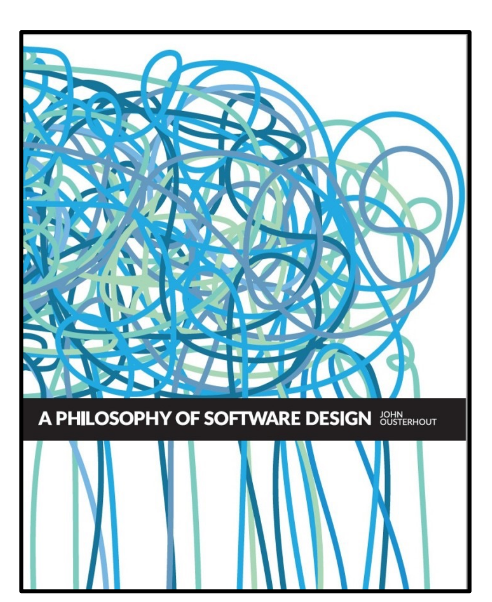
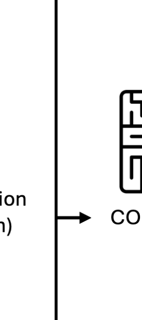
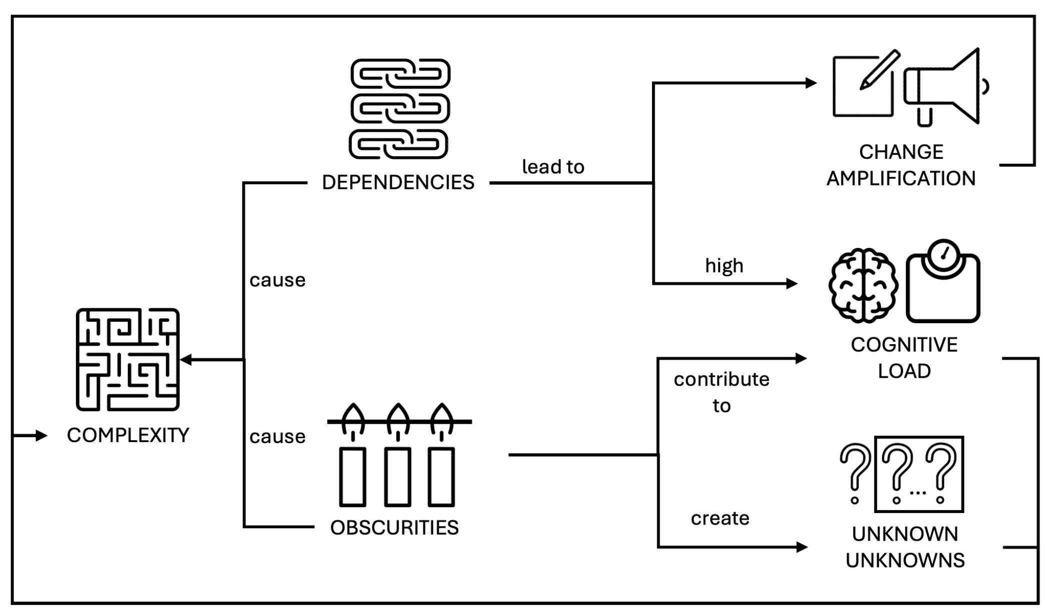
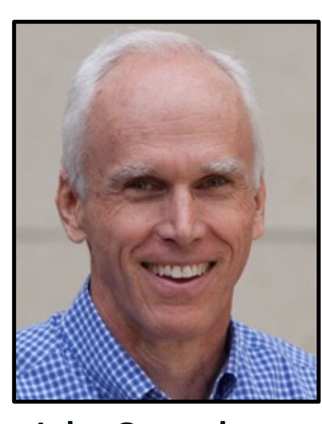
The Nature of Complexity in John Ousterhout's Philosophy of Software Design

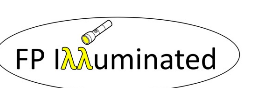








slides by @philip_schwarz

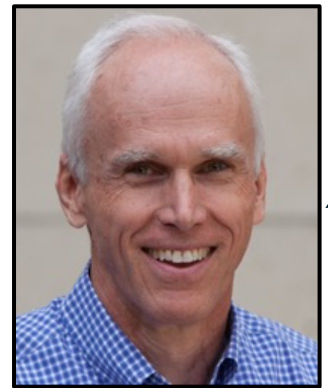


https://fpilluminated.org/

@philip_schwarz

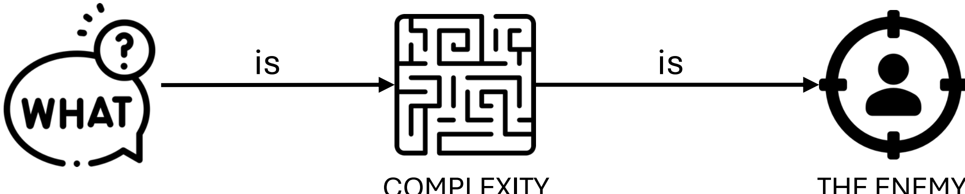
I hope **John Ousterhout** will forgive me for conveying to you, so extensively, what he has to say (in the **second chapter** of his **book**) about the **nature of complexity**, and I hope that the **graphical approach** that I have taken to **capturing** *some* of his **ideas**, might encourage anyone who hasn't already done so, to read the book.

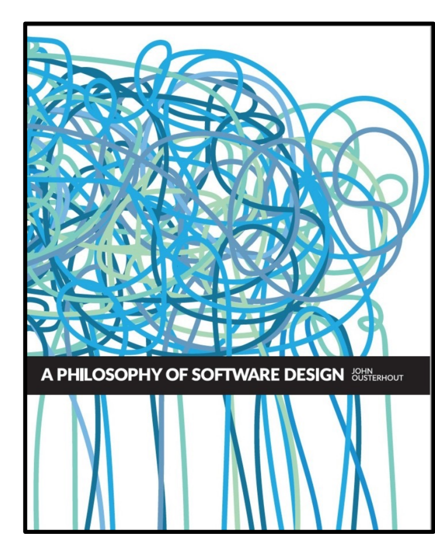
This deck was designed using resources from Flaticon.com flaticon

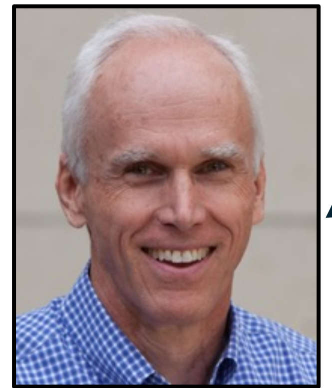


John Ousterhout

This book is about how to **design** **software systems** to **minimize** their **complexity**. The **first step** is to **understand** the **enemy**. Exactly what is **"complexity"**?

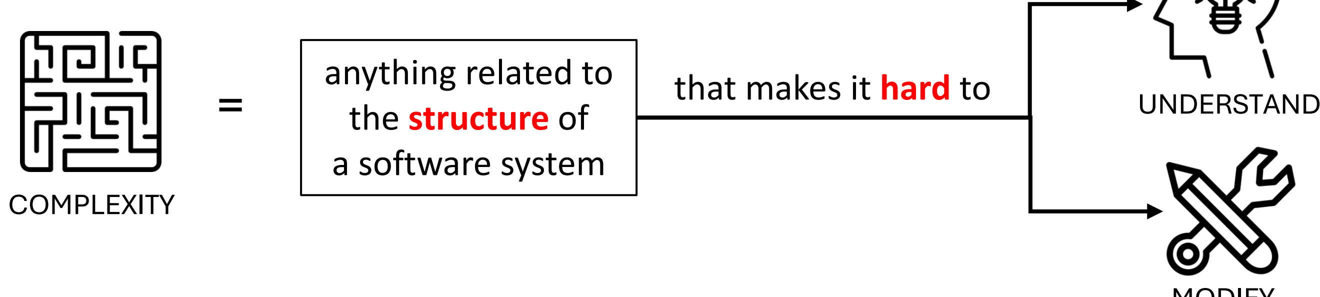


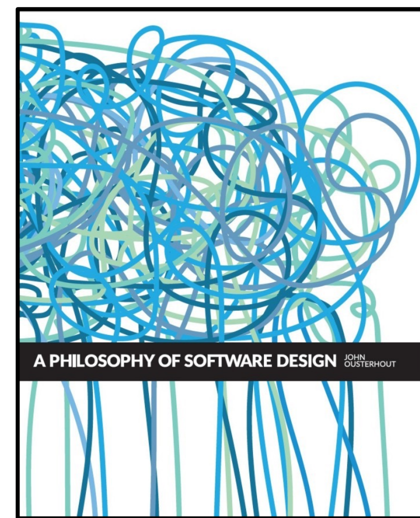


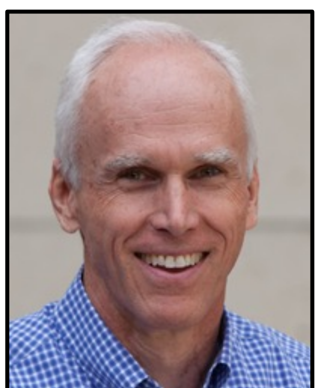


John Ousterhout

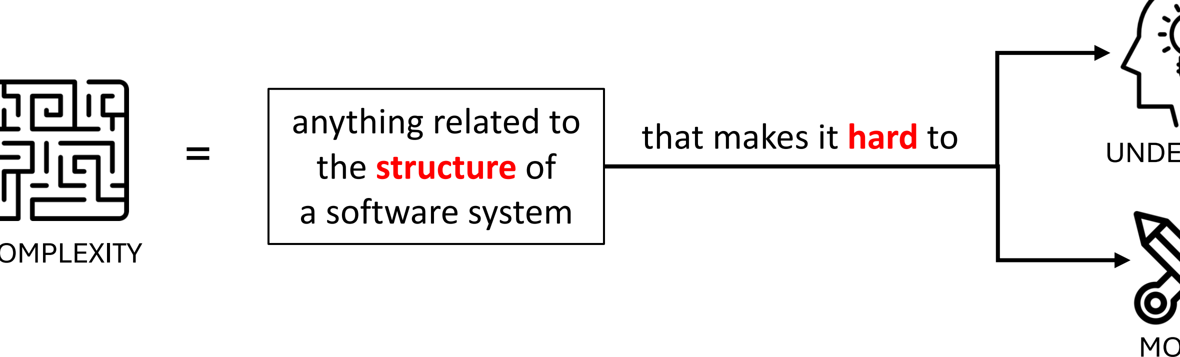
Complexity is anything related to the **structure** of a **software system** that makes it **hard** to **understand** and **modify** the **system**.

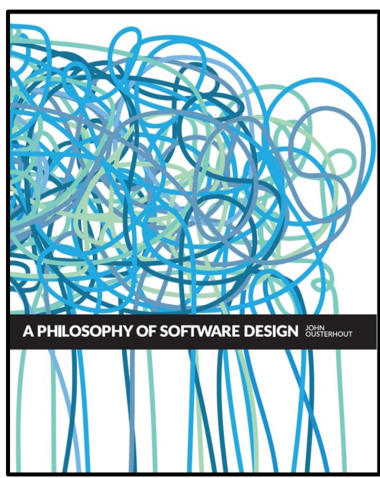






John Ousterhout





Robert Martin

Every **software module** has **three functions**.

First is the function it performs while executing.

This function is the reason for the module's existence.

The second function of a module is to afford change.

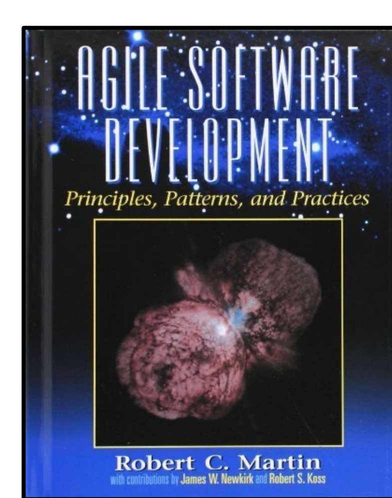
Almost all modules will change in the course of their lives, and it is the responsibility of the developers to make sure that such changes are as simple as possible to make.

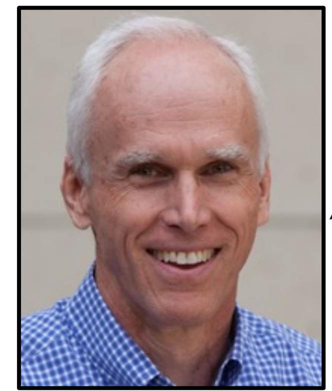
A module that is difficult to change is broken and needs fixing, even though it works.

The third function of a module is to communicate to its readers.

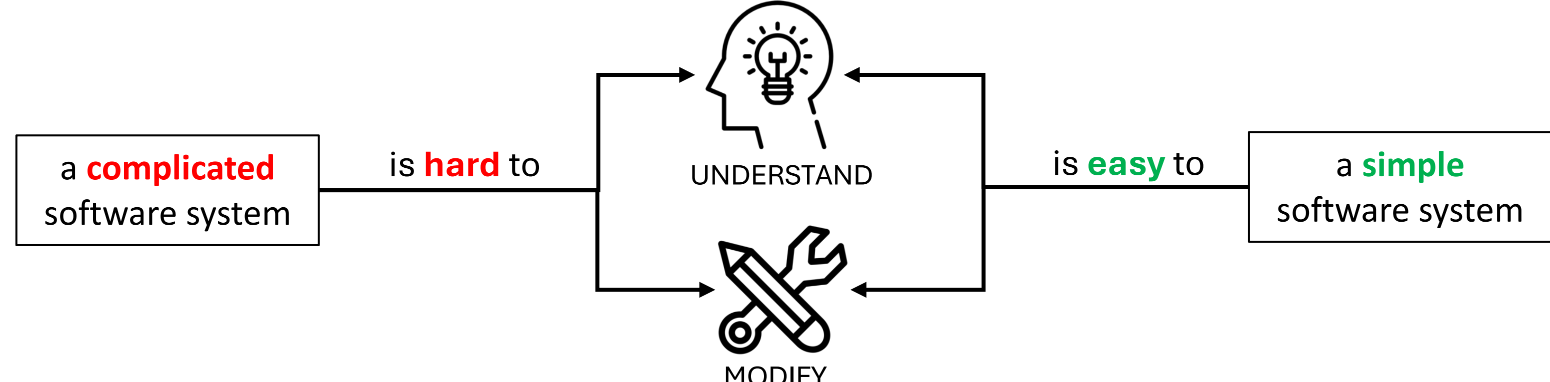
Developers who are not familiar with the module should be able to read and understand it without undue mental gymnastics.

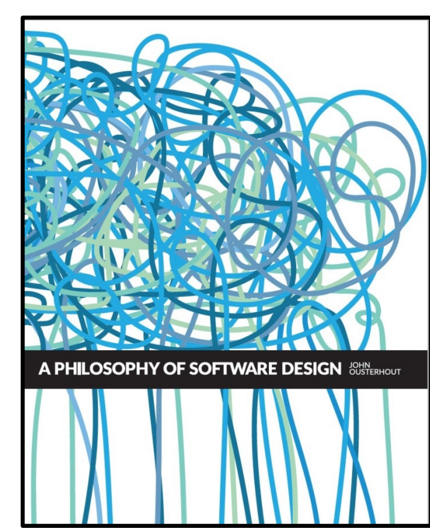
A module that does not communicate is broken and needs to be fixed.





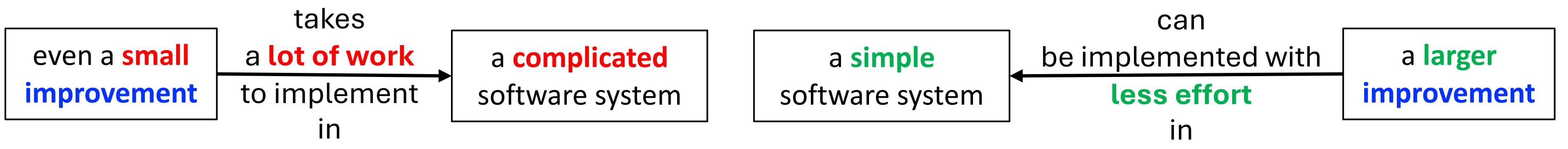
If a **software system** is **hard** to **understand** and **modify**, then it is **complicated**; if it is **easy** to **understand** and **modify** then it is **simple**.

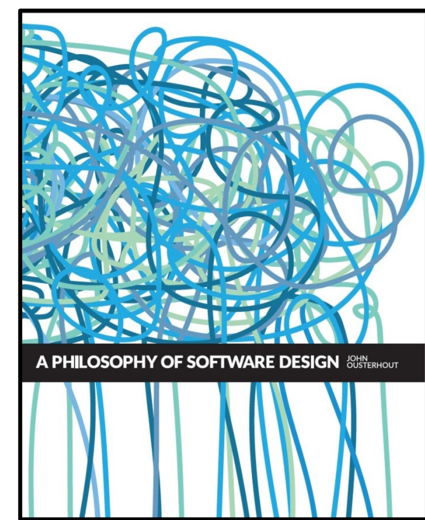


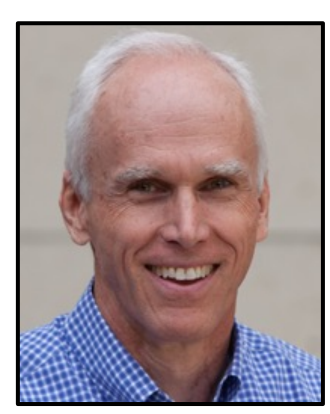


John Ousterhout

You can also think of **complexity** in terms of **cost** and **benefit**. In a **complex system**, it takes a lot of **work** to **implement** even **small improvements**. In a **simple system**, **larger improvements** can be **implemented** with **less effort**.







John Ousterhout

Complexity is what a developer experiences at a particular point in time when trying to achieve a particular goal. It doesn't necessarily relate to the overall size or functionality of the system.

…

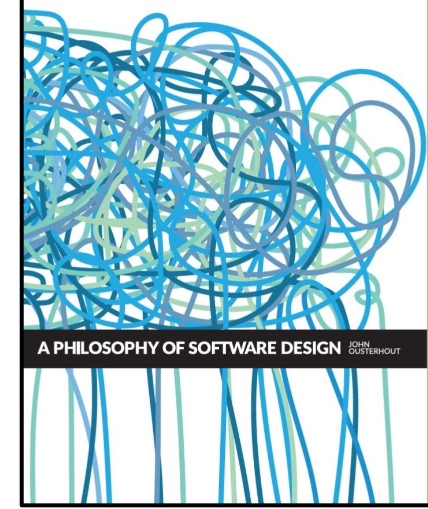
Complexity is determined by the activities that are most common. If a system has a few parts that are very complicated but those parts almost never need to be touched, they don't have much impact on the overall complexity of the system.

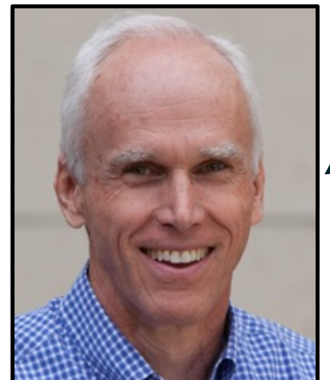
…

Isolating **complexity** in a place where it will never be seen is almost as good as eliminating the **complexity** entirely.

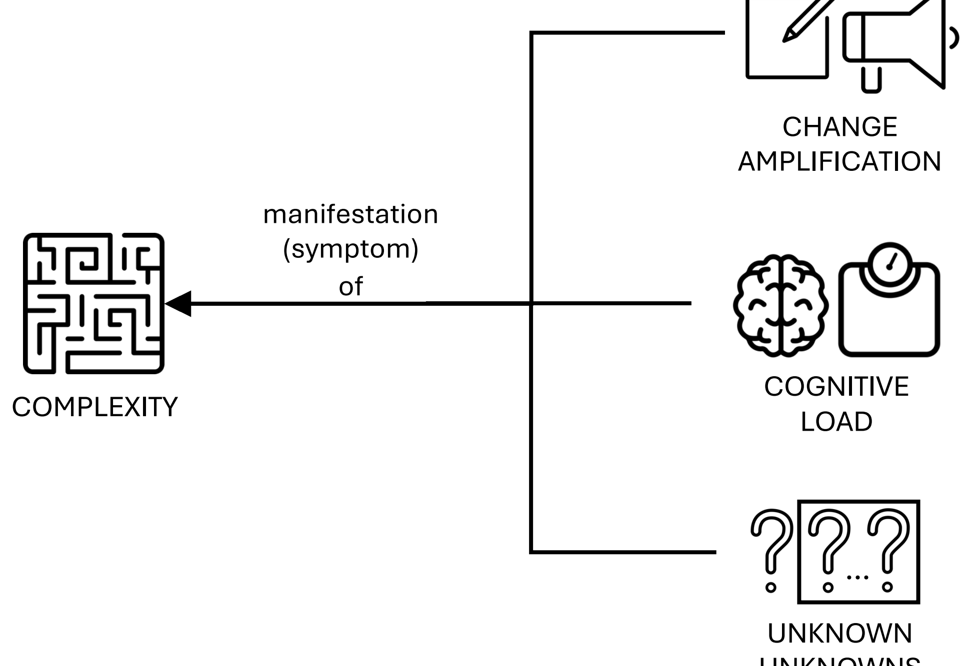
Complexity is more apparent to readers than writers.

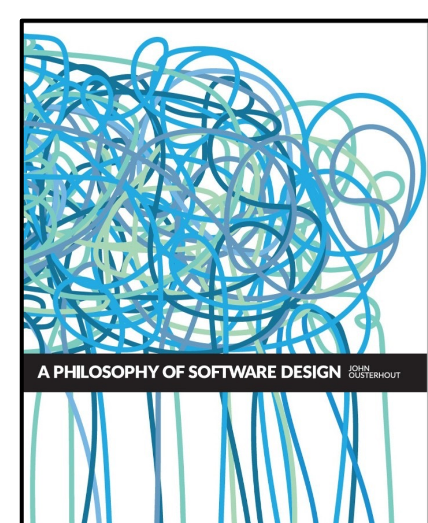
…

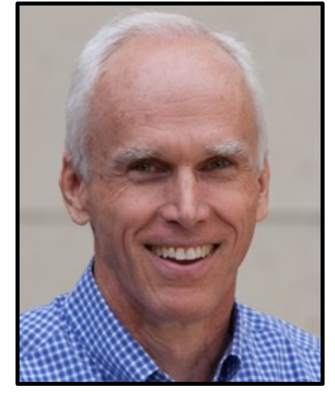



John Ousterhout

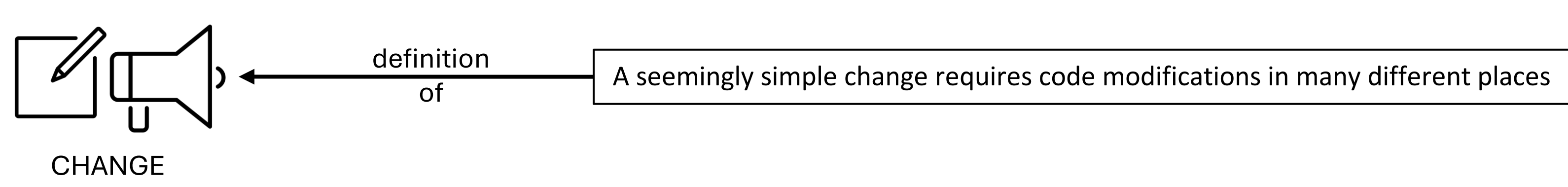
Complexity manifests itself in three general ways... Each of these manifestations makes it harder to carry out development tasks.

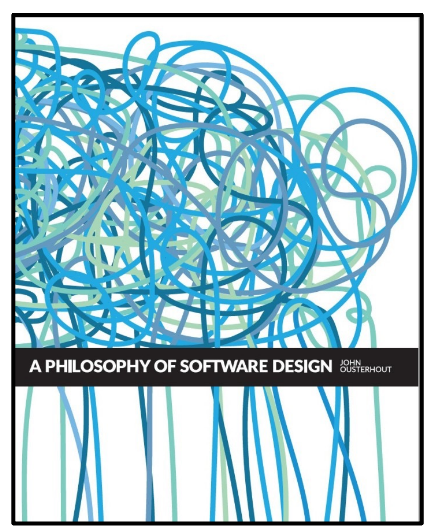


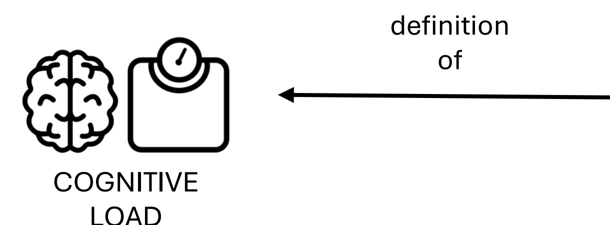




John Ousterhout

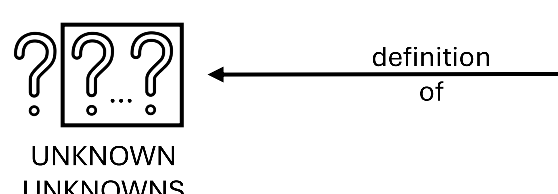






How much a developer needs to know in order to complete a task. A higher cognitive load means that developers have to spend more time learning the required information, and there is a greater risk of bugs because they have missed something.

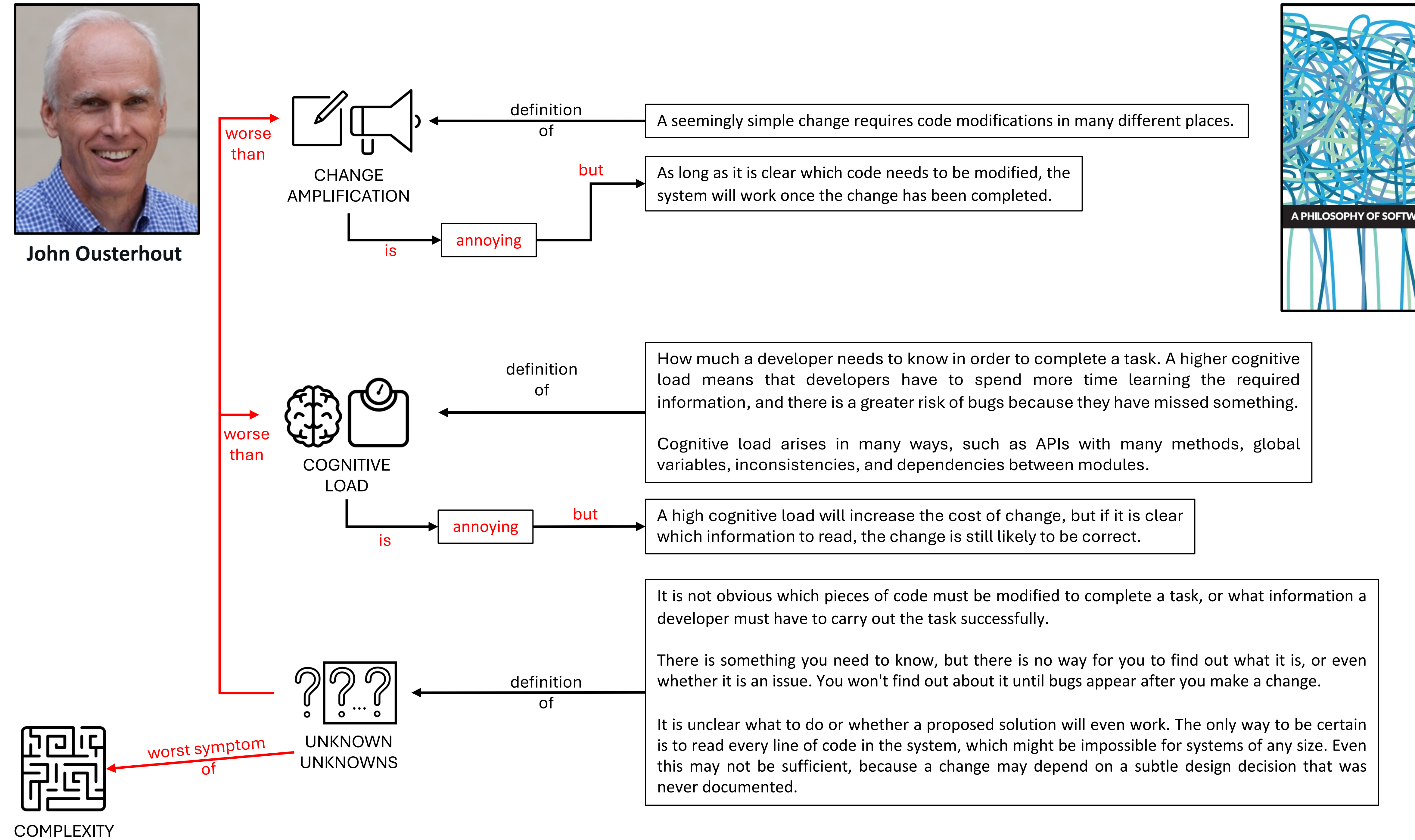
Cognitive load arises in many ways, such as APIs with many methods, global variables, inconsistencies, and dependencies between modules.

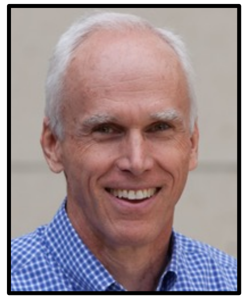


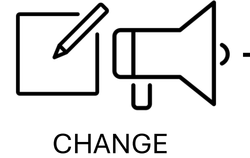
It is not obvious which pieces of code must be modified to complete a task, or what information a developer must have to carry out the task successfully.

There is something you need to know, but there is no way for you to find out what it is, or even whether it is an issue. You won't find out about it until bugs appear after you make a change.

It is unclear what to do or whether a proposed solution will even work. The only way to be certain is to read every line of code in the system, which might be impossible for systems of any size. Even this may not be sufficient, because a change may depend on a subtle design decision that was never documented.

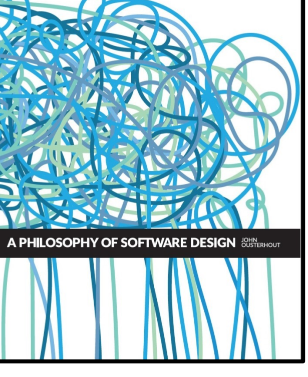

John Ousterhout
A PHILOSOPHY OF SOFTWARE DESIGN JOHN OUSTERHOUT
worse than
CHANGE AMPLIFICATION
definition of
A seemingly simple change requires code modifications in many different places.
is
annoying
but
As long as it is clear which code needs to be modified, the system will work once the change has been completed.
worse than
COGNITIVE LOAD
definition of
How much a developer needs to know in order to complete a task. A higher cognitive load means that developers have to spend more time learning the required information, and there is a greater risk of bugs because they have missed something.
Cognitive load arises in many ways, such as APIs with many methods, global variables, inconsistencies, and dependencies between modules.
is
annoying
but
A high cognitive load will increase the cost of change, but if it is clear which information to read, the change is still likely to be correct.
UNKNOWN UNKNOWNS
definition of
It is not obvious which pieces of code must be modified to complete a task, or what information a developer must have to carry out the task successfully.
There is something you need to know, but there is no way for you to find out what it is, or even whether it is an issue. You won't find out about it until bugs appear after you make a change.
It is unclear what to do or whether a proposed solution will even work. The only way to be certain is to read every line of code in the system, which might be impossible for systems of any size. Even this may not be sufficient, because a change may depend on a subtle design decision that was never documented.
worst symptom of
COMPLEXITY



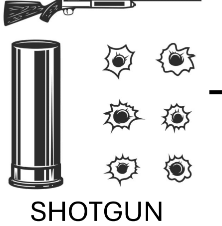


CHANGE
AMPLIFICATION

A seemingly simple change requires code modifications in many different places.



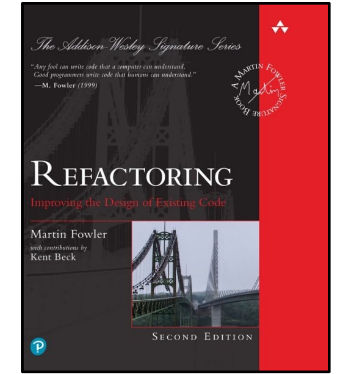
Martin Fowler

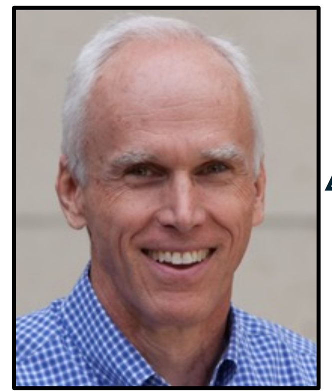


SHOTGUN
SURGERY

Shotgun Surgery

Shotgun surgery is similar to divergent change but is the opposite. You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.





John Ousterhout

One of the most important goals of good design is for a system to be obvious. This is the opposite of high cognitive load and unknown unknowns. In an obvious system, a developer can quickly understand how the existing code works and what is required to make a change. An obvious system is one where developer can make a quick guess about what to do, without thinking very hard, and yet be confident that the guess is correct.

GOOD DESIGN

one
of the most important goals
of

OBVIOUS

OBVIOUS SYSTEM

the
opposite

of high

COGNITIVE LOAD

of

UNKNOWN UNKNOWNS

definition | of

In an obvious system, a developer can quickly understand how the existing code works and what is required to make a change.

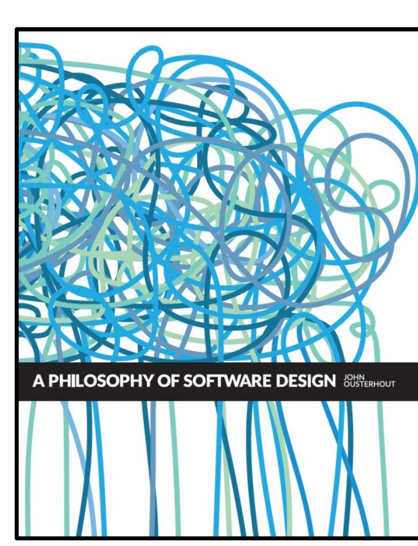
An obvious system is one where a developer can make a quick guess about what to do, without thinking very hard, and yet be confident that the guess is correct.

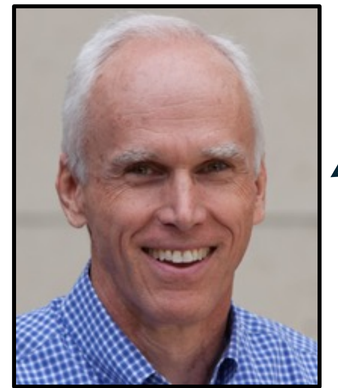
one
of the goals
of

CHANGE AMPLIFICATION

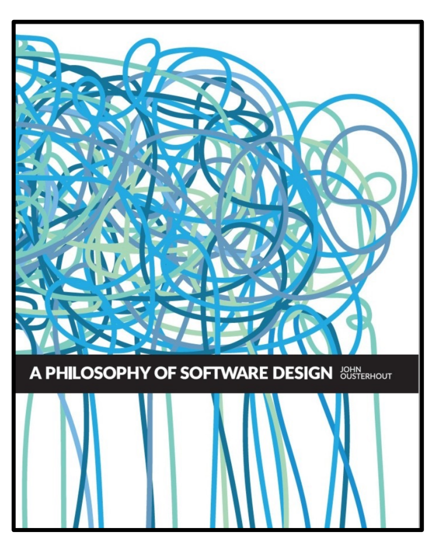
Reduce the amount of code that is affected by each design decision, so design changes don't require very many code modifications.

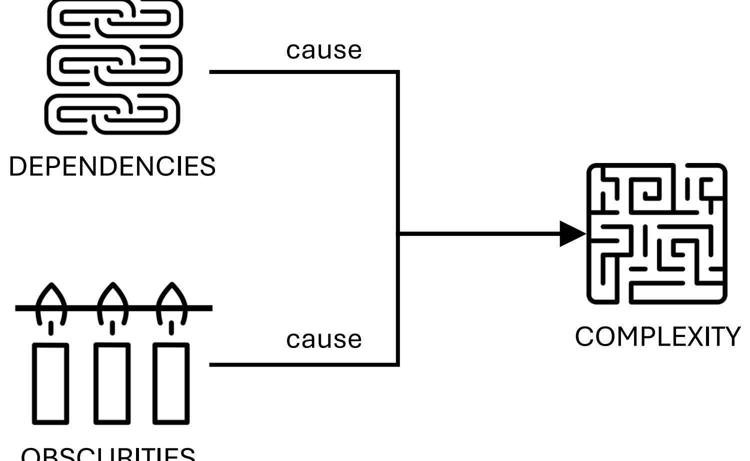
i.e. reduce

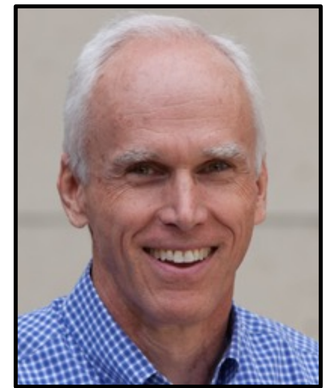



John Ousterhout

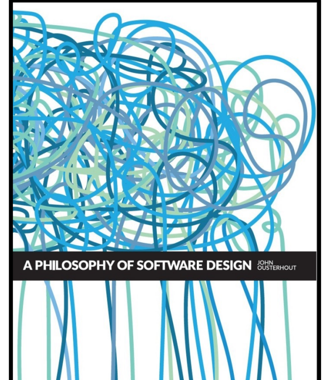
Complexity is caused by two things: **dependencies** and **obscurity**.







John Ousterhout



DEPENDENCIES

A dependency exists when a given piece of code cannot be **understood** and **modified** in isolation; The code relates in some way to other code and the other code must be **considered** and/or **modified** if the given code is changed.

Dependencies are a fundamental part of software and can't be completely eliminated. In fact, we intentionally introduce dependencies as part of the software design process.

However, one of the goals of software design is to reduce the number of dependencies and to make the dependencies that remain as **simple** and **obvious** as possible.

often associated with

OBSCURITIES

Obscurity occurs when **information** is **not obvious**.

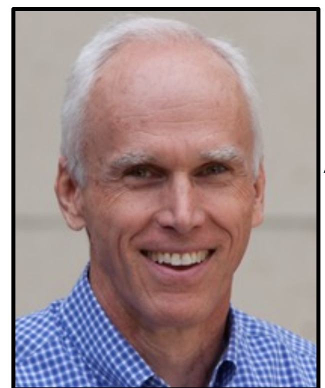
The best way to reduce obscurity is by **simplifying** the **system design**.

Obscurity is often associated with **dependencies**, where it is not **obvious** that a dependency exists.

Inconsistency is also a major contributor of obscurity.

In many cases, obscurity comes about because of **inadequate documentation**.

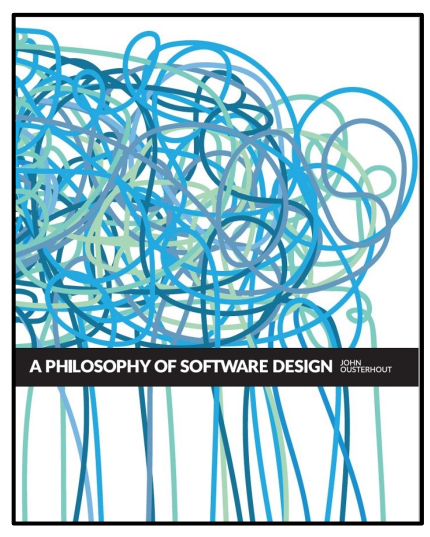
However, obscurity is also a design issue. If a system has a clean and obvious design, then it will need less documentation. The need for extensive documentation is often a red flag that the design isn't quite right.

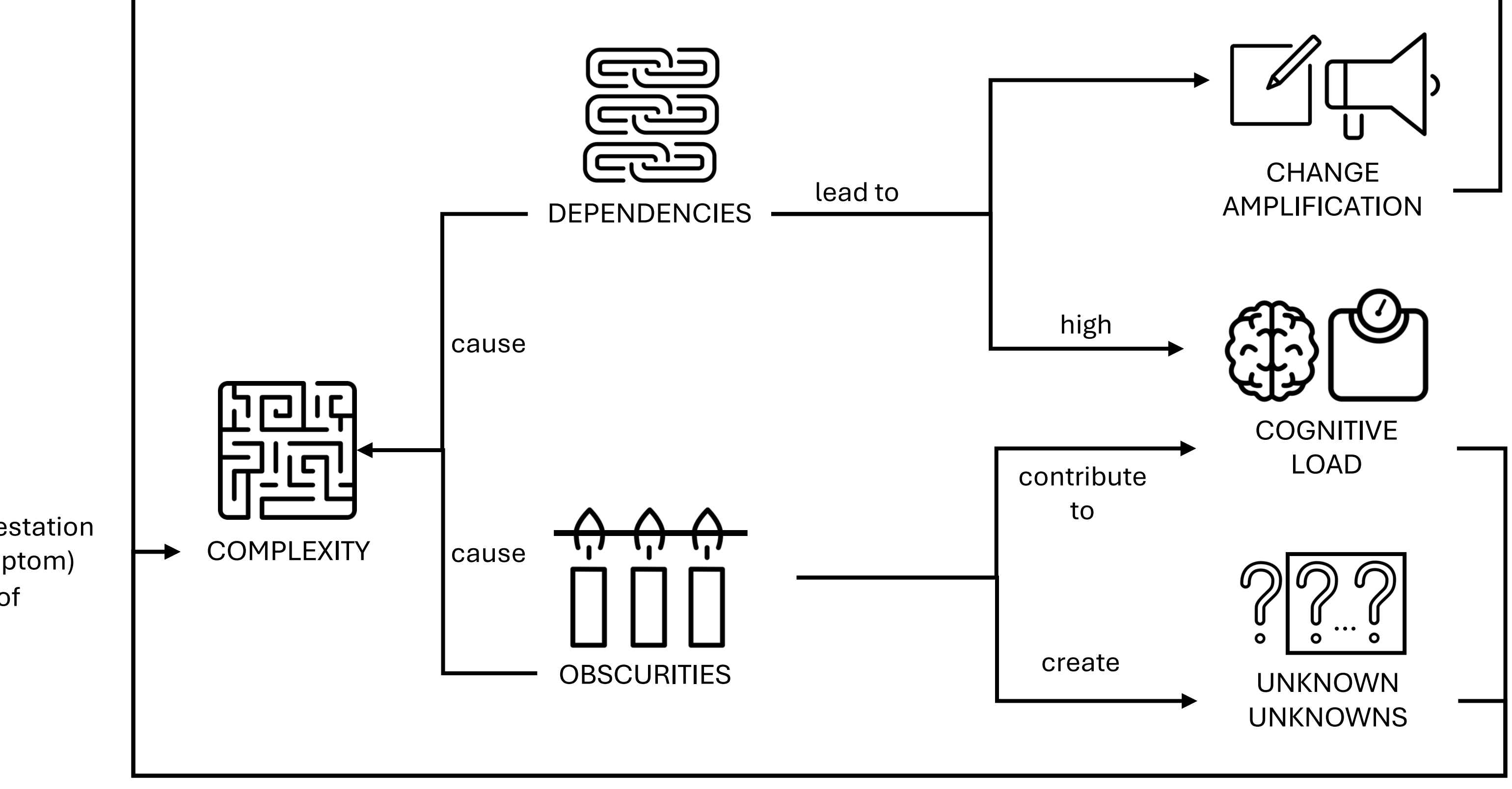


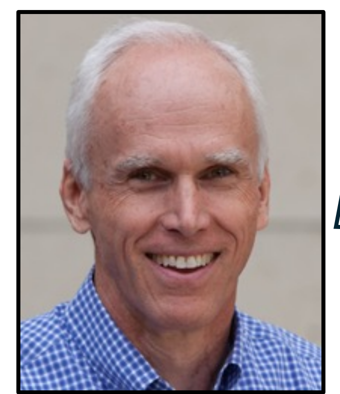
John Ousterhout

Dependencies lead to **change amplification** and a **high cognitive load**. **Obscurity** creates **unknown unknowns**, and also contributes to **cognitive load**.

If we can find **design techniques** that minimize **dependencies** and **obscurity**, then we can **reduce** the **complexity** of software.







John Ousterhout

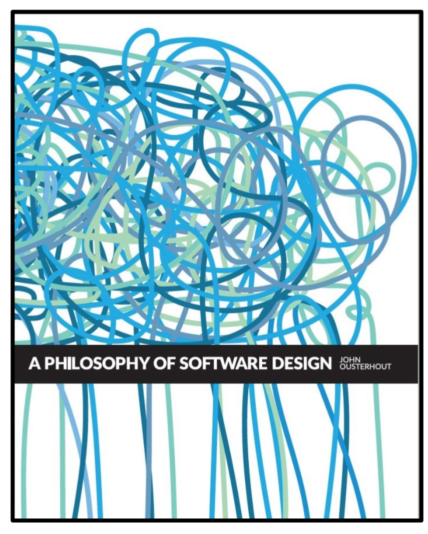
The **incremental nature** of **complexity** makes it **hard** to **control**.

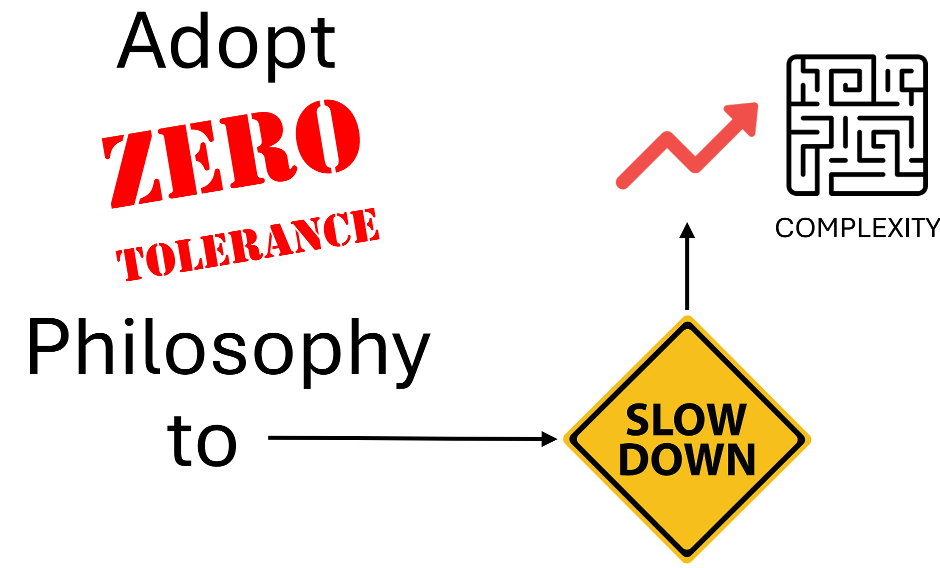
It's easy to convince yourself that a little bit of complexity introduced by your current change is not big deal.

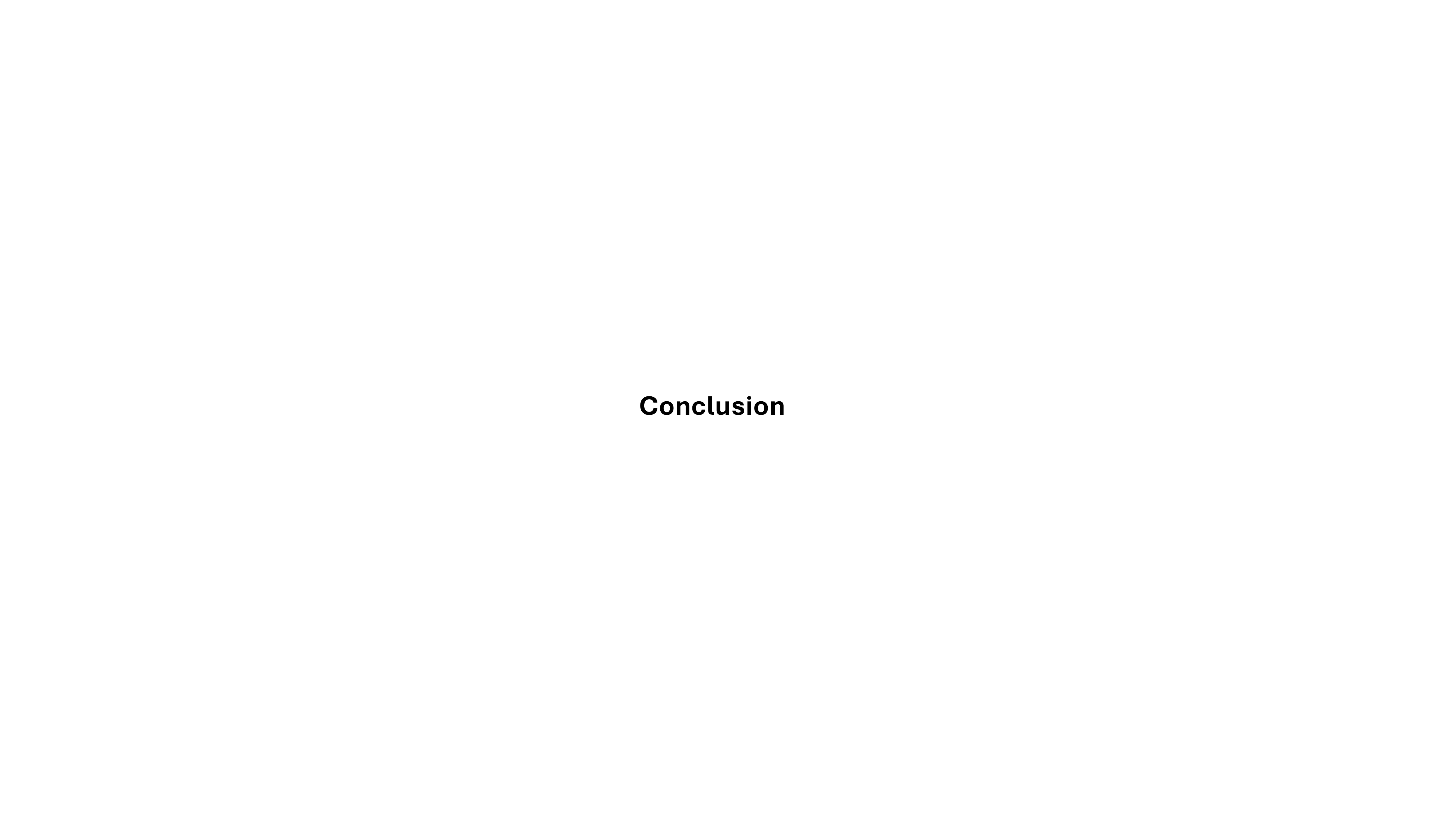
However, if every developer takes this approach for every change, complexity accumulates rapidly.

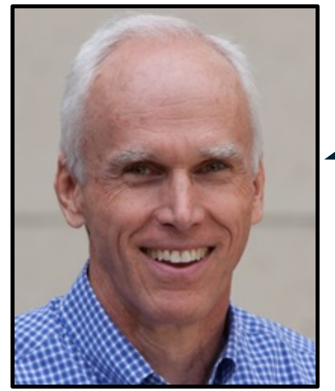
Once **complexity** has accumulated, it is **hard** to **eliminate**, since fixing a single **dependency** or **obscurity** will not, by itself, make a big difference.

In order to **slow** the **growth** of **complexity**, you must adopt a **"zero tolerance" philosophy**.



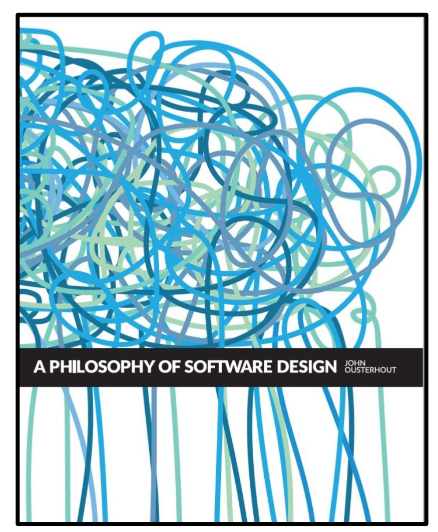


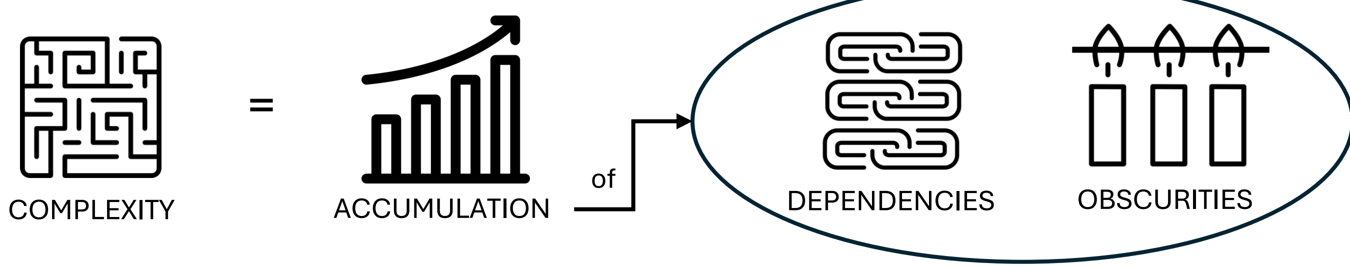

Conclusion

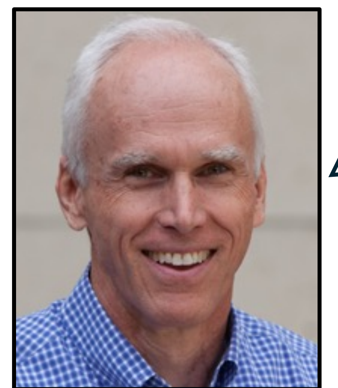


John Ousterhout

Complexity comes from an **accumulation** of **dependencies** and **obscurities**.

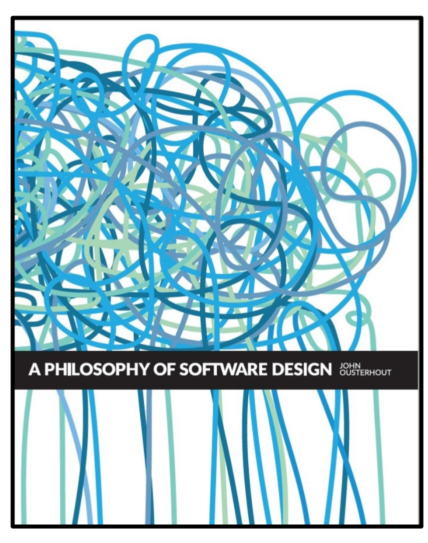


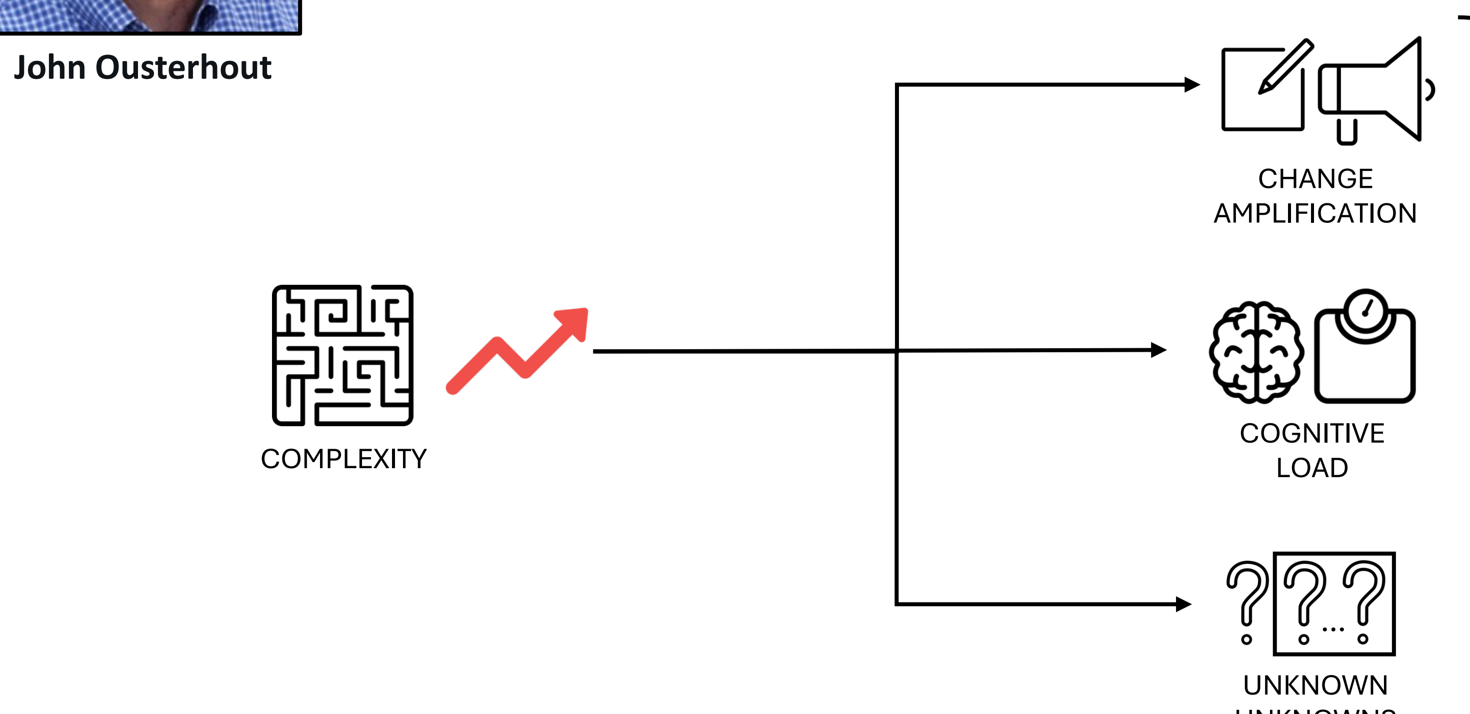




John Ousterhout

As **complexity** increases, it leads to **change amplification**, a **high cognitive load**, and **unknown unknowns**.





As a result, it takes more and more code modifications to implement each new feature. In addition, developers spend more time acquiring enough information to make the change safely, and in the worst case, they can't even find all the information they need.

The Bottom Line

Complexity makes it **difficult** and **risky** to **modify** an existing codebase

I hope you found that useful.