Computer "Science"

Is Computer "Science"?

What **kind** of science is "computer science"?

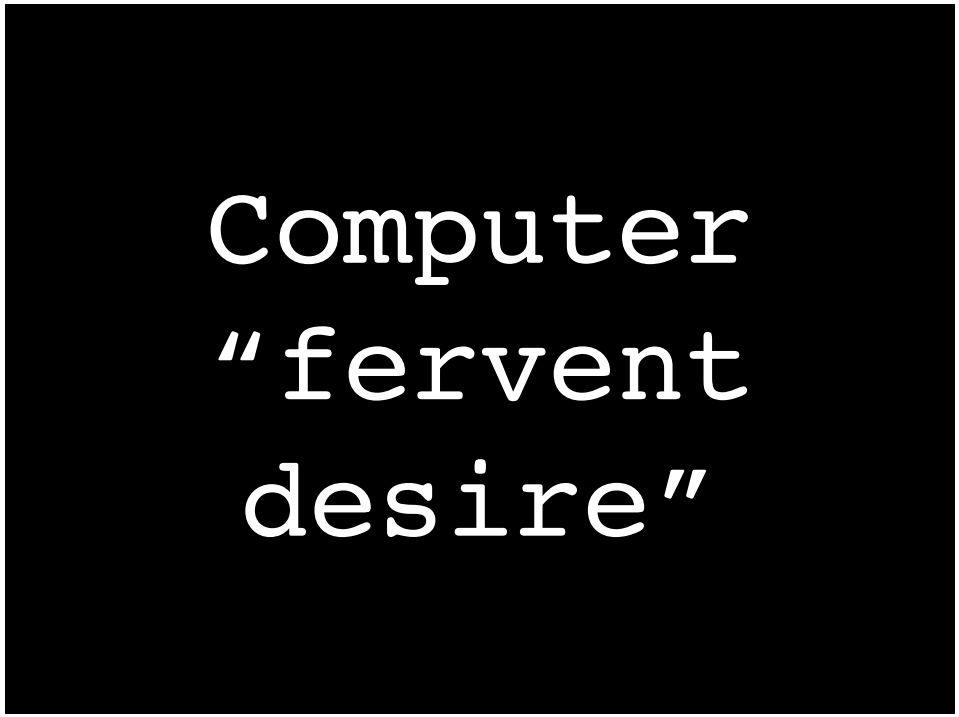
- algorithms -- mathematics
- operating systems?
- programming languages?
- software engineering?

Software engineering studys how we make software -- **software process** -- rather than the computational processes that define the field more generally.

We should make software engineering more than...

Computer
“strong
opinion”

or...



or...

Computer

"I heard about this guy once..."

Empirical Software Engineering

Since the early 1990s we have seen a growing emphasis (and more fervid desires!) to study the way people make software empirically:

- controlled laboratory experiments (for small things)
- case studies and data mining (for large ones)

What is true of small systems may not be true of large ones.
“There's a difference between molecules bouncing off each other and fluid turbulence.” — Greg Wilson

... or 5, or 100, or some other number. Every time we see this claim, it seems to use a different value for **n**.

The original study was done by Sackman et al. and published in 1968. The study followed only 12 subjects.

(1968! Most programmers were self-taught, which would make wide variation more likely.)
And And "up to" can hide a lot of sins

Boehm (1975) claims up to a factor of five. This is consistent with what we see in other creative disciplines.

Productivity depends
on the **length of a program's text,**
independent of
language level.

This reflects human (in)ability to manage detail.

It means more powerful languages make **us** more powerful.

Problem: programs written in such languages run slower.

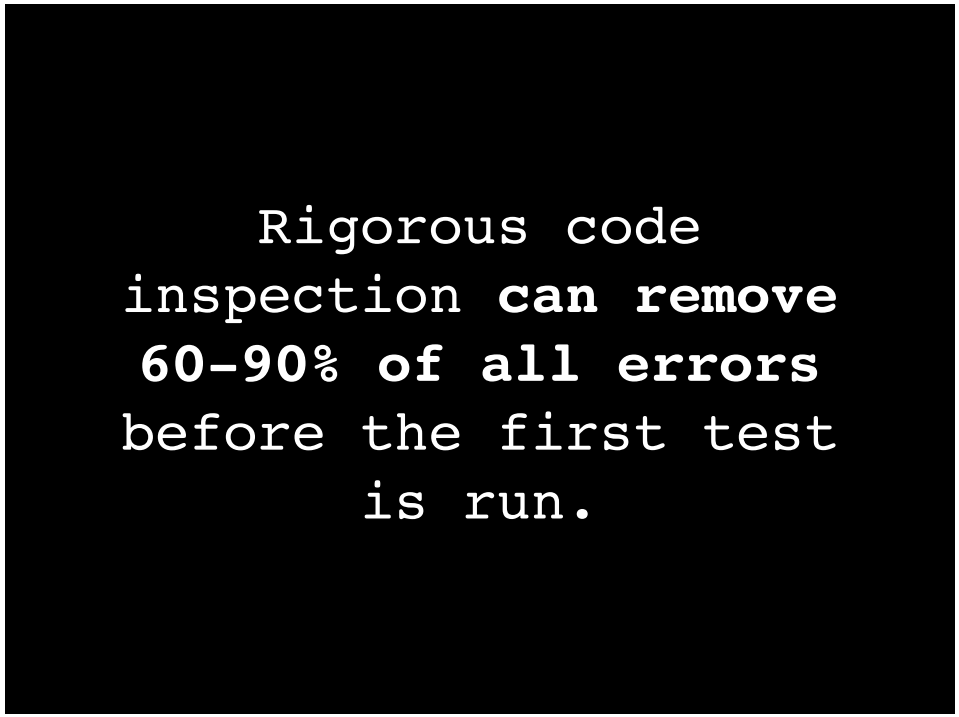
Practice: Build the first version using high-level tools. Profile. Rewrite code in the bottlenecks. Port to a lower-level language only if essential.

Recent examples: Ruby on Rails.

Typically 20% each on requirements, design, and coding, then 40% on fixing things -- either immediately or later.

All that traditional and agile processes do is change **the sizes of the chunks**, not their proportions.

Is refactoring fixing, or not? It depends.



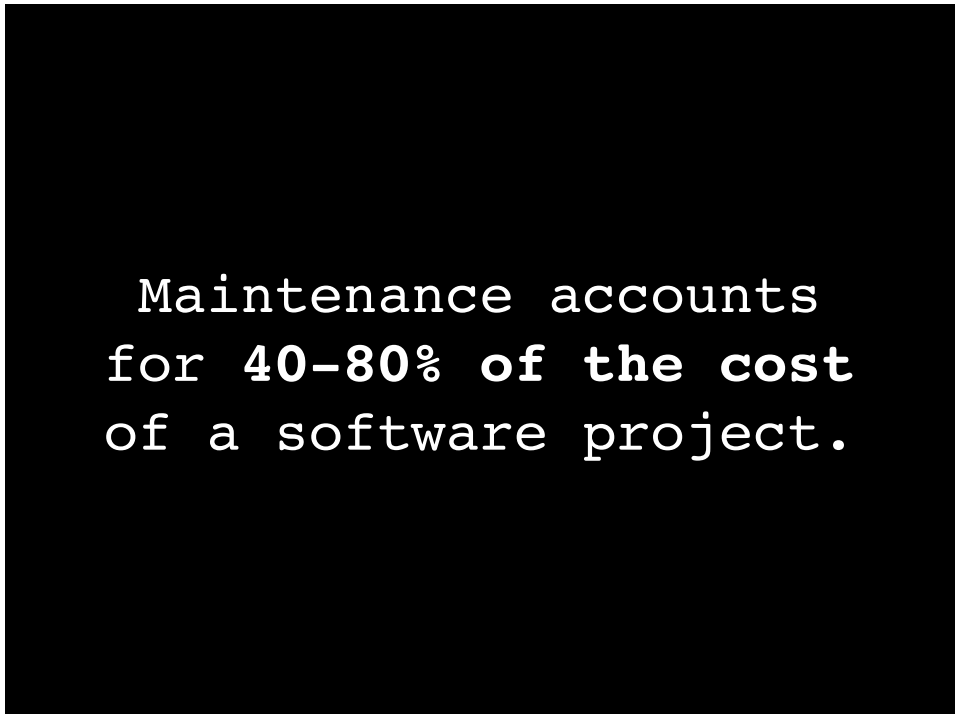
That is a wide range, but the number is big in any case.

But is this more economical than writing tests?
Especially because most code is modified several times before being shipped?

Yes! Several studies have shown that code inspections are the single most cost-effective error removal technique.

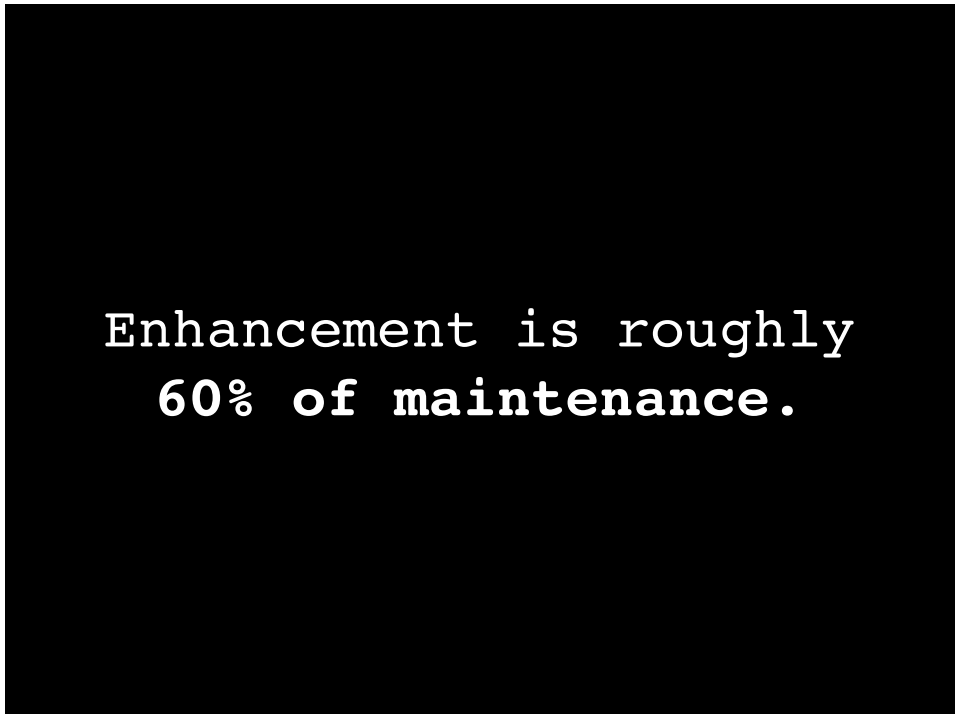
Pair programming is a form of continuous code review used in the agile world. Finding defects more than makes up for the cost of two programmers.

The "culture of review" in many open source projects is one of the reasons their code is so good. But there is no evidence to support the claim that "Given enough eyeballs, all bugs are shallow."



[BOE75] is an early reference, but the finding has been validated many times since.

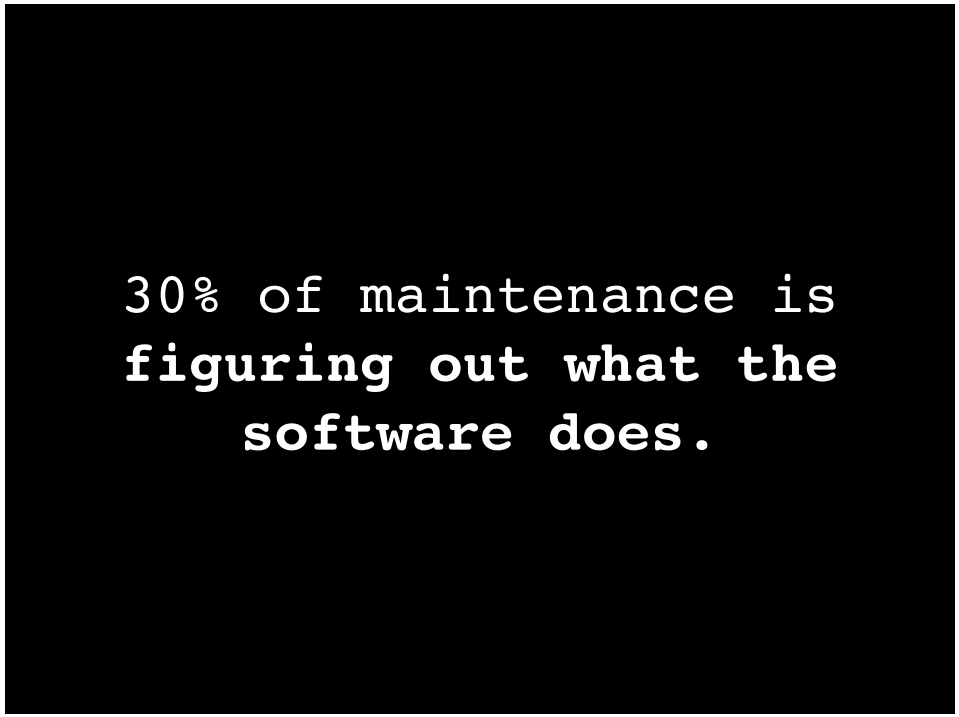
Its is the single largest cost in most projects, **but almost always underestimated**.



Enhancement is a result of **changing requirements**.

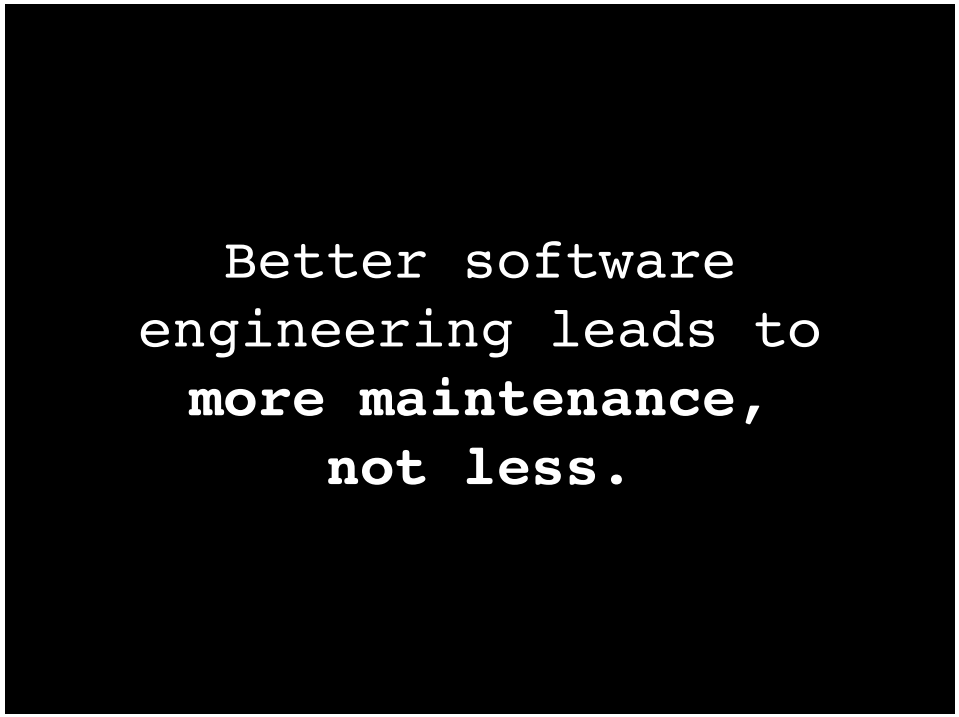
Yes, they keep changing after software is in production.

How much effort goes into other kinds of maintenance?
18%: adaptive maintenance (i.e., keeping up with a changing environment)
17%: error correction
5%: miscellaneous



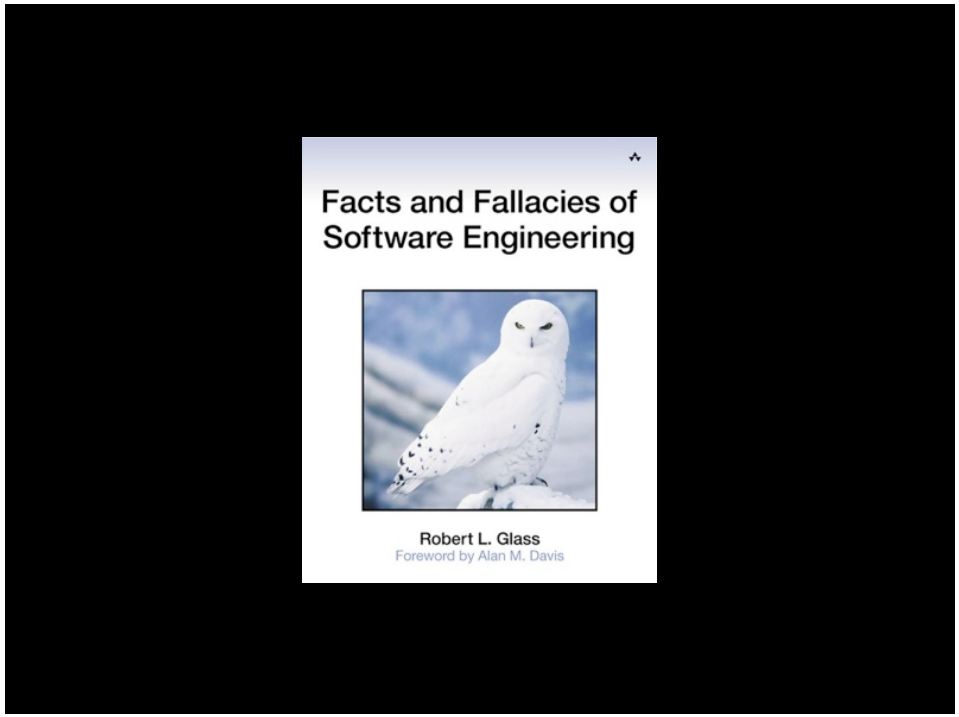
This figure rises as software ages.

Documentation.
Institutional memory.



Why?

The better the system, the longer it will live.
The longer a system lives, the more changes are possible!



If you want to be a software engineer, you should know what is true about practices in our discipline.

If you can only read one book, this is probably the single best summary of empirical SE results:

Robert L. Glass, **Facts and Fallacies of Software Engineering**, Addison-Wesley, 2002.

Here are some of the papers documenting the claims made in this session:

- Barry Boehm, "The High Cost of Software", **Practical Strategies for Developing Large Software Systems**, Ellis Horowitz, 1975.

- H. Sackman, W. I. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performances", **Communications of the ACM**, 11(1), 1968.