NODE Technical Book Club

A Philosophy of Software Design - John Ousterhout

Modifying Existing Code

- A system's design evolves over time but we need to keep complexity from growing.
- Take a strategic approach and refactor after the change if needed.

Maintaining Comments

- Place comments near the code they describe.
- Spread implementation comments throughout the code.
- Document changes in the comments in addition to the commit message.
- Avoid duplication.

Consistency

- Powerful tool for reducing complexity and making its behavior more obvious.
- It can be applied to:
  - Naming
  - Coding style
  - Interfaces
  - Design patterns
  - Invariants

Ensuring Consistency

- It is hard to maintain consistency. Tips:
  - **Document** the conventions.
  - **Enforce** them using tools.
  - Don't change existing conventions.

Code Should be Obvious

- Non-obvious code is hard to understand and prone to bugs due to misunderstandings.
- Good names and consistency help making code obvious.
- Judicious use of white space makes code more readable.
    - Use comments to explain non-obvious parts.

🚩 Red Flag: Nonobvious Code

If the meaning and behavior of code cannot be understood with a quick reading, it is a red flag. Oten this means there is important information that is not immediately clear to someone reading the code.

Things that make code less obvious:

- Event-driven programming
- Generic containers
- Different types for declartaion and allocation
- Code that violates reader expectations

Ensure that readers always have the information they need to understand the code. To do this:

- Reduce the amount of information needed.
- Take advantage of common knowledge.
- Present the important information using good names and comments.

Software Trends

Let's discuss the trends in software development from the book's perspective.

Object-Oriented Programming

- If used carefully, it can help to produce better software design.
- Private members and methods are useful for information hiding.
- Interface inheritance can lead to more deep interfaces.
- Implementation inheritance reduces duplication so reduces change amplification. But it creates dependencies and results in information leakage.

Agile Development

- Agile development suggest incremental and iterative development as in the book.
- The risk about agile is that it can lead to tactical programming.

Unit Tests

- A good software design requires continuous improvement and refactoring.
- Tests are crucial for refactoring because they provide a safety net.

Test-Driven Development

- TDD focuses on getting the features running, rather than finding the best design.
    - It makes sense for bug fixing.

Design Patterns

- They represent alternative to design: rather than creating a new design, you can use a pattern.
- They can help making code more obvious and consistent.
- But be careful about overusing them.

Getters and Setters

- A popular design pattern in Java.
- They are shallow and should be avoided.

Designing for Performance

- How should performance considerations affect software design?
- The key is to develop a sense of what is likely to be a performance problem. Examples:
  - Network communication
  - Disk I/O
  - Dynamic memory allocation
  - Cache misses

- Usually, a more efficient solution is as simple as slower one.
- If the only way to improve efficiency is by adding complexity:
  - If you can hide it, it may be worthwile.
  - Otherwise consider starting with a simpler solution and only add complexity if needed.
  - If you have clear evidence that the performance will be a problem, only then consider starting with the more complex solution.

Measure Before (and After) Optimizing

- Don't rely on your intuition.
- **Measure before** to identify the bottlenecks and to have a baseline.
- **Measure after** to see the impact of your changes.

Design Around the Critical Path

- If you need to redesign for the performance gain:
- Find the minimal code that must be executed in the most common case.
- Ignore current design and all special cases.
- Look for a new design that is good and critical path is close as possible to the minimal code.
- Try to minimize the number of special cases to check.

Decide What Matters

- Structure software systems around what matters most.
- To decide what matters, look for leverage.
    - Try to make as little matter as possible.
    - Emphasize things that matter.

Conclusion

- The book is about one thing: **complexity**
- Described root causes of complexity and strategies to reduce it.
- Presented red flags to help identify complexity.
- Promoted investment mindset over tactical programming.