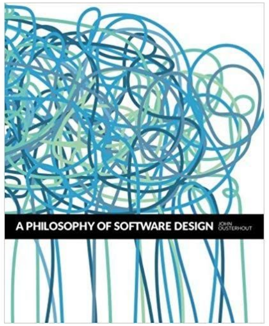
Software Design Notes



Diego Pacheco

About me...

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

diegopacheco

@diego_pacheco

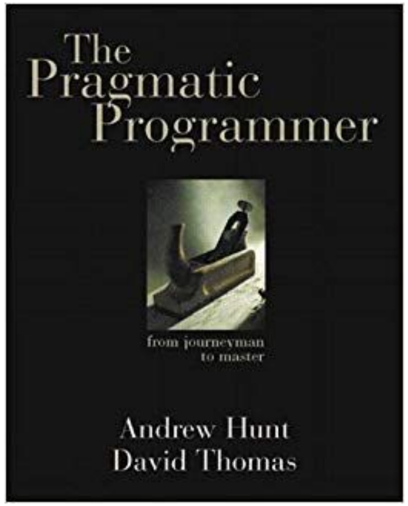
http://diego-pacheco.blogspot.com.br/

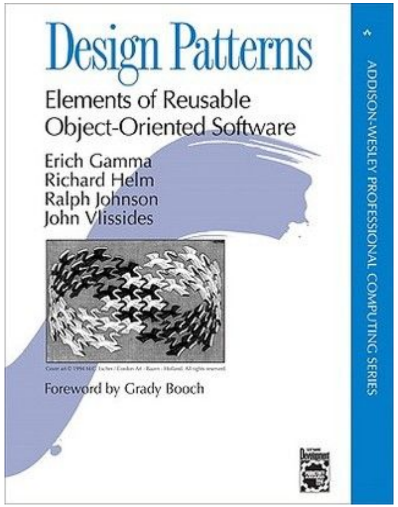
Diego Pacheco
Building Applications with Scala
Write modern, scalable, and reactive applications with the power of Scala
Packt>

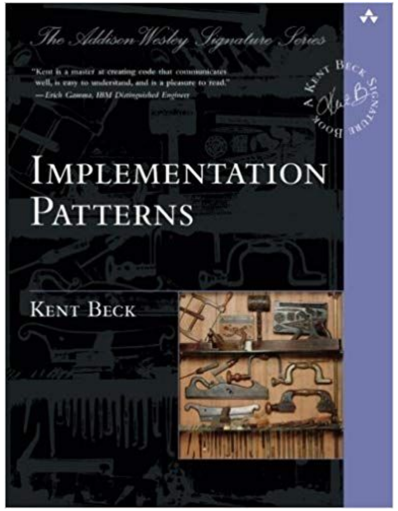
Diego Pacheco
Building Effective Microservices
Explore microservices and their implementation hands-on
Packt>

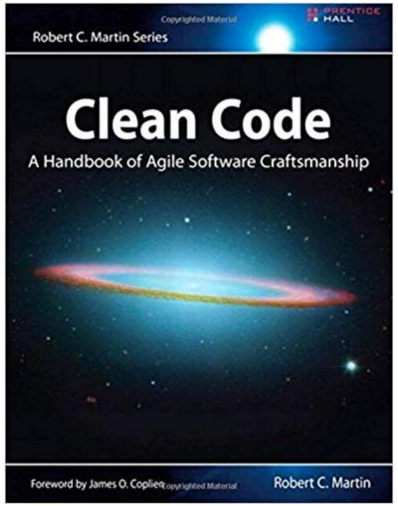
https://diegopacheco.github.io/

Software Design References (for me)

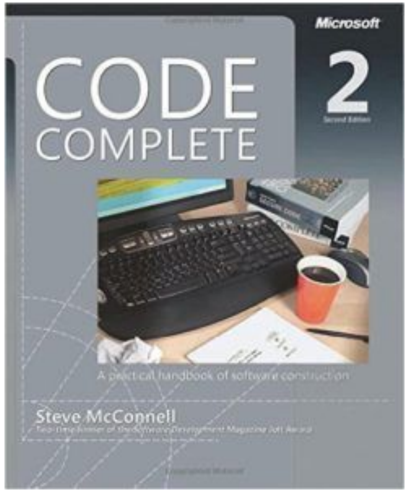


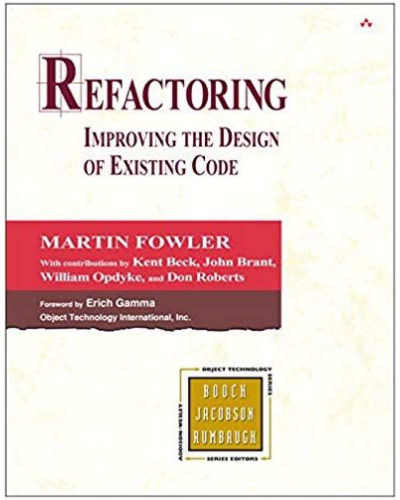

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

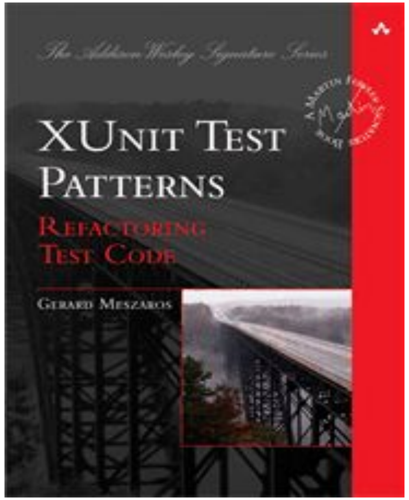

The Addison Wesley Signature Series
A Kent Beck Signature Book
Implementation Patterns
Kent Beck

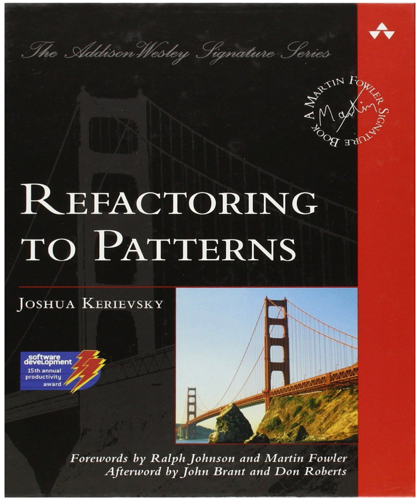

Robert C. Martin Series
Prentice Hall
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Software Design References (for me)

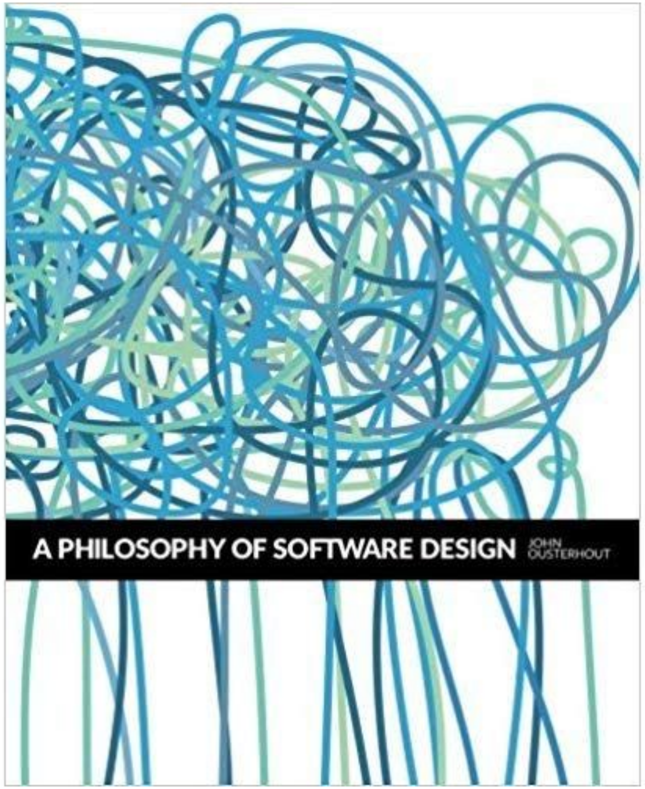



REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BOOCH JACOBSON RUMBAUGH



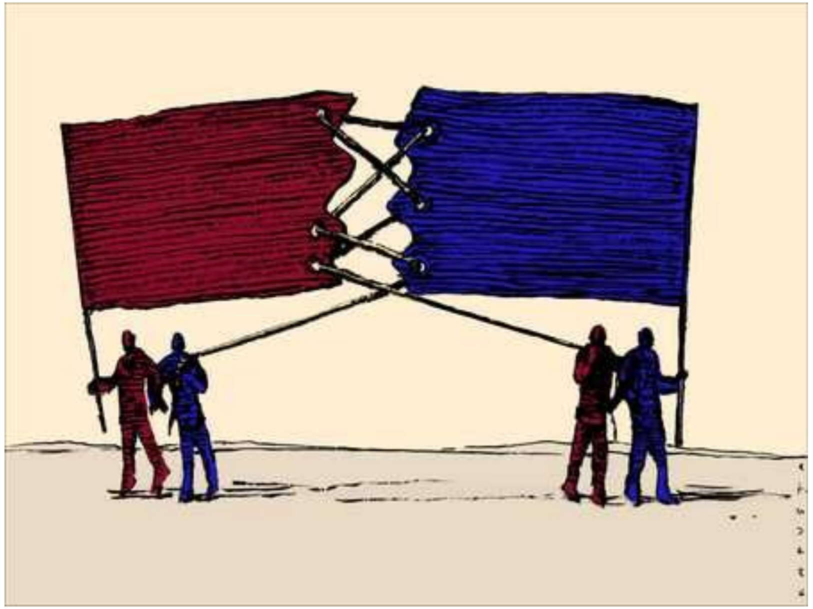

The Addison Wesley Signature Series
A Martin Fowler Signature Book
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

Best Software Design Book in long time(2018)



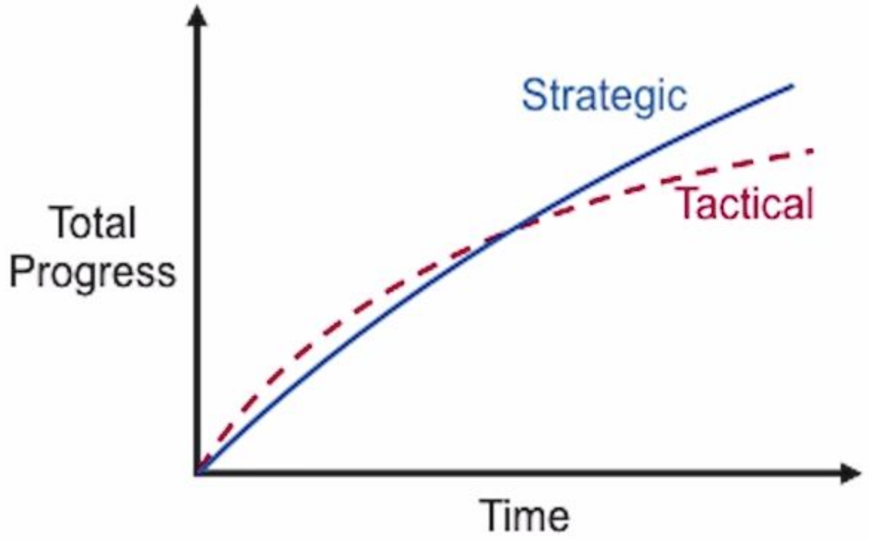
Movement vs Anti-Moviment

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

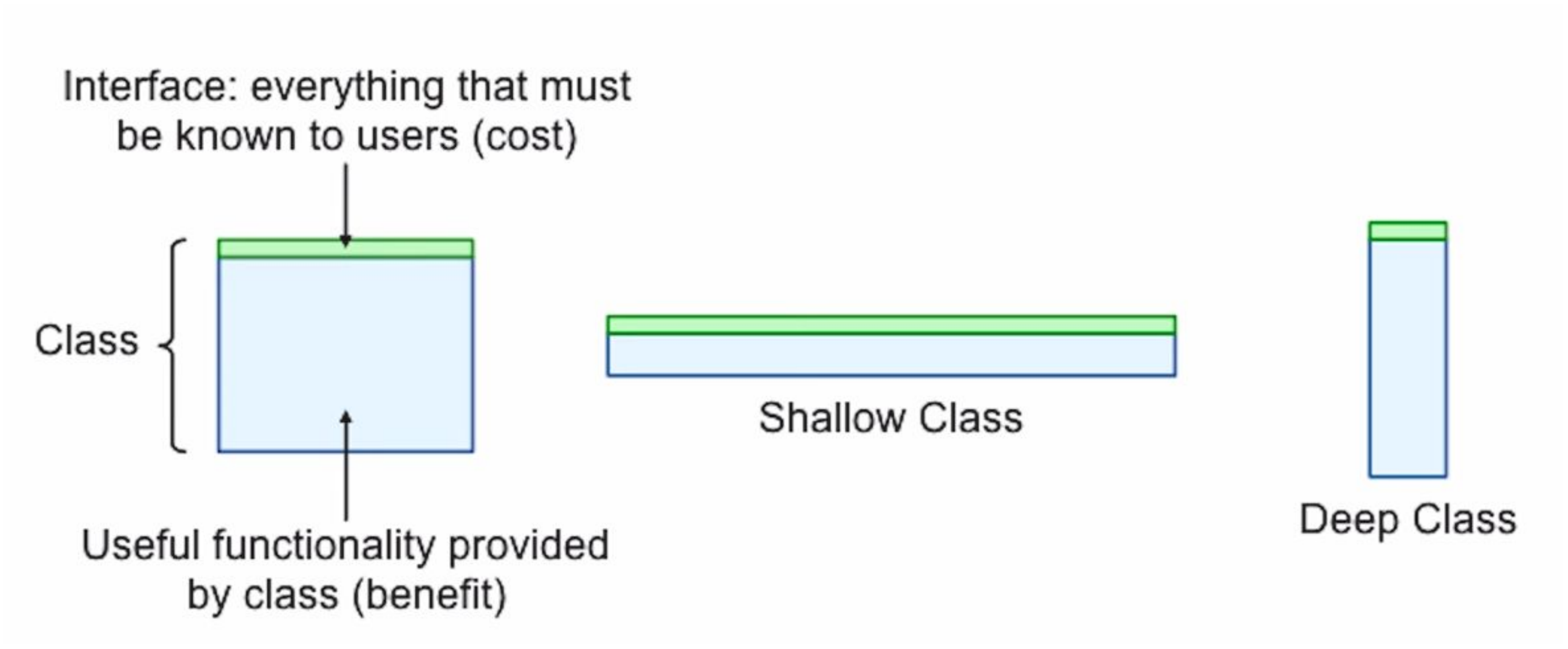


Strategic VS Tactical

- **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 and Simple Interfaces


Interface: everything that must be known to users (cost)
Class
Useful functionality provided by class (benefit)
Shallow Class
Deep Class

Notes

- Modules are Deep
- Classes are modules == Classes are Deep
- Deep Modules are better than shallow modules
- Classitis when you have too many classes disease
- FAT classes hide Information, therefore, creates abstractions
- When an interface is similar to implementation means its shallow thus leaking or poor.
- Decorator Pattern creates shallow modules - Therefore should be avoided.
- Global Context Variable instead of pass-through variables.
- Is more important to have a simple interface than a simple implementation
- Shallow/lots of private methods are also bad
- Exceptions are easy to throw but hard to handle
- Better avoid Exceptions as much as possible.

Notes

- Pull Complexity downward not upward(configuration) avoid exposing configs that won't change
- Comment what is not obvious
- Comments are part of the design
- Comment on interfaces should describe just how to use - this should not have comments of the impl.
- Comments need to be close to the code otherwise they will get outdated
- Long Variable Names VS Short Ones(Go prog Style)
- Obscurity is one of the main causes of complexity.
- Solution to obscurity is always written code that makes it obvious.
  - Obvious code means:
    - Read quickly without much thought
    - Easily guess meaning or what it does
    - Guess should be write

Notes

- If the Code is not obvious:
  - You spend lots of energy to understand.
  - So its hard to understand and its likely to increase misunderstanding, therefore, creating more bugs
- Code Review is the best tool to determine if the code is obvious
- Precise and meaningful names make the code more obvious.
- 2 Effect that's is important is consistency; Same names == Same Patterns easy to recognize things.
- Things that make the code less obvious:
  - event-driven code
  - generic objects == containers

Notes

- OOP = Composition over inheritance.

- Agile could easily lead to tactical programming.

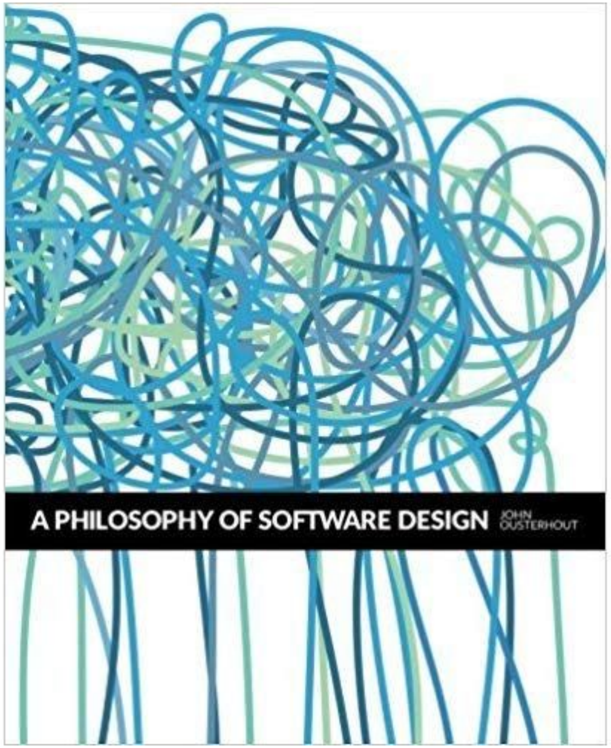
- TDD focus on getting some specific features working rather than have a better design.

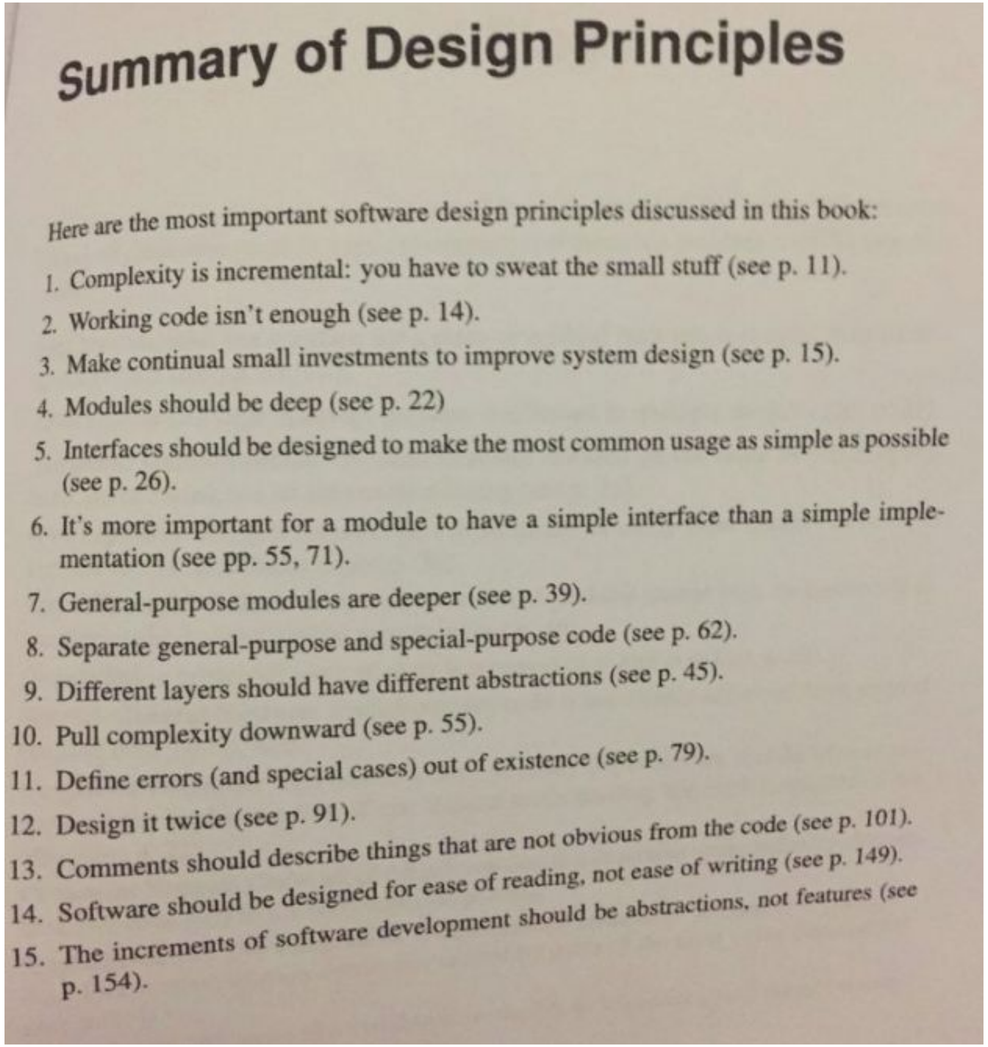
- Great risk of design patterns - over-application.

- More Design Patterns don't mean better design.

- Getters / Setters are shallow and should be avoided as much as possible.

Design Principles



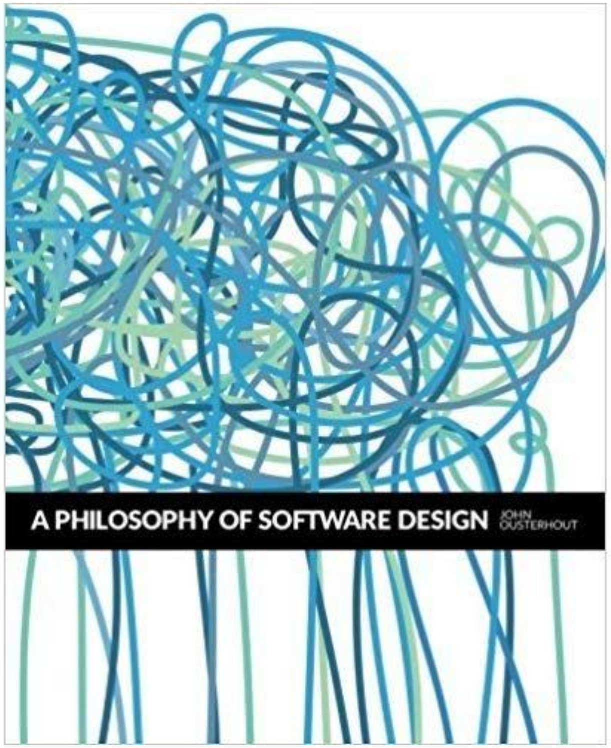


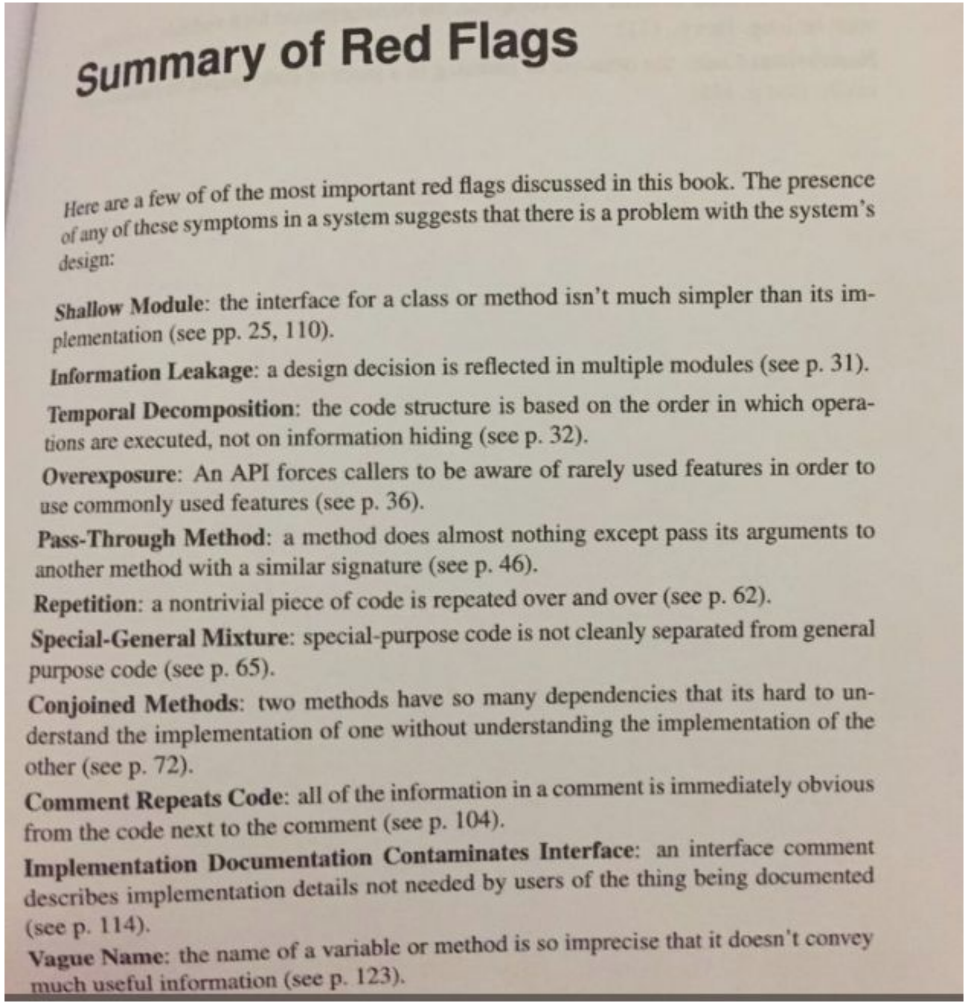
Summary of Design Principles

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).

Red Flags





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).

Closing Thoughts

- The Book is amazing.
- It's great that the book challenge ideas/assumptions.
- I'm not 100% into Comments.
- The Book don't cover Functional programing in detail :(
- The Book don't cover DevOps Engineering :(
- Some sample examples are not practical: Text Editor.

Software Design Notes

Diego Pacheco