of the requirements, review process becomes easier. It is possible to find missing requirements, during code review as well.

BASIC CHECKS

Code review can be done in two stages. First is to check for basics. Code that fails these basic checks, should not make it into the review process itself. If the developer is missing out on these basic checks, figure out training process to improve developer's skills. Below are some of the basic checks.

Is the code understandable?

First thing a reviewer should ask is “Can I understand this code?”. An easy to understand code will never require reviewer to ask a developer. Variable naming convention, properly commented code etc are some of the ways developer can make code easy to understand.

Are the requirements followed?

Requirements are the way, a developer understands the whole project. If a developer is not following requirements, it means he/she has not understood the project properly.

Is the code duplicated?

Duplicating code once is acceptable, as in lot of cases, that is required. But if the developer is duplicating code for more than two times, that means he/she has not found the right solution.

Is the code easy to debug?

Idea here is that developer who is fixing a piece of code might not even be the one who was part of initial development team. A code that is difficult to debug will cause the problems for anyone who is trying to improve the code.

Is the class/function too big?

While “too big” is a subjective term, a general rule is that a function should not do more than one thing, and not have more than 20 lines of code. E.g. if a function is making a connection and executing the query, it is too big and should be divided in two.

ADVANCED CHECKS

Second step of the code review should consider these checks. These checks are used to determine if the code is of good quality. While piece of code that fails these checks can be used in a pinch (during deadlines, or crunch time), it is generally advisable not to use such code. These checks go into more depth than basic checks, and require careful review.

Code formatting

While in basic check, we consider that the code is readable, here we are concerned with “how readable the code is?”. For example, wrong alignment, too big lines, unintelligible function names etc can cause readability issue. While the code can still be understood, it reduces the productivity of the developer maintaining the code.

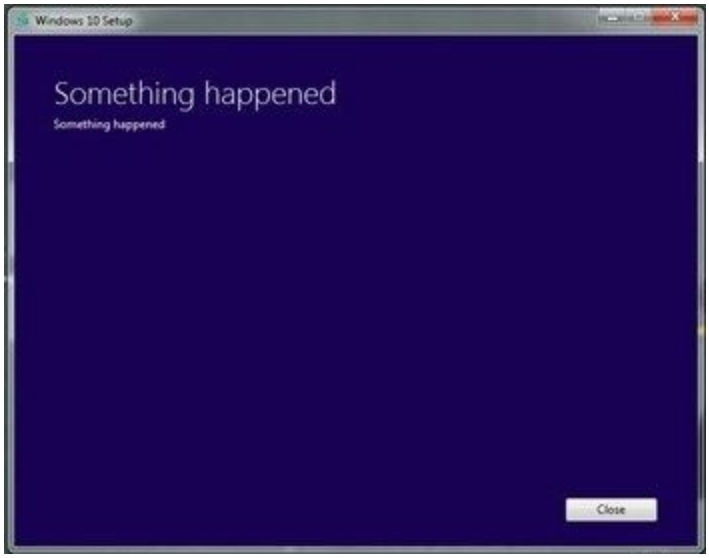
Architecture

Making sure, code is following the architectural guidelines can solve a lot of problems in the future. A piece of code that is in wrong layer will require changes in two layers at the later stage.

A class that is not closed for modification will cause changes in modules other than the one being maintained.

Best Practices

There are few coding best practices that improve usability of code. For example, removing hard coded values, reducing access levels, using proper looping structure, displaying proper error messages can help improve maintainability. Below is an example of a bad error message.



During advanced checks, reviewer should make sure that this is not a common occurrence.

Non Functional Requirements

Following are some of the non functional requirements, that should be considered during advanced checks.

- Reusability - Code should be easy to reuse as necessary, and as generic as possible.
- Extensibility - Code should allow developers to extend and add features as needed.
- Reliability - Code should be reliable for all types of interactions possible, and should have proper exception handling, input filtering.
- Security - Code should consider all possible forms of attack like sql injection, cross site scripting.
- Performance - Code should perform well under heavy load, using parallel processing, caching and other similar technologies.
- Usability - Code should be easy to use by someone who is not well versed in the application. A user should not be required to read the manual for basic tasks.

CHECKLIST EXAMPLE

The checklist for code review should reflect the technology used, the application architecture and client's requirements. E.g. if client does not need resolution independence, there is no need to check for that. Since a properly reviewed code is extendable, the feature can be added if required.

Considering that, here is a brief checklist that can be used as a template.

Use Intention-Revealing Names	Meaningful Names
Pick one word per concept	Meaningful Names
Use Solution/Problem Domain Names	Meaningful Names
Classes should be small!	Classes
Functions should be small!	Functions
Do one Thing	Functions
Don't Repeat Yourself (Avoid Duplication)	Functions
Explain yourself in code	Comments
Make sure the code formatting is applied	Formatting
Use Exceptions rather than Return codes	Exceptions
Don't return Null	Exceptions

Make class final if not being used for inheritance	Fundamentals
Avoid duplication of code	Fundamentals
Restrict privileges: Application to run with the least privilege mode required for functioning	Fundamentals
Minimize the accessibility of classes and members	Fundamentals
Document security related information	Fundamentals
Input into a system should be checked for valid data size and range	Denial of Service
Avoid excessive logs for unusual behavior	Denial of Service
Release resources (Streams, Connections, etc) in all cases	Denial of Service
Purge sensitive information from exceptions (exposing file path, internals of the system, configuration)	Confidential Information
Do not log highly sensitive information	Confidential Information
Consider purging highly sensitive from memory after use	Confidential Information
Avoid dynamic SQL, use prepared statement	Injection Inclusion

Limit the accessibility of packages,classes, interfaces, methods, and fields	Accessibility Extensibility
Limit the extensibility of classes and methods (by making it final)	Accessibility Extensibility
Validate inputs (for valid data, size, range, boundary conditions, etc)	Input Validation
Validate output from untrusted objects as input	Input Validation
Define wrappers around native methods (not declare a native method public)	Input Validation
Treat output from untrusted object as input	Mutability
Make public static fields final (to avoid caller changing the value)	Mutability
Avoid exposing constructors of sensitive classes	Object Construction
Avoid serialization for security-sensitive classes	Serialization Deserialization
Guard sensitive data during serialization	Serialization Deserialization
Be careful caching results of potentially privileged operations	Serialization Deserialization

Avoid excessive synchronization	Concurrency

Keep Synchronized Sections Small	Concurrency
Beware the performance of string concatenation	General Programming
Avoid creating unnecessary objects	Creating and Destroying Objects

Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	Exceptions
Favor the use of standard exceptions	Exceptions
Don't ignore exceptions	Exceptions
Check parameters for validity	Methods
Return empty arrays or collections, not nulls	Methods
Minimize the accessibility of classes and members	Classes and Interfaces
In public classes, use accessor methods, not public fields	Classes and Interfaces
Minimize the scope of local variables	General Programming
Refer to objects by their interfaces	General Programming

Adhere to generally accepted naming conventions	General Programming
Avoid finalizers	Creating and Destroying Objects
Always override hashCode when you override equals	General Programming
Always override toString	General Programming
Use enums instead of int constants	Enums and Annotations
Use marker interfaces to define types	Enums and Annotations
Synchronize access to shared mutable data	Concurrency

Kindly consider that these are some of the possible checks, and are not required to be used all the time. Based on the project, few of these checks can be eliminated. These checks are just to provide a starting point for the actual checklist defined for the project.