Development - Code Review Checklist

12/29/2023 1:32 pm EST

Code Review Checklist

It's hard to see your own mistakes, so a code review by another person is the best way to catch these. What follows is a checklist of what you should look for when reviewing other people's code (adapted from Evoke Technologies' Code Review Checklist):

Code Formatting

☐ **Is the code properly aligned?** It should be trivial to identify the starting point and end point of a given block of code.

☐ **Are the proper naming conventions of the language being followed?** e.g., `javaScriptMethods()` have different formatting than `CSharpMethods()`.

☐ **Are the lines short enough?** Not all of us have 40" monitors. If you have to scroll horizontally to read the rest of the code, it probably should be broken off into multiple lines.

☐ **Are large blocks of commented code removed?** While it's nice to comment out stuff that you won't need and squirrel it away for later, we have IDEs that are integrated into version control, and as such, can see old code pretty quickly.

Architecture

Are Separation of Concerns followed? Does each layer perform only its given responsibility? Does each class have a specific single responsibility? Each method? Is the JavaScript in `.js` files and the CSS in `.css` files, etc.?

☐ **Are the team's code patterns being followed?** If it makes sense to define something in a particular area of the application, is the new code following this?

☐ **Are the appropriate design patterns being used?** Make sure that the right pattern is being used for the job (this requires understanding the problem and its context).

Best Practices

☐ **Are there any "magic strings"?** These are a bad practice, as they are subject to errors and debugging headaches. Instead, consider using constants and configuration values.

☐ **Are common values grouped under an enum?** Once again, this helps avoid "magic strings," and groups similar logic together.

☐ **Do the comments make sense?** What you are doing should be apparent from the code. However, *why* you are doing it is another matter. These should be well documented, especially if it "looks" wrong. Annotate hacks, tweaks, workarounds, and temporary fixes. Document todos as well.

☐ **Are the `if/else`s reasonable?** Too many `if/else` blocks can make the code unreadable. Consider replacing some with `switch`es, or extracting out the logic into separate methods.

☐ **Are wheels being reinvented?** If there is a perfectly good framework solution to your problem (or a well documented/supported

NuGet/Node package), don't reinvent the wheel and produce a custom solution.

☐ **Are Asyncs Awaited?** Especially during shared context calls, all `asyncs` should be `awaited` before any new requests are made on that context (and before any controller actions return).

Non-Functional Requirements

☐ **Is it maintainable?** Imagine that you're going to have look at this code one day years from now. Do you want to have to fix any problems with it?

☐ **Is it readable?** If the code doesn't tell you what it does, then it needs to be made more clear, either by refactoring or by adding comments.

☐ **Is the code clear?** If what the code does is not clear, then it either needs to be refactored to be more clear, or it needs comments to explain what it's doing. If you can't reasonably follow the flow of the code, then it's not clear.

☐ **Is it testable?** One of the main reasons we don't test our code is because we don't write *testable* code. Consider writing code that is easy to test, and then we can know quickly if it's right or if it's wrong.

☐ **Is it debuggable?** Make sure that if the code breaks, it logs where and why it breaks, so we can get in there and fix it.

☐ **Is it configurable?** Put our config values somewhere where they can be changed without redeploying (e.g., in a config file or in the database).

☐ **Is it reusable?** Make sure the code is DRY (Don't Repeat Yourself). That is, don't replicate business logic. If a business requirement changes, you want to have to only change it in once place. If you're copying and pasting code, then that's a place to consider a different approach.

☐ **Is it reliable?** Does is handle exceptions gracefully? Does it clean up after itself and dispose its resources?

☐ **Is it extensible?** Is it easy to enhance the code with minimal changes? Is it able to be easily replaced by another component providing the same functionality with better quality?

☐ **Is it secure?** Does it need to use authentication, authorization, input validation, or encryption? If so, make sure that performs these duties. Don't trust the front-end.

☐ **Is it performant?** Make sure that if you make a database call, you're only pulling back what you need when you need it. Cache where appropriate and `.ToList()` only when it makes sense to.

☐ **Is the code too complex?** Cyclomatic complexity is the measure of the complexity of a program. If a given method is too complex, it has a higher probability of breaking, either through an unforeseen bug, or by a future developer altering the code without understanding the complex ramifications. Where necessary, break out more complex pieces of the code into smaller, testable methods. Also, consider the Single Responsibility Principle (the S in SOLID).

☐ **Is it scalable?** Can this method handle being run concurrently a dozen times? A hundred? A thousand?

☐ **Is it usable?** How would the end-user feel using this code? Is it cumbersome or clunky? If it's not enjoyable for you to use the product, imagine somebody who has to use it *in order to do their job* .

SOLID Code

Single Responsibility Principle Does each class or function have a single responsibility? Is that responsibility held by only a single class or function?

- [ ] **Open/Closed Principle** New functionality should not modify existing code, but rather should be in its own new class or function (this does not include bug fixes, which specifically should modify existing code).
- [ ] **Liskov Substitution Principle** A derived class should be able to be substituted for its parent without changing the behavior. See DotNetCurry's article for a better understanding of this.
- [ ] **Interface Segregation** Make sure that the interfaces are split up based on functionality.
- [ ] **Dependency Injection** Instead of passing in a dependency as a class, pass it in as an interface.