http://www.youtube.com/watch?v=HxTZ446tbzE
Tacoma Narrows Newsreel

software crisis

a term coined at the first NATO Software Engineering Conferences (1968)
http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

does not meet requirements

inefficient

of low quality

difficult to maintain

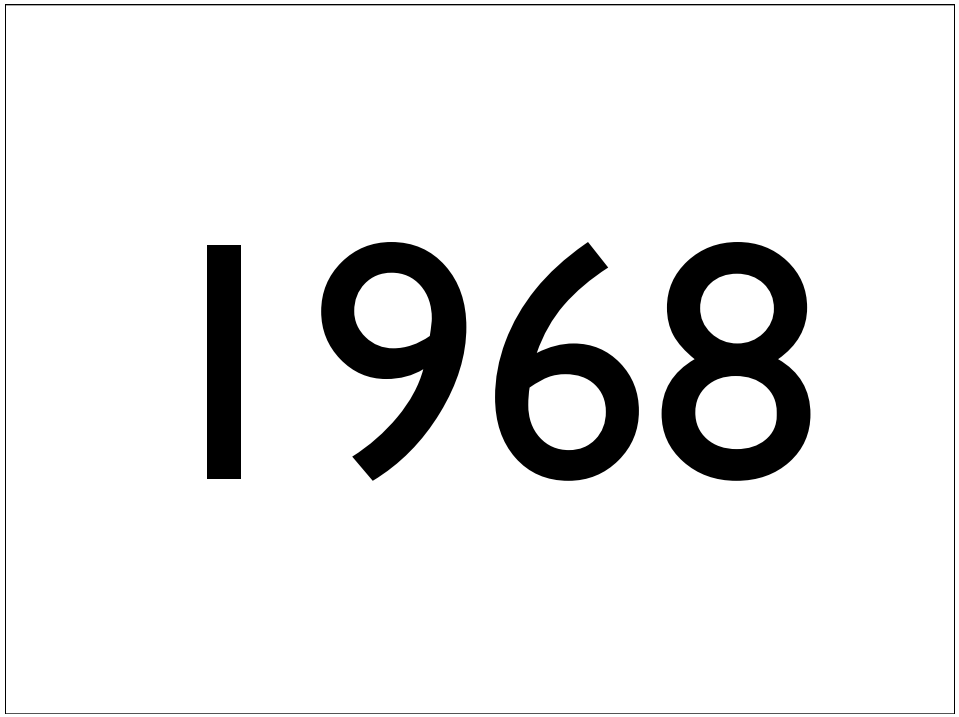
Software is ...

over-budget

over-time
(or never delivered)

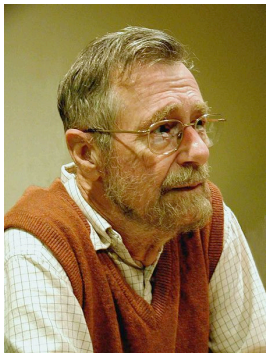
unmanageable

Projects are...


1968

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Edsger Dijkstra, in
The Humble Programmer



This is an issue of **size**. Small programs are relatively easy to write and maintain. As the tasks we want to solve grow, so do the programs we write. As a program grows, so does its complexity. We need to find ways to tame the **complexity**.

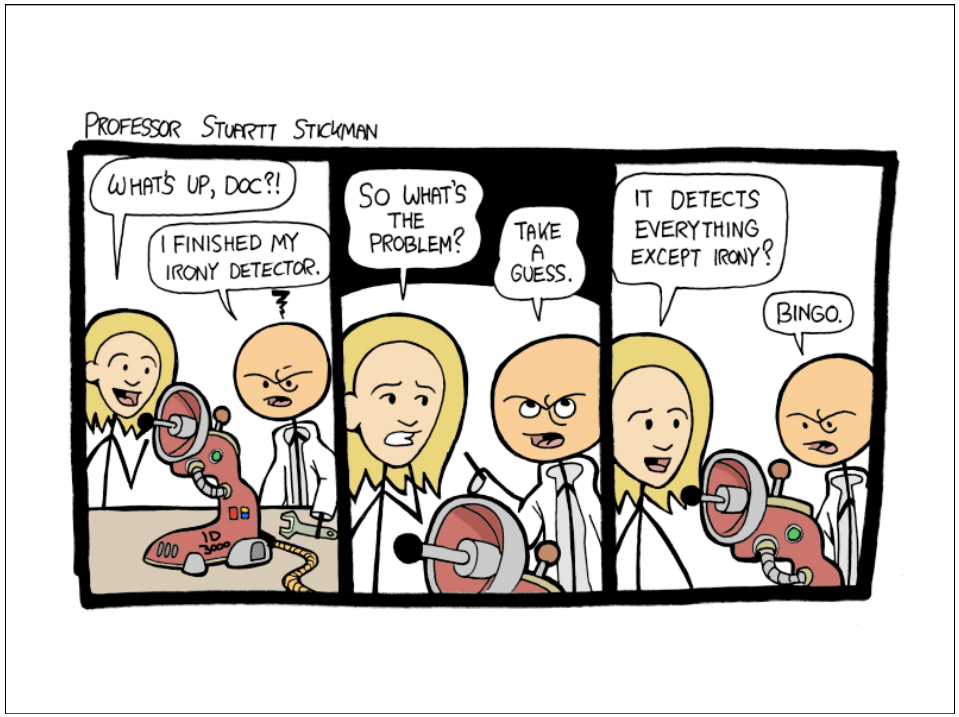
Complexity matters.

Size matters, but only because of complexity.
Size of what we **can** do, and size of what we **want** to do.

... the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes ... with full cognizance of their design ... as respects an intended function, economics of operation, and safety to life and property.

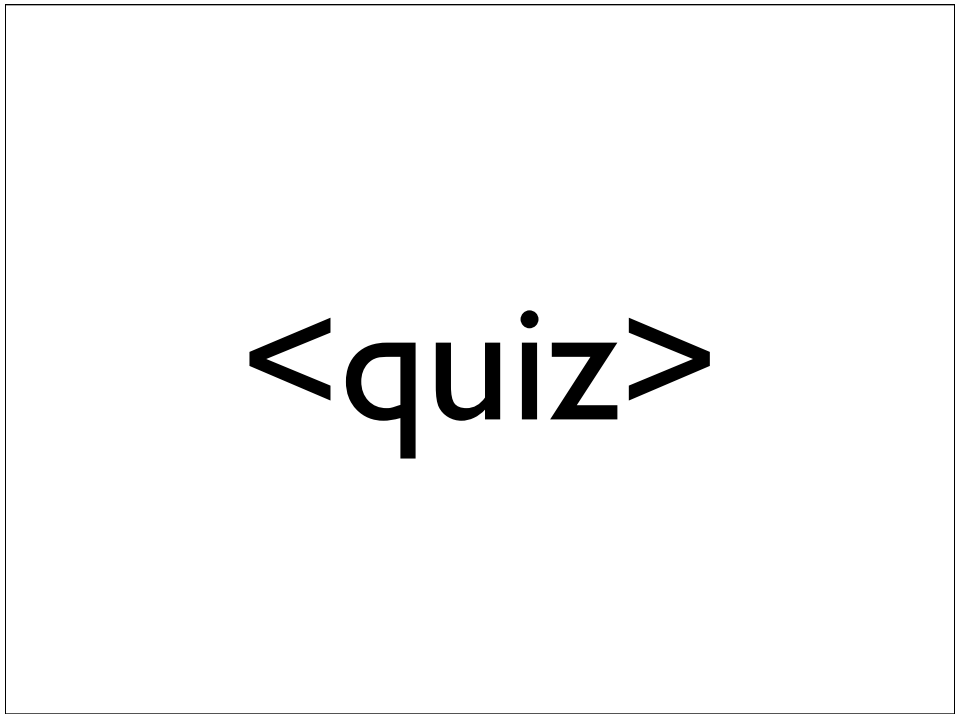
How to manage the complexity? Engineers practice disciplines and follow processes that enable the creation of artifacts repeatable, predictable, measurable, high quality.

Engineering as defined by ABET's predecessor.



After having watched the video at the top of class, I can't help but chuckle at the irony...

(courtesy of http://www.viruscomix.com/page329.html)



I am not grading this. It isn't a quiz. It is a **self-assessment**.
Honesty is the best policy.
Your answers will help me design and implement this course better.

Software Development Lifecycle

analysis
design
implementation
testing
deployment
maintenance

These are things we can or should do when we create a program.
They are often called **stages** because software roughly progresses through them in order.

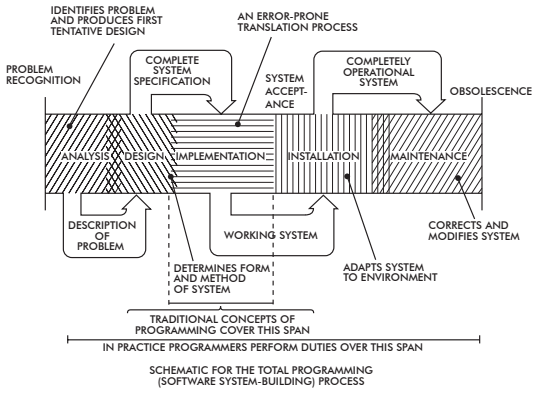


Figure 2. From Selig: Documentation for service and users. Originally due to Constantine.

Notice: Page 13 of the report of the 1968 NATO conference.

-- no testing

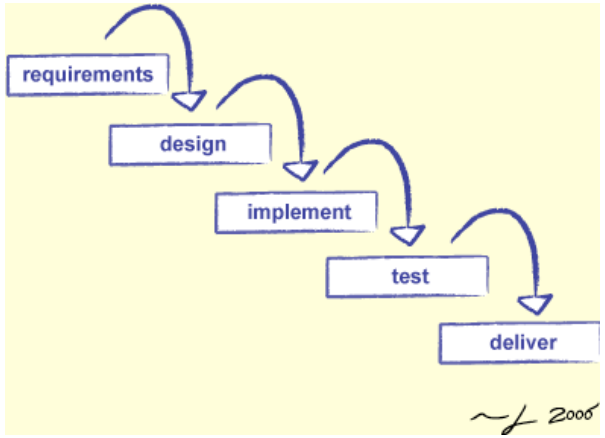
-- “obsolescence”

-- discussion of programming and what programmers do

That diagram had feedforward, but no feedback.
When we lock these **stages** in a time progression, they become a **process** for development.

Waterfall Model

This once was how people were taught to build software.



It is now frequently derided in the literature and in practice.
But...
It is also still a model you will seen used in practice by many, many companies.
(Some, even as they tell you they aren't doing it!)

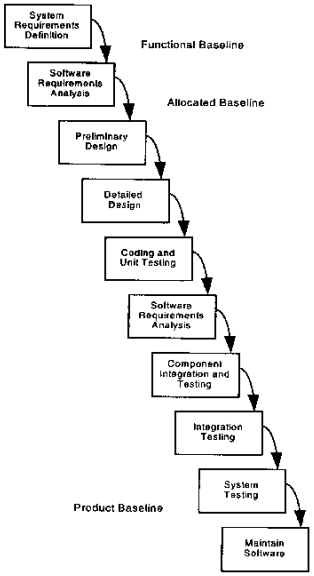


FIGURE 1: WATERFALL MODEL.

http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp
"A Comparison of Software Development Methodologies"
Reed Sorensen, Software Technology Support Center

metaphor

We liken our activity to another and learn from the connection.

Making software

is

engineering.

If so, then we should do what engineers do.

Making software

is like

engineering.

Many people take a broader view.
SE is like engineering, and we should learn as much as we can from engineers, who have been practicing much longer than we, and in harsher circumstances. (Bridges fall down!)

analysis
design
implementation
testing
deployment
maintenance

These things are definitely part of making software. Engineers can teach us a lot.

But making software is also not like engineering. How?

Programming is not like material craft in that undoing something you've done is easy far more often than not.

Courtesy of Brian Marick, a long-time software developer and consultant.

This is one example. There are others. (Can you think of some?)

We will consider similarity and dissimilarity, and learn what both can teach us.
The dominant metaphor of this courseis engineering.
But we will also look at responses to traditional engineering, at other approaches.
Learning anything involves learning it both inside and out. Skepticism.

http://www.cs.uni.edu/~wallingf/teaching/172/

The mechanics of course: readings, assignments, project, exams.
My course design is agile development:
-- identify essential requirements, prioritized, implement the top priorities first.
The questions you answered earlier are a part of my analysis.

what this course is and isn't

not
(only)
the software engineering
of large corporations

software at different scales
Principal and Microsoft Word, and also the small companies you will work for
Google and boutique web sites
large programs — and small programs

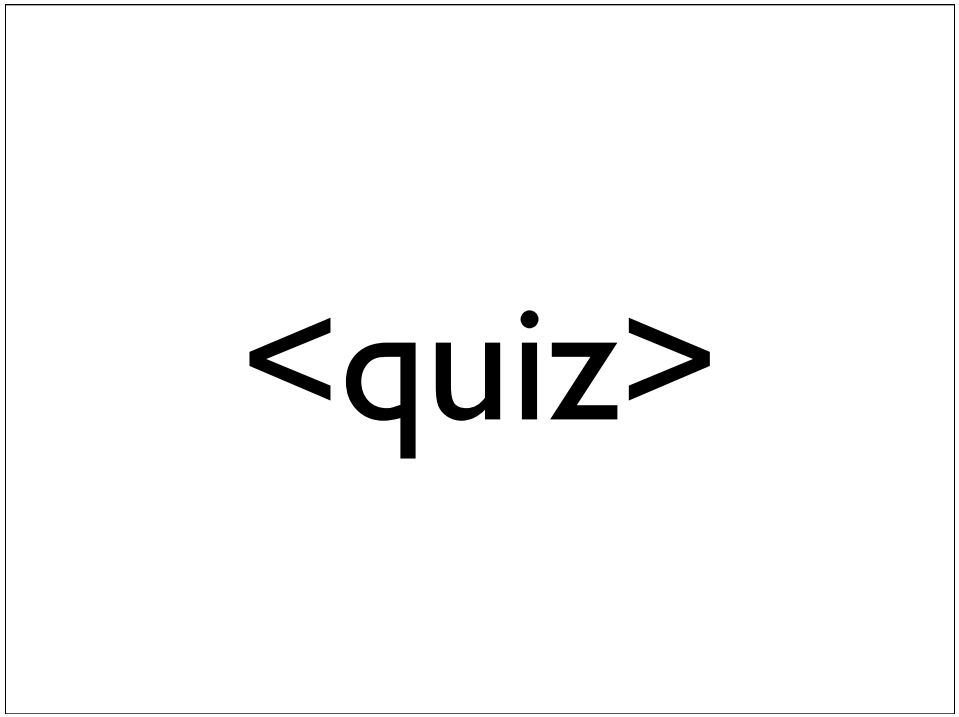
the skills and techniques
that make us
productive
and good

I would like for you to begin thinking about software development as something worth doing better, something worth thinking about so that you can do better.
Learn and develop practices that make you a star.

One can never trust an engineer
who does not have to wash his
hands before he eats dinner.

Kiichiro Toyoda,
the founder of Toyota

Software engineers make software.
We should learn **skills and tools** that will help us do that better.



problem ... weakness ... pain ... gap in your knowledge or skill set

Be specific.