Self Documenting Code: Parts I, II, & III

Comments for readability?

- Readability → ability of a programmer to understand what a piece of code does, correctly, in the shortest amount of time.
- There are times when we need to write comments, However, if we do a good job writing a self-documenting code, we would need fewer of those comments.

Clear instead of clever

- Clever code is hard to understand and tricky.
- Clear code is easy to follow.
- A clear code ends up being self-documenting. It’s quite transparent.
- Clever code often hides details and leaves our things to be discovered/identified. It’s often opaque.

No rush

- Take the time to read the code you write.
  - Are we handling different situations?
  - Are we looking at edge cases?
  - Is the code clear?
  - Is the code walking through in a logical order?
  - Is the code short enough?
  - Are the variable/function names good?
- Review

    - Give others your code to review.
    - Saves time — as it lessens the bugs.
- Write automated tests
    - When writing automated tests, we are thinking through the consequences of this code —what the expectations are, positive, negative, and exception situations.
- Code defensively

Tests as documentation

- Automated tests are/can be used as a form of documentations.

Comments whys not whats

- Comment why the code was written the way it way — maybe there's a constraint or the code only works under certain conditions.
- Commenting what the code does violates DRY.

Keep the documentation in code

- When it comes to **classes**, we want a very high level description; Just a short paragraph on the purpose of this abstraction — what does it represent.
    - The name of abstraction should be meaningful and usually it is good enough. However, if you feel that there's something that needs to be said about this abstraction, then that can be documented with a short paragraph.
- Take the courage in not writing a documentation when the documentation provide any value at all.
- When it comes to writing documentations for **methods**, there are four things that should be there.
    1. **Purpose**

a. What's the essential reason for the existence of this function in the application?
b. In a very short paragraph — 1-2 sentences are adequate.
c. Write what the functions achieves/provides for us and what are the consequences of running this method.

2. **Requirements**
    a. What are the pre-conditions before we can call this method?
        i. What should the state of the object be before this method is called.
        ii. What are the expectations on the parameters?
            1. Can a number parameter be negative?
            2. Does a number parameter be in some range? If so, what's that range and why?
3. **Promises**
    a. Post-conditions of calling a method.
        i. If a method is called, in what state will it leave an object?
        ii. The expectations that we can have on the results the function returns.
        iii. Will there be any change in any state?
        iv. Are there any side effects?
4. **Exceptions**
    a. A list of exceptions that can be thrown by this method and why.
        i. This will help the programmer know what to expect of this method.
        ii. When someone is overriding this method in a derived class, they will know what exceptions could possibly be thrown and for what reasons and they would know not to throw any exceptions that they not supposed to throw.
            1. Throwing any new exception would be a violation of the Liskov Substitution Principle.

    - The pre-conditions and the post-conditions of the method should be supplemented with automated tests as well alongside documentation.
        - If you only document these with words, there's a risk of a programmer eventually modifying those conditions. If this happens,
            - the documentation becomes stale.
            - we will have violated some expectation in the code and that may go unnoticed until it's discovered as a bug.
        - Anytime the code is modified (if one of the pre-conditions or the post-conditions has been modified), the automated tests will let us know that the code is violating some expectations.
- Tools can be used to create human readable documentation (javadoc, doxygen, ndoc, rdoc, …)
    - These can be used to create a documentation.
    - As we evolve the code, we can keep up with the documentation as well.
- At least provide a pointer to where the documentation is, if it cannot be in the code.
    - Especially important if you documentation is going to talk about the design reasoning.

Variable names

- Not too long, but not cryptic either. Concise and expressive.
- Avoid magic numbers.
    - Eg. `barista.order(3) // large`
        - What does the 3 mean in this case? Does it mean 3 cups of coffee?
        - In the `order()` method, we might find the variable `int l`, which doesn't really mean anything. Going through the documentation might tell that it mean a large cup of coffee.
        - Better → `barista.order(CoffeeSize.LARGE)` or `barista.order(LARGE_COFFEE)`.

Assertions for assumptions

- `assert(quantity > 0)`
- Asserts can be helpful of specifying the pre-conditions we are expecting.

Document unique, special, unexpected conditions

- Some of these can be expressed as assertions.
- Great to have automated tests for these as well.

What if comments become redundant, totally?

- What if most of the documentation is in a foreign language?
- If the code is written really well and is self-documenting, you can just read the code and see what’s going on.

Fix a smell instead of commenting it

- Rather than writing a comment about a code smell, just fix and refactor the smell out.
- If you cannot remove the smell right away, write an automated test that describes the particular smell and why it cannot be removed, and an expiration label, so it can be refactored later on when it is appropriate.