Goals and Principles

We now have a definition for software engineering. We also know about the software crisis. It should be obvious that some of the main goals of software engineering would be to ameliorate the software crisis. There are some additional problems that software engineering can address. We will start by discussing these goals.

Much of the early work in software engineering was devoted to figuring out ways to achieve these goals. Attacking the complexity issue was first among them. As we have seen, any engineering discipline follows a set of well-defined and repeatable processes and procedures. We will introduce some of the principles that underlie these software engineering processes.

Goals of Software Engineering

- Modifiability
- Efficiency
- Reliability
- Usability

D. T. Ross, J. B. Goodenough and C. A. Irvine, "Software Engineering: Process, Principles, and Goals," in Computer, vol. 8, no. 5, pp. 17-27, May 1975, doi: 10.1109/C-M.1975.218952.

One of the main concerns at the 1968 NATO conference on software engineering was to begin the discussion of goals of software engineering. This prompted a host of papers in the literature. One of the most influential of these papers was by Ross, Goodenough and Irvine, in which they discussed four main goals. Of course, if you do a quick search, you will find that other authors identified many more goals. Let's think about these four goals as a starter.

The first is modifiability. As we discussed in an earlier video, one of the main problems we had with the production of software was that there was never enough time to get things right. We were lucky to get something out the door on time. As we noted, one aspect of the software crisis was that what was delivered was not what the customer wanted or needed. However, when we tried to make changes to the software, we frequently made things worse. The software wasn't designed to make it easy to change. A change to one part of the program would result in ripple effect problems in other parts. When we go to fix those defects, we would introduce more ripple effect defects. Things would just compound. Most of the time we would need to just rewrite the whole thing from scratch.

Efficiency has always been a problem with software and has thus been goal of software engineering. We always tend to want to push the envelope of the capabilities of the hardware in terms of both speed and memory usage. This has particularly been true in the early days of software engineering. It may be less of a

concern today because the cost of memory and processing power are not as much of a gating factor.

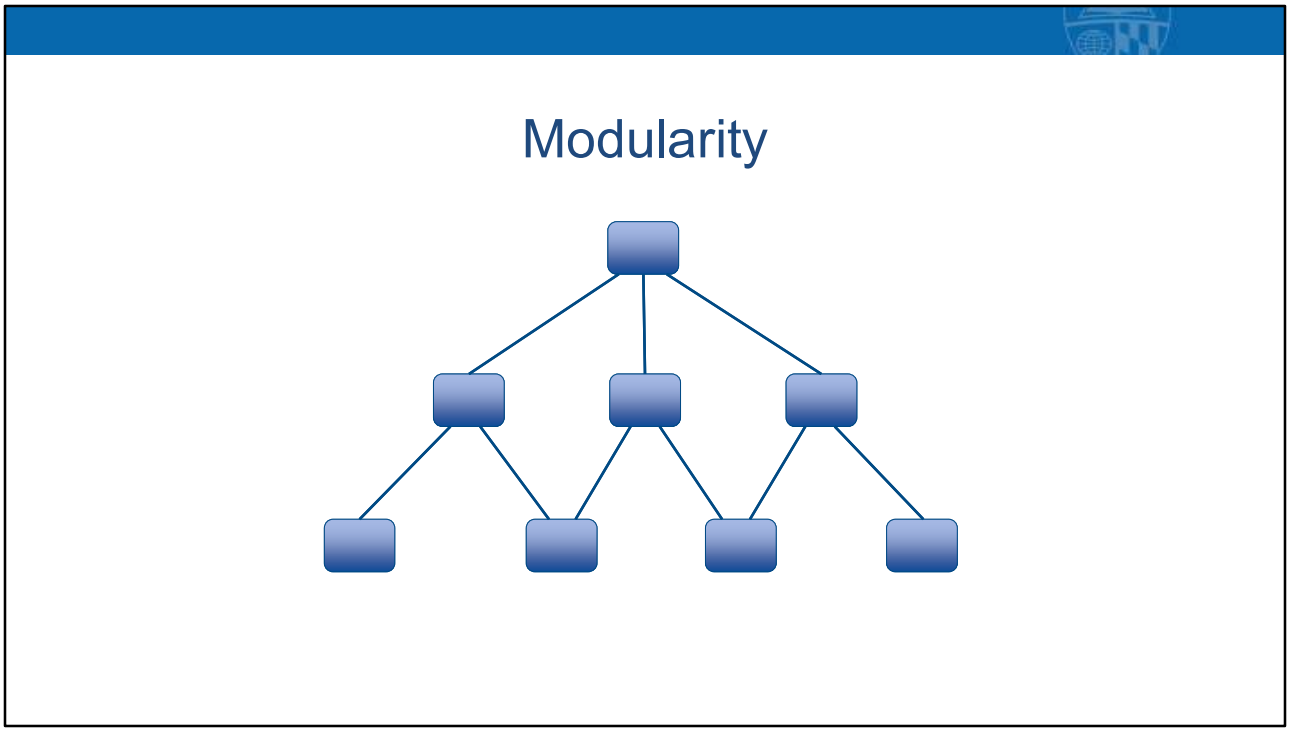
Reliability means can we rely on the software to do the right thing? How long will it run before it breaks? Reliability is measured in something called mean time to failure: how long will our software run properly and satisfy our needs until it breaks?

And usability: can we use the software? Is it useful for us? And as a measure of this: is the customer satisfied with what they're getting? Is it something they need? Is it something they can use? Usability is related to human factors, and this is one of the main goals that they'd identified early on. This fits in with part of our definition of software engineering: real products for real people.

Software Engineering Principles

- Modularity
- Abstraction
- Encapsulation
  - High cohesion
  - Low coupling
- 4 C's: Correct, Complete, Consistent, Confirmable

There are some principles that are embodied by software engineering, and we'll be talking about a lot of these principles throughout the course. First, we want to introduce them here. Modularity, abstraction, encapsulation, the 4 Cs.



The best way to control complexity is to reduce the number of things that need to be connected. So, we saw that the complexity of code was very high because of the interconnections between lines of code. Well, is there something we can do to reduce the number of things that need to be connected?

Well, instead of lines of code let's view the program as a collection of modules or components. There are well defined connections between the modules.

Modules could be procedures, or functions, or classes, or packages for example.

One way to identify the modules would be to use the familiar divide and conquer approach to problem solving. If a problem is too big to solve, break it up into several smaller problems. If you can solve those, then you can combine their solutions in order to solve the original problem. If you can't solve one of the smaller problems, break it up into even smaller problems. Repeat this process until all the problems are solved.

Some modularizations are better than others. A poor approach is random modularization, or sometimes called accidental modularization. As an analogy, consider printed circuit boards. Most electronic equipment today, such as computers or TVs, have a number of small modular circuit boards. Each one has its own purpose. In the old days, a piece of equipment, such as a TV, would typically have one large

circuit board. Now, I could modularize this board with my table saw and cut it into six or eight smaller boards, but this wouldn't be considered good modularization. I would be slicing through connections. The individual boards wouldn't have their own functions and purpose the way we see in the equipment of today.

Encapsulation

- High Cohesion
- Low Coupling

Good modularization involves building modules that have well defined purposes and fit well together, with well defined connections.

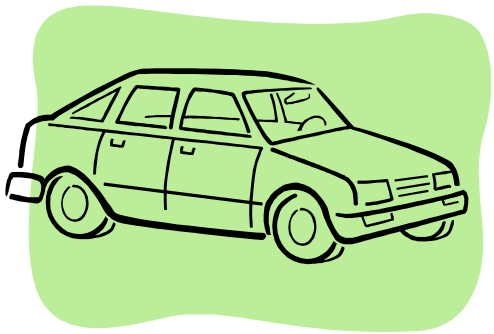
We use the term encapsulation to convey this idea.

A module should have a single purpose. We say it should be cohesive. What that means is that it "hangs together." There is a principle of maximum cohesiveness. If a module is at its maximum state of cohesion this cohesiveness will be reduced in one of two ways: either adding something (data or behavior) that is not related to the purpose of the module, or by taking something (data or behavior) out of the module and putting it somewhere else. In either case, it no longer hangs together.

Modules should have well defined connections to other modules. This is referred to as coupling. We prefer what we call lightweight coupling. The interfaces should be as simple as possible.

Things work best if the module hides its complexity and provides a simplified interface to the other modules that need to make use of its capabilities.

Abstraction



A simplified interface can be specified in terms of abstractions.

An abstraction is a view of something that highlights the essential properties while at the same time suppresses unneeded details.

For example, we are familiar with the abstract interface for driving a car. We must know how to start it, accelerate and decelerate, turn, stop, operate the wipers and turn signals. Not much more than that. To the driver, that's a car.

To a mechanic who must fix the car, there's a whole lot more complexity that they need to know about. Fortunately for most of us drivers, the abstract interface provided by the car hides all that complexity.

The 4 C's

- Complete
- Correct
- Consistent
- Confirmable

There are four other qualities of software that we should strive for. We call them the four c's.

Our software should completely satisfy the requirements or needs of the user. No missing capabilities.

Our software should correctly implement the required behavior and information. No bugs.

All parts of our solution should be consistent. They should have the appearance of having been created by the same person. This means that there should be some sort of standards and conventions.

Software should be confirmable. It should be possible to demonstrate that our software is correct through some sort of testing or proof of correctness.

Next

- Is Software Engineering a Profession?

One hot button topic that is currently being debated in our industry is whether software engineering is a profession. The answer is not clear. We will consider this question in the next video.