NODE Technical Book Club

A Philosophy of Software Design - John Ousterhout

Introduction

(It's all about complexity)

Complexity is the root of all evil in software.

Two approaches to fight complexity:

- Making code simpler and more obvious
- Encapsulating it (Modular design)

Goals of the book:

- Describe the nature of software complexity
- Present techniques to minimize complexity

The Nature of Complexity

- Complexity is anything related to the structure of a software system that makes it **hard to understand and modify the system**.
- More common parts of code has more impact on complexity.

- Symptoms of complexity:
  - **Change amplification**: Small changes require large amounts of code to be modified.
  - **Cognitive load**: Hard to keep the entire system in your head.
  - **Unknown unknowns**: Hard to predict the impact of changes.

- Causes of complexity:
    - **Dependencies**: A change in one part of the system requires changes in other parts.
    - **Obscurity**: Important information like a dependency is not obvious.

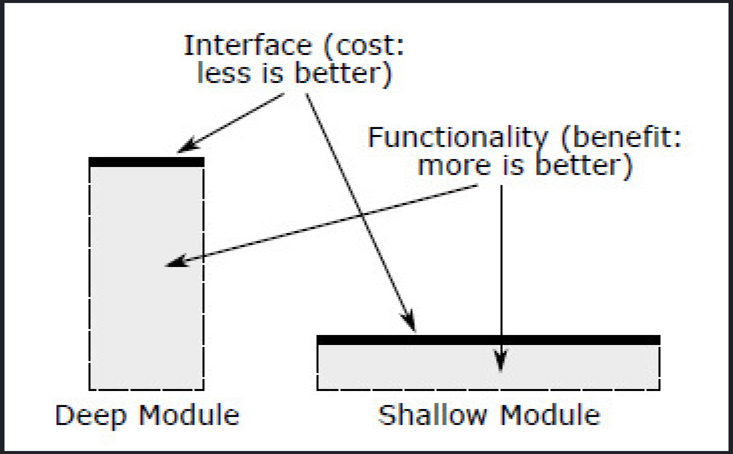
Working Code Isn't Enough

(Strategic vs. Tactical Programming)

- Tactical programming: Get it to work as quickly as possible.
- Strategic programming: Think about the long-term implications of your design decisions.
    - The question is how much to invest?

Modules Should Be Deep

- The goal of modular design is to minimize dependencies between modules.
- Think of a module in two parts:
  - **Interface**: What does the module do?
  - **Implementation**: How does it do it?
- Best modules are those whose interfaces are much simpler than their implementations.



- Module depth is a way of thinking about cost versus benefit.
  - **Benefits**: The functionality provided by the module.
  - **Costs**: Its interface

🚩 Red Flag : Shallow Module

A shallow module is one whose interface is complicated relative to the functionality it provides. They don’t help much in the battle against complexity.

```
private void addNullValueForAttribute(String attribute) {
  data.put(attribute, null);
}
```

Classitis

Example: Java class library

```
FileInputStream fileStream =
  new FileInputStream(fileName);
BufferedInputStream bufferedStream =
  new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
  new ObjectInputStream(bufferedStream);
```

Interfaces should be designed to make the common case as simple as possible.

Information Hiding (and Leakage)

- **Information hiding**: Each module should encapsulate a few pieces of knowledge. It is embedded in implementation and not visible in the interface.
    - It simplifies the interface.
    - It makes it easier to evolve the system.
- Design question: What information can be hidden in that module?

🚩 Red Flag : Information Leakage

Information leakage occurs when the same knowledge used in multiple places, such as two different classes that both understand the format of a particular type of file.

🚩 Red Flag : Temporal Decomposition

In temporal decomposition, execution order is reflected in the code structure: operations that happen at different times are in different methods or classes. If the same knowledge is used at different points in execution, it gets encoded in multiple places, resulting in information leakage.

Example: HTTP Server

Common Mistakes:

- Excessive class division led to information leakage.
- Shallow parameter handling interfaces exposed internal representations.
- Inadequate defaults in HTTP responses increased complexity. (🚩 **Red Flag: Overexposure**)

General Purpose Modules are Deeper

- Specialization leads complexity.
- There is a trade-off between generality and current needs.
- **Make classes somewhat general purpose.**
- The functionality should reflect your current needs, but its interface should be general enough to accommodate future needs.

Example: Text Editor

- Specialized Approach:
  - Text class designed with specialized methods like backspace and delete.
  - Methods tailored to specific user interface operations.
  - Increased cognitive load for developers and information leakage between text and user interface classes.

- General-Purpose Approach:
  - Text class with generic methods for text modification (insert, delete).
  - Utilizes a generic type Position instead of specific UI-related types.
  - Cleaner separation between text and UI classes, promoting better information hiding.

Questions to Ask Yourself

- What is the simplest interface that will meet my current needs?
- In how many situations will this method be used?
    - Is this API easy to use for my current needs?

Push Specialization Upwards and Downwards

- Specialized features located at top-level classes, keeping lower-level classes general-purpose.
- Specialized features delegated to lower-level classes, maintaining core functionality as general-purpose.
- **Example**: specialized undo handlers separated into a History class while core functionality remains general-purpose.

Different Layer, Different Abstraction

- In a well-designed system, each layer provides a different level of abstraction.
- 🚩 **Red Flag**: If a system contains adjacent layers with similar abstractions, it suggests a problem with the class decomposition.

🚩 Red Flag : Pass-through Methods

```
public class TextDocument ... {
public Character getLastTypedCharacter() {
  return textArea.getLastTypedCharacter();
}
public void insertString(String textToInsert,
int offset) {
  textArea.insertString(textToInsert, offset);
}
public void willInsertString(String stringToInsert,
int offset) {
  if (listener != null) {
    listener.willInsertString(this, stringToInsert, offset);
  }
}
```

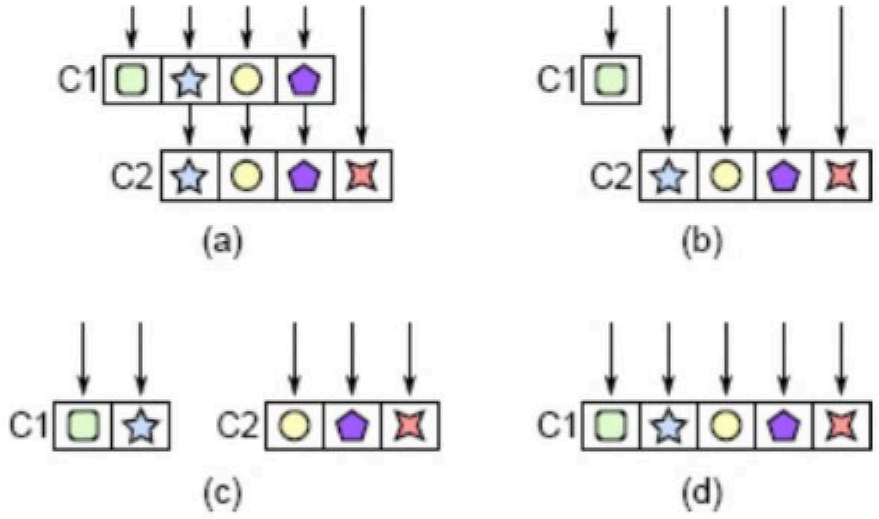


Figure 7.1: Pass-through methods. In (a), class C1 contains three pass-through methods, which do nothing but invoke methods with the same signature in C2 (each symbol represents a particular method signature). The pass-through methods can be eliminated by having C1's callers invoke C2 directly as in (b), by redistributing functionality between C1 and C2 to avoid calls between the classes as in (c), or by combining the classes as in (d).

When is interface duplication okay?

- Each method should contribute significant functionality.
- **Example**: Dispatcher

Decorators

- Decorator is used to seperate specializations from general-purpose classes.
- It encourages API duplication across layers and they tend to be shallow.
- **Consider the alternatives**:
  - Add directly to underlying class.
  - Merge with an existing decorator.
  - Implement as a standalone class.

Pass-Through Variables

- A variable is passed through a long chain of methods without being used.
    - **To eliminate**:
        - Store in a shared object.
        - Use a global variable.
        - Use a context object.

Pull Complexity Downwards

- Handle complexity internally instead of exposing to users.
- It is more important to keep the interface simple than to keep the implementation simple.
- **Example**: Configuration parameters
  - **Bad**: Let the user choose the parameters(handle complexity).
  - **Good**: Determine right defaults internally.
- Do not take it too far, consider overall complexity.

Comments