A PHILOSOPHY OF SOFTWARE DESIGN

JOHN OUSTERHOUT

The overall goal of design is to reduce complexity

A good design fight complexity by making a system obvious

WHAT IS COMPLEXITY ?

Complexity in a software is anything that make it

Hard to understand OR Hard to modify

Dependencies, code that can't be understand / modified in isolation

Obscurity, any nonobvious info

AND

Cause complexity

Complexity manifest itself through

Change amplification, simple change require lots of modifications

Unknown unknowns, anything not obvious or hidden

Cognitive load, a dev need to know a lot complete a task

HOW DOES COMPLEXITY HAPPEN ?

Complexity isn't caused by a single catastrophic error; it accumulates in lots of small chunks

Strategic programming

VS

Invest in your software

- Try multiple design
- Take time to fix design problems when discovered
- Take 10%-20% dev time on investments

Invest & design incrementally

Tactical programming

Code as quickly as possible

Tactical tornado developer

- Implements feature really quick
- Pumps out code faster than anyone else
- Rest of the team clean slowly after the tornado

Evolution of Facebook's Motto

Before 2014 "*Move fast and break things*"

After 2014 "*Move fast with solid infrastructure*"

DESIGN SIMPLE SYSTEMS

Deep modules

In modular design, the software is splitted into modules, relatively independant

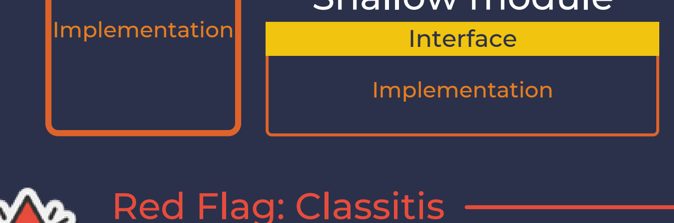
A module (class, subsystem, function, ...) is composed of two parts :

An interface *What*
Everything a developer using the module must know in order to use it

An implementation *How*
The code that carries out the promises made by the interface

Interface includes uninportant details
Cognitive load

Interface omits important details
Obscurity



A module should compensate the complexity (its interface) it brings with the functionality it offers (implementation).

Red Flag: Classitis
"Classes should be small": minimize the amount of functionality in each new class. Creates a lot of shallow classes, increasing the complexity of the overall system

Information Hiding

Module encapsulates (hides) pieces of knowledge, representing design decisions

- Simplify the interface
- Erase outside dependencies on those information

Red Flag: Information leakage || Temporal decomposition
The same knowledge is used in multiple places. It may happen when the execution order is reflected in the code structure, the same knowledge is used at different points in execution.

Hiding might be situational

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

Red Flag: Overexposure
API for a commonly used feature forces users to learn about other features that are rarely used (increase cognitive load)

FIGHT EXISTING COMPLEXITY

You can fight complexity by

Eliminating it OR

Encapsulating it away
Isolating complexity where it will never be seen is almost as good as eliminating the complexity entirely

General purpose modules

Deeper modules: interface should be general enough to support multiple uses

What is the simplest interface that will cover all my current needs ?

In how many situations will this method be used ?

Is this API easy to use for my current needs?

Different layer Different abstraction

Systems contains layers, where higher layers use functionnalities provided by lower layers

Red Flag: Pass-Through Method
A method that does nothing except pass its arguments to another method. It indicates that there is not a clean division of responsibility between classes

How to remove pass-through, decorator, interface duplication...

- Add the new functionality directly to the underlying class
- If the new functionality is specialized for a particular use case, merge it with the use case
- Merge the functionality with an existing decorator rather than creating one
- Implement the functionnality as a stand-alone class

> "An element must eliminate some complexity that would be present in the absence of the design element"

If different layers have the same abstraction, they may not provide enough benefit to compensate for the additional infrastructure they represent

Pull complexity Downwards

Tactical programming: solve the easy problems and punt the hard ones to someone else

Strategic programming: It is more important for a module to have a simple interface than a simple implementation

Ex: **Configuration parameters**, moving complexity upwards
Will users (or higher-level modules) be able to determine a better value than we can determine in the module ?
↪ Default values ↪ Dynamic auto-configuration

Together or Appart

Bring together if

- Information is shared
- It will simplify the interface (temporal decomposition)
- It eliminate duplication, except if
  - the snippet is only one or two lines long
  - the new method require a complex signature

Separate if

- It mix general-purpose and special-purpose code
- It results in cleaner abstractions, overall

Define Errors Out Of Existence

> "Code that deals with special conditions is inherently harder to write than code that deals with normal case"

90%+ of catastrophic failures in distributed data-intensive systems were caused by incorrect error handling

Limit exception handling complexity by

1.Removing exception
Define API so that there is no exception (ex: file deletion in Windows vs Unix)

2.Masking exception
Handle it at lower level (pull down) if not needed outside

3.Aggregating exception
Replace multiple special-purpose handling with one high level general-purpose handling

4.Just Crashing
Meaningful action on "out of memory" error ?

5.Removing special case
Ex: no user selection, not represented has is in code

Meaningful comments

Developers should be able to understand a module without reading any code other than its externally visible declarations

Low level comments add precision

- Unit of variable
- Inclusive/exclusive boundary ?
- What null implies

High level comments enhance intuition

- What is the most important thing about this code ?
- What is this code trying to do ?

The best way to ensure that comments are useful and get updated is to position them close to the code they describe

Red Flag: Comment Repeats Code
If the information in a comment is already obvious from the code next to the comment, then the comment isn't helpful

Pick conventions ensure that you actually write comments

- **Interface**: before a module (class, function, method,...)
- **Data structure member**: next to the declaration of a field
- **Implementation**: comment inside the code of a method
- **Cross-module**: describe dependencies that cross module boundaries.

> "Every class should have an interface comment, every class variable should have a comment, and every method should have an interface comment"

Explicit naming

Good names are a form of documentation, they make code easier to understan. They create an image in the mind of the reader about the nature of the thing being named

Red Flag: Vague Name
If a name is broad enough to refer to many different things, then it doesn't convey much informationand the entity is more likely to be misused.

Red Flag: Hard to Pick Name
If it's hard to find a simple name creates a clear image of the underlying object, that's a hint that the underlying object may not have a clean design

Be consistent

Consistency creates **cognitive leverage**

Once you have learned how something is done in one place, you can use that knowledge to immediately understand other places that use the same approach.

Examples of consistency

- Names
- Coding style
- Interface with multiple implementation
- Using design pattern

Consistency is hard to maintain

Document standards Enforce standards

Make the code obvious

"Obvious" is in the mind of the reader. Best way to determine the obviousness of code is through code reviews

Previous points make the code more obvious

Less obvious code

Event-driven programming
Obscurate the flow of control. Compensate with comments

Generic containers
Ex: tuple. Obscure the element meaning

Code that violates reader expectations

> "Software should be designed for ease of reading, not ease of writing"

BE A BETTER DESIGNER

Design it twice

Consider multiple options for each major design

- Does one alternative have a simpler interface ?
- Is one interface more general-purpose ?
- Does one interface enable a more efficient implementation ?

This habit also improves your design skills

Write comments first

Writing the comments first improves the system design

Iterate over interface and signatures comments until the basic structure feels about right

It makes writing documentation part of the design process

DESIGNING FOR PERFORMANCE

Simpler code tends to run faster than complex code.

Use basic knowledge of performance to choose design alternatives that are "naturally efficient" yet also clean and simple

Measure before modifying

Programmers' intuitions about performance are unreliable

- Benchmark classic structures and functions to enhance knowledge
- Measure app performance, identify places with biggest impact
- First measurement provides a baseline, for comparaison

Design around the critical path

Best solution: "fundamental" change

Different algorithm, add cache, ...

Last resort: redesign

Consider only the critical path

- Minimum amount of code
- Only the data needed
- Most convenient data structure

Remove special cases from critical case. Ideally, have a single if at the beginning, handling all special cases