anti-pattern

lack of documentation + comments

Why it's a Bad Pattern

1. **Increased Cognitive Load**
    - When code lacks proper documentation, it becomes difficult for all developers (including future-you) to understand the logic and purpose behind decisions, big and small.
2. **Poor Maintainability**
    - If code logic and purpose remain a challenge, developers are less likely to implement regular maintenance cadences.
    - Without regular maintenance cadences, your code is at risk of becoming esoteric knowledge at best, and losing overall functionality and value at worst.
3. **Increased Time Spent on Debugging**
    - If code logic and purpose remain a challenge, developers are less likely to effectively identify, respond to, and resolve bugs.
    - Not only do bugs become more expensive, but one risks not resolving them at all.
    - No effective debugging methodology risks destructive consequences and accelerates code degradation.
4. **Overall Team Frustration with Company-Wide Consequences**
    - Poor documentation practice puts the team at risk of perpetuating engineer toil that can ripple across additional team processes, possibly creating prolonged and unproductive team ceremonies due to unknowns and misalignment.
    - Documentation can be a tool in which developers effectively share knowledge, but with poor documentation practice, we all risk perpetuating bus problems, difficult onboarding processes, and unnecessarily complex shipping.

Examples of the Problem

Popular documentation strategies include inline multi-line remarks that risk adding noise, not context. For example, using Javadoc comments can be an effective strategy to ease cognitive load, but if you're not using this real estate to provide meaningful information, you're just creating noise. Using the Javadoc structure to produce a parameter list without description ends up looking like unintended code duplication.

```
/**
* Method to create a subscription invoice. This method calls another
* service and records the invoice generation to table_invoice_requests
* reporting in that service.
*
* @param customerId
* @param invoiceDate
* @param startDate
* @param endDate
* @param productId
* @param amount
* @param description
* @param uniqueId
* @return response message that is displayed to the user
*/

public String createSubscriptionInvoice (
       final int customerId,
       final LocalDate invoiceDate,
       final LocalDate startDate,
       final LocalDate endDate,
       final int productId,
       final double amount,
       final String description,
       final String uniqueId) {
```

Beware of orphaned TODO tasks. Inline commenting is only encouraged for areas involving edge case handling, short-span fixes, product quirks and magic numbers or constants that require a quick explanation.

```
// TODO: remove this
// TODO: I should not set this here, like this. Should just return the ID.

// TODO server side validation and rerendering is a little odd, consider
JavaScript front-end validation
```

These TODO tasks are marginally more valuable than the first. We have some direction, but no meaningful explanation as to why the current implementation is not preferred. We've made the effort to create a TODO task without successfully passing responsibility. The codebase is now littered with orphaned TODO tasks, creating unnecessary IDE warning noise, making identifying and solving bugs harder.

Recent outdated service documentation recently led to an internal event (<Jira ticket link here>), which introduced breaking changes to 162+ projects (see this <internal event report> here).

Best Pattern(s)

1. Write for future-you, but not only future-you.
    - Aim away from implicit knowledge and be as literal, explicit as possible.
    - Disambiguate as much as you can while writing for the engineer that knows nothing, which will be future-you, because future-you is not going to remember.
    - Even if future-you remembers, stop putting pressure on yourself to become a code librarian, so that you can allot your time to more complex problems.
2. Be decisive, don't leave strays.
    - Write kinetically. Write in a way that shares valuable information and passes responsibility. Meaningful comments document *the why, not just the what*.
    - If you must create a TODO task, use this opportunity to be descriptive in a way that transfers knowledge because "anything you do in a team should follow the needs of that team" (Martin Fowler, Code As Documentation).
    - A good approach may include adopting a team standard in which a Jira ticket is required in any, and or all TODO

comments. Consider adding a code review checklist item to examine any comments that could or should be updated.
  - A better approach might be to *never leave* TODO tasks at all. Either complete the task or finalize the decision not to. If the scope of the change is too much for an immediate resolution, a ticket is a better way to share information and manage change, rather than leave stray comments in the code. A ticket can centralize context, while comments often require imperfect cycles of context reinforcement - where one is forced to learn and relearn context.

3. Aim to be concise
   - Good code should document itself, but that becomes a challenge when you're working with legacy systems that require explanation for complex business processes or distinctive legal constraints. If possible, suggest refactoring to make the code itself more clear to eliminate the need for inline comments as documentation. It is often enough to extract a method with a meaningful name to expose the information you might otherwise put into a comment.
   - After considering refactoring, if you think inline comments are necessary, focus on the why and allow the code to speak for itself to explain the how. If staying concise is a challenge, consider a different approach to documenting the information. Could this information live among your team's internal documentation, repository README, or external service documentation? Inline comments with too much information can very easily go from helpful to not.
   - Finding the right balance just takes regular maintenance cadences. Aim to include necessary refactoring, read good docs, have others review your docs, remain open to feedback that improves user-friendliness and practicality.
4. Adopt tools that help you build a solid documentation practice
   - Harness the power of your IDE and don't ignore the warnings.
   - Use current recommendations to support larger initiatives
   - Leave code better than you found it

Examples of the Best Pattern(s)

If you're writing a repository README, aim to be comprehensive but not exhaustive. Strive to create a user-friendly onboarding experience. Include an introduction to the project with visuals and text. Offer clarity around various entry points such as quickly contributing code and how one might respond to an outage. Seek to produce content that remains practical and lean circumstantially agnostic. Ensure that if you're new to the project, this content is a great place to start and that if you're a site reliability engineer responding to an issue, this is also a great place to start.

Sweat the small stuff and acknowledge system limits. Notice that the documentation below offers meaningful direction because it is not unnecessarily abstract and is descriptive of the limitations. Something too concise such as `add the Jenkins user` would not tell us much.

```
You will need to type "Jenkins" into the search and scroll down to
the "Users" section of the drop down results where Jenkins will pop
up. Even though Jenkins is dead, we still require this. After adding
these, the option will read 1 role, 1 user. You might also see some
extra users like the Timebound Merger and that is fine.
```

Summary

1. Write for the engineer who has no context, which will include future-you, because future-you is not going to remember.
2. Code can be self-documenting, but it probably isn't. Discuss team strategies to eliminate the need for inline comments as documentation, but as long as it exists, encourage best practices.
3. Write meaningful comments and project documentation, regularly update it, and solicit feedback for improvements.
4. You don't need to reinvent the wheel
    - Seek out current best practices guidelines and documentation recommendations from others
    - Experiment with IDE code inspection tools
    - Read good docs and share good docs