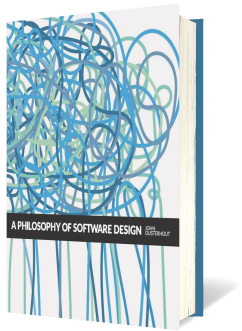


A Philosophy of Software Design

Diego Pacheco

About me...

- ❑ Cat's Father
- ❑ Principal Software Architect
- ❑ Agile Coach
- ❑ SOA/Microservices Expert
- ❑ DevOps Practitioner
- ❑ Speaker
- ❑ Author

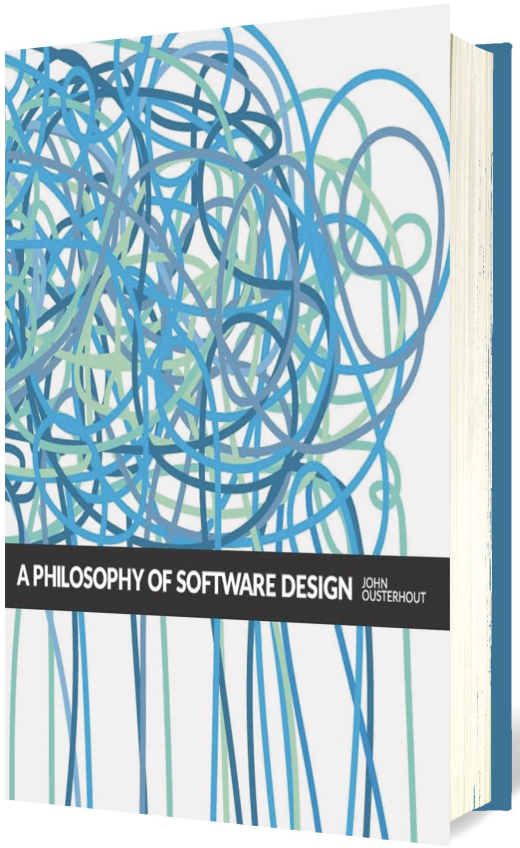
diegopacheco

@diego_pacheco

http://diego-pacheco.blogspot.com.br/

https://diegopacheco.github.io/

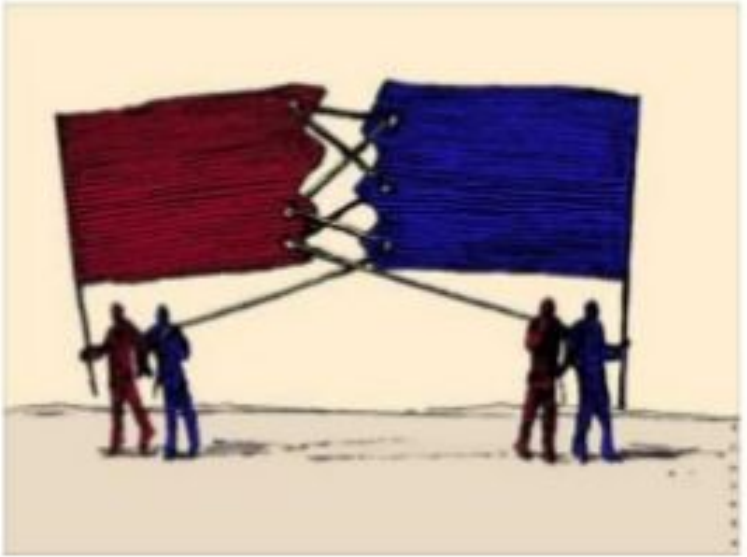
Just read it...



As time goes by...

Movement vs Anti-Moviment

- SOA
- Agile
- REST
- Docker
- Clean Code
- ...



It's all about Complexity

Complexity is related to Structure which makes complex to Understand and Modify.

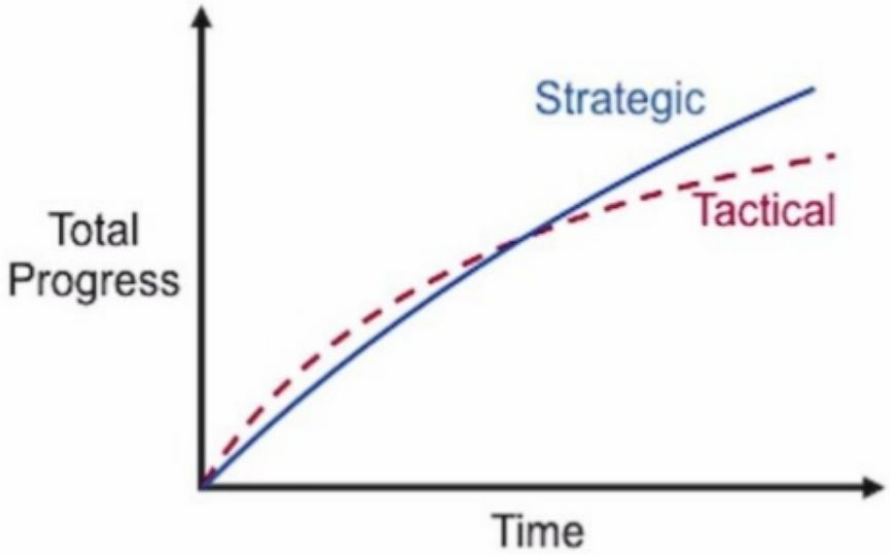
It's all about Complexity

Symptoms of Complexity

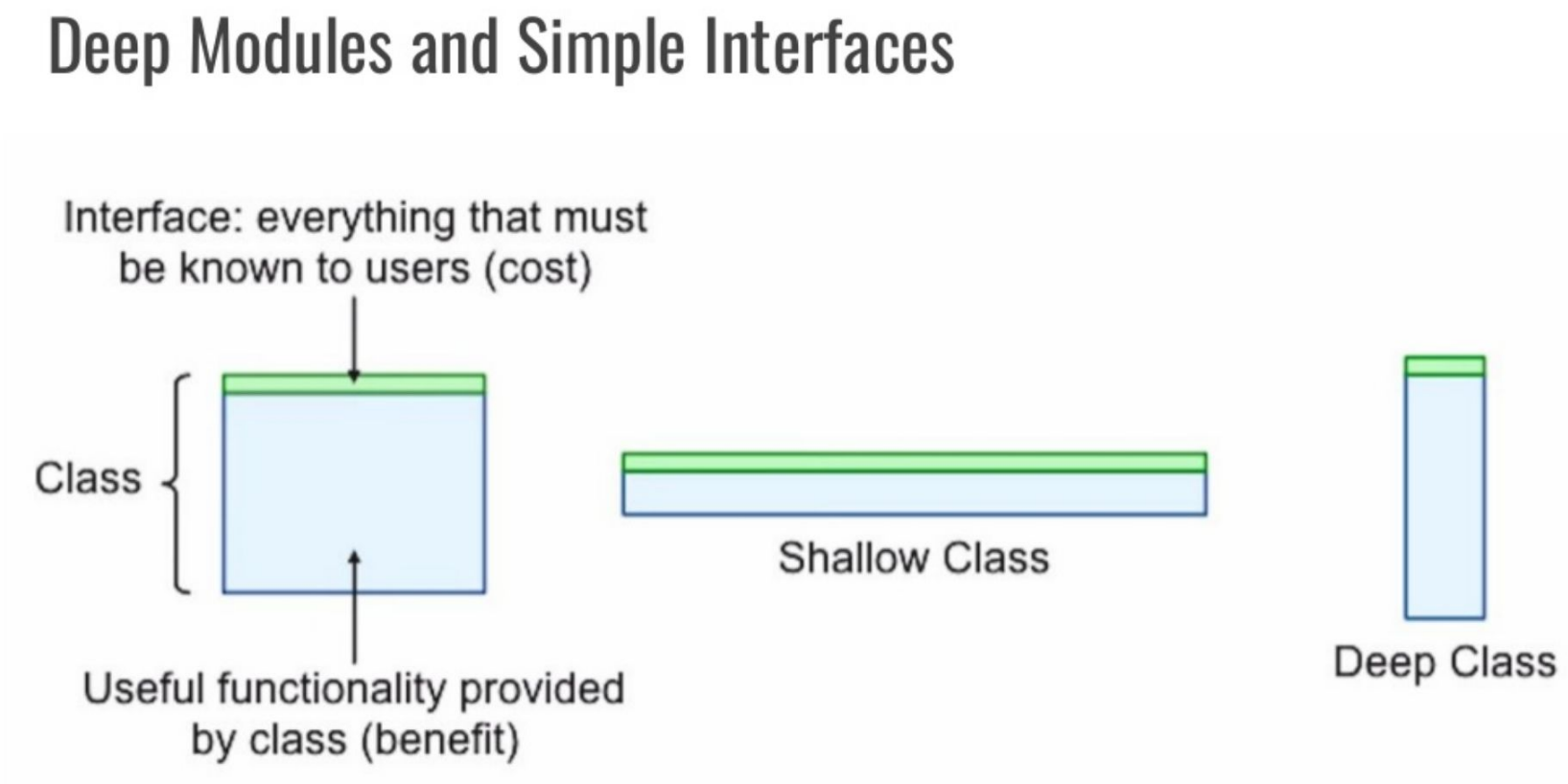
- Change Amplification
  - Change require change in many different places
- Cognitive Load
  - Sometimes more lines of code is better -- because is simple to understand.
- Unknown Unknowns
  - Not obvious places need to be modified for a change

Working code is not enough

- **Strategic programming**
  - Goal: produce a great design
  - Simplify future development
  - Minimize complexity
  - Must sweat the small stuff
- **Investment mindset**
  - Take extra time today
  - Pays back in the long run

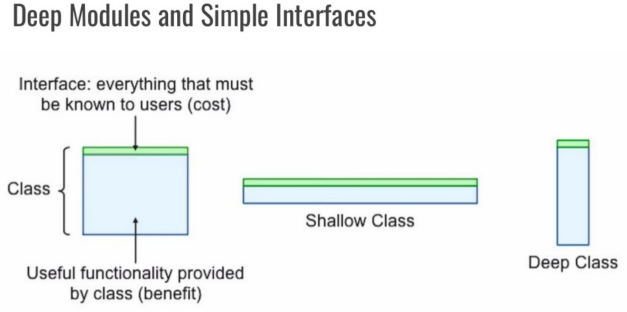


Deep Modules (opposite from clean-code "small")



Deep Modules & Simple Interfaces

- Modules should be Deep
- Classes are modules
- Classes should be Deep
- Classitis: Disease when you have too many classes.
- FAT Classes hide information
- When Interface is similar to impl means it's shallow



Pull Complexity Downwards

"It's more important to a module to have a simple interface than a simple implementation".

Obscurity

- A Great Source of Complexity
- Obscure code is:
  - Hard to Understand
  - Creates more Bugs
- Generic/Event-Drive makes code Obscure
- Solution to Obscurity
  - Always write obvious code
- Obvious Code Means:
  - Read Quickly without much thought
  - Easy to guess what code does
  - Guess Should be Write
  - Precise and meaningful

Code Review is the n#1 tool to fight Obscurity

Information Hiding and Leakage

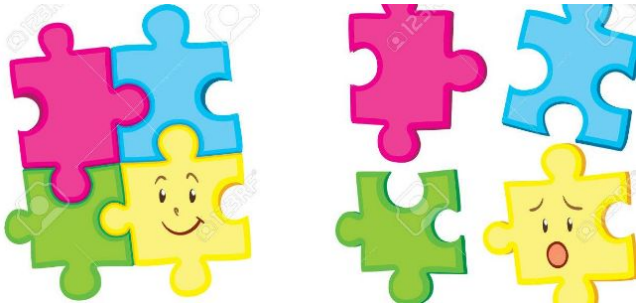
- Information Hiding
  - Encapsulate pieces of knowledge
- Information Package
  - Knowledge in multiples places/classes
- Red Flag - Overexposure
- Red Flag - Temporal decomposition - execution order is reflected on code structure. Operations that happen in different times are in different classes (results in Leakage).
- Too Many Classes
  - Divide the code in too many shallow classes (Leakage)

Different Layers Different Abstractions

- Red Flag: Pass-Through Method
  - Does nothing but pass args to other method.
- Decorators Sucks
  - Don't add much value
  - Easier to change the original implementation
  - Creates shallow modules

Better together or Better apart

- Bring Together:
  - IF Information is shared
  - IF will simplify the interface
  - To eliminate duplication
- Separate
  - General Purpose (1 generic code)
  - Specific Purpose
- Red Flag: Repetition (same/similar code over-and-over means you have wrong abstraction)
- Red Flag: Special-General Mixture (Leakage)



Each method should do one thing and do it completely

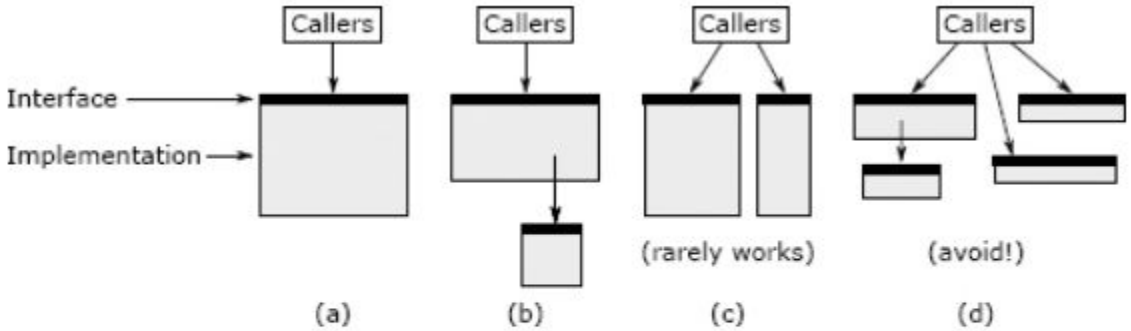


Figure 9.3: A method (a) can be split either by by extracting a subtask (b) or by dividing its functionality into two separate methods (c). A method should not be split if it results in shallow methods, as in (d).

Red Flag: Conjoined Methods (Need other methods to understand 1 method).

Define Errors out of Existence

- Exceptions add complexity and one of the worst sources of complexity
- Exception handling code is more difficult to write than normal code.
- A piece of code might found exceptions in different ways:
  - Caller may provide bad arguments or configuration
  - Invoked method may not be able to complete the request
  - In distributed systems network packets may be lost or delayed or peers communicate in unexpected ways
  - The code may detect bugs or internal inconsistencies or situations that are not prepared to handle.

Define Errors out of Existence

- Exception handling code creates more opportunities for Exceptions to happen.
- Language support for exception is verbose
- Too Many Exceptions
- Classes with lots of exceptions have complex interfaces
- Reduce the number of places that exception should be handled
- Best way to reduce bugs is to make code simpler

Writing Comments

- Writing comments correct will improve the design
- IF Users need to read method code to use it, there is no abstraction
- Comments that get out of date become misleading
- Comments should describe things that are not obvious on the code
  - Best Practices:
    - Pick Conventions
    - Don't repeat the code
    - Higher level comments enhance intuition
    - Delayed comments are bad comments
    - Use Comments as Design Tool

Naming

- Bad Names cause Bugs
- Create an image (name should create a image on mind of the reader about the nature of the thing being named).
- Names should be precise
- Use Names consistently
- Different Opinion Go Style Guide.
  - Short
  - Single character
  - Long names obscure what code does

Closing thoughts

- Getters / Setters are shallow ~ Avoid as much as possible
- Agile could easily lead to tactical programing
- TDD Focus on getting specific features working rather than have a better design.
- General Risk of Design Patterns == Over Application
- More Design Patterns does not make the design better

Design Principles & Red Flags

Summary of Design Principles

Here are the most important software design principles discussed in this book:

1. Complexity is incremental: you have to sweat the small stuff (see p. 11).
2. Working code isn't enough (see p. 14).
3. Make continual small investments to improve system design (see p. 15).
4. Modules should be deep (see p. 22)
5. Interfaces should be designed to make the most common usage as simple as possible (see p. 26).
6. It's more important for a module to have a simple interface than a simple implementation (see pp. 55, 71).
7. General-purpose modules are deeper (see p. 39).
8. Separate general-purpose and special-purpose code (see p. 62).
9. Different layers should have different abstractions (see p. 45).
10. Pull complexity downward (see p. 55).
11. Define errors (and special cases) out of existence (see p. 79).
12. Design it twice (see p. 91).
13. Comments should describe things that are not obvious from the code (see p. 101).
14. Software should be designed for ease of reading, not ease of writing (see p. 149).
15. The increments of software development should be abstractions, not features (see p. 154).

Summary of Red Flags

Here are a few of of the most important red flags discussed in this book. The presence of any of these symptoms in a system suggests that there is a problem with the system's design:

Shallow Module: the interface for a class or method isn't much simpler than its implementation (see pp. 25, 110).

Information Leakage: a design decision is reflected in multiple modules (see p. 31).

Temporal Decomposition: the code structure is based on the order in which operations are executed, not on information hiding (see p. 32).

Overexposure: An API forces callers to be aware of rarely used features in order to use commonly used features (see p. 36).

Pass-Through Method: a method does almost nothing except pass its arguments to another method with a similar signature (see p. 46).

Repetition: a nontrivial piece of code is repeated over and over (see p. 62).

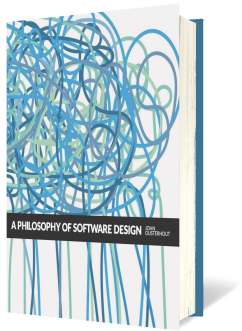
Special-General Mixture: special-purpose code is not cleanly separated from general purpose code (see p. 65).

Conjoined Methods: two methods have so many dependencies that its hard to understand the implementation of one without understanding the implementation of the other (see p. 72).

Comment Repeats Code: all of the information in a comment is immediately obvious from the code next to the comment (see p. 104).

Implementation Documentation Contaminates Interface: an interface comment describes implementation details not needed by users of the thing being documented (see p. 114).

Vague Name: the name of a variable or method is so imprecise that it doesn't convey much useful information (see p. 123).



A Philosophy of Software Design

Diego Pacheco