Editor in Chief: Steve McConnell ■ Construx Software ■ stevemcc@construx.com

Who Needs Software Engineering?

Steve McConnell

The traditional distinction between software and hardware was that software was easily changeable and therefore "soft," whereas hardware was captured on a physical medium like a chip, was hard to change, and was therefore "hard." This traditional distinction is breaking down today. Software delivered via the Internet is clearly "soft" in the traditional sense, but software delivered via CD or DVD is hardly "soft" in the sense of being "easy to change."

We now commonly see software being delivered on EPROMs; the electronic control module that controls my car's fuel injection is an example. I can take my car to my dealer to have the chip reprogrammed, so in some sense the program on the chip is soft, but is it software? Should the chip developers be using software engineering?

Computer chip designers are now doing much of their chip development using software-engineering-like tools. Only at the last minute is the code committed to silicon. Do we really think that committing code to a CD-ROM makes it software but committing it to a silicon wafer makes it hardware? Have we arrived at a point where even computer hardware is really software?

If software and hardware are totally different, then electrical engineers designing computer chips don't need to know about software engineering. But if modern chip design involves a significant amount of programming, then perhaps electrical engineers should know something about software engineering. Should computer hardware be designed using software engineering?

Throw a few other disciplines into the mix such as Web programming and games development, and I think a fundamental question lurks here: What is software? This question is important because it leads to a second question: What is software engineering today, and who needs it? I recently posed these questions to several *IEEE Software* board members.

Blurred distinctions

Wolfgang Strigel: No doubt, the distinction between software and hardware has blurred. When the term *software* was coined, there was a clearer distinction, or nobody cared because it sounded good and made intuitive sense. Moreover, it is not important that software is modifiable (or "soft" once it is completed). Software does not change its nature by being turned into something "hard" or unmodifiable. After all, we have accepted the concept of selling software on CDs. And RAM can also be write-protected. What matters is whether there is a "program" that can be executed by a device.

The project plan for building a high-rise is a set of complex instructions, decision points, and so on that could be interpreted as a software program. But it is not executed by a device. But how about multimedia, say an animated movie? It has instructions for movement, rendering, and so on, and is executed by a device. It can also be modified. How about a set of MIDI instructions that produce music if delivered to an electronic

IEEE Software

instrument? This does not fundamentally differ from a ladder diagram that controls the execution of a programmable logic controller.

Larry Graham: I agree that the line between software and hardware is blurry. Patent law has a fairly rich tradition that equates the two—virtually every hardware device can be described in terms of a function it performs and vice versa.

Is software engineering invariant?

Annie Kuntzmann-Combelles: I think the basic practices needed to develop software properly are always the same: get clear and complete requirements from customers; manage changes to requirements; estimate, plan, and track the work to be done; select an adequate life cycle; define and perform QA activities; and maintain product integrity. The software architecture and coding might differ from one application to the other, but the process aspects are invariant.

Tomoo Matsubara: I don't think software development is always the same. My recommendation for improving software process is to apply domain-specific methodologies and tools and conduct problem-focused process improvement. The levels of importance and priorities are different between domains.

For example, one of the most critical processes for commercial software is fixing data flow. For scientific software, a key to success is choosing the right algorithm. For COTS, it's designing a good human–machine interface. For embedded systems, it's pushing instructions into the fewest memory chips. For maintenance, it's rigorous testing with regression tests. Software development practices should vary accordingly.

Grant Rule: Think about the games industry, where the "software" is delivered on CD-ROM or game cartridges. Game development can take vast amounts of schedule. Teams can be quite large—25 to 30 people from a variety of disciplines including analysts, designers, coders, testers, QA staff, and project management—and lots of nontraditional software personnel such as writers, artists, and so on. Schedules must be managed carefully to meet holiday sales seasons. If a game misses its marketing window, it might be a commercial failure; there's no second chance. Reliability is important: the game is mass-produced on CD-ROM, and from that point forward there is no real chance to correct it—it is infeasible to recall hundreds of thousands of copies to fix a defect.

The result seems to be that, in some cases at least, game developers take a more rigorous approach to "engineering" their software than do some developers of commercial data-processing applications. All this seems to be "software engineering" to me.

What's unique about software?

Robert Cochran: I use the following definition to describe what is unique or special about software:

1. Software is intangible (which I think is true even if it gets embedded).
2. It has high intellectual content (many other intangibles have low intellectual content).
3. It is generally not recognized as an asset by accountants and so is off the balance sheet.
4. Its development process is labor intensive, team based, and project based. We forget sometimes how little of the rest of the world regards projects as the normal way to work.
5. Software doesn't exhibit any real separation between R&D and production.
6. Software is potentially infinitely changeable. Once you make a physical widget, there are severe limits on how you can change it. In principle, we can keep changing software forever.

Is there any real distinction between printing out a source code listing, creating a binary, or burning software onto a chip or CD-ROM? In all these cases, we are just "fixing" a copy of the software in some form that cannot

directly or easily be modified. That does not have any relevance to the nature of the software as such.

Grant: That makes software just like any written work, art book, or design drawing. The medium (technology) might differ—wax tablets, canvas, paper—but anything that can have multiple copies made in a mutable medium sounds like it could be this thing called "software."

Martin Fowler: Robert's Number 5 seems to be the key point. Until you deploy you can build, modify, and evolve the software, regardless of whether you eventually deploy to PROMs or CD. That ability to evolve while building is a key aspect of software that has no equivalent in disciplines where you must separate design from construction.

This question of how "soft" is software is quite an important point and one that gels particularly with me. One of the reasons that I'm so much in favor of light methods is that they try to make software softer while many methodologies try to make it harder. In the information systems world, softness is an important and valuable asset.

Making software softer

Steve McConnell: That does seem to be the age-old challenge: How do you keep software from becoming brittle? How do you keep it soft? Whether you're creating software, or a computer chip, or even a building, it seems as though it would be advantageous to keep the thing you're building "soft" as far as possible into the project.

Terry Bollinger: This overall question helps to deal with trying to understand the baffling diversity of production styles in the software marketplace. "Hard" software such as that found in processor chips is catastrophically expensive to fix after fielding, and so drives the entire software design process to be very conservative and validation intensive. "Fluid" software that can be changed automatically over the Internet drives the opposite behavior. I think that understanding these kinds of issues is quite fundamental to software management.

I do think we need to have a better overall definition of software. The very fact that I had to mangle together a phrase as awful as "hard" software to describe algorithms encoded into silicon shows that the industry has become more complex than it was in the days when you had vacuum tubes and bits on punched cards, and not much in between.

I think the distinction needs to focus on the idea of "information machines"—what we have traditionally called software—versus the particular method of distribution and update of such machines. Those are two separate dimensions, not one. A bunch of gears is not an information machine, because it relies primarily on physical materials for its properties, even if it happens to do a little information processing at the same time (for example, odometers in cars). A structured algorithm masked into an addressable array that is an integral part of a processor chip most emphatically is an information machine, because its complete set of properties can be represented as structured binary information only, without any reference to the chip's physical properties.

Don Bagert: In defining what's really software, my thought is to look at what the object code does (rather than where it resides), as well as the source code. I would define it as follows: "Software is a set of instructions that are interpreted or executed by a computer." The source code is definitely "soft" not in the sense of the deployment medium but in that it is a nonphysical entity. It consists of a series of instructions, by definition nonphysical, that can be translated into object code or interpreted by a computer processor. My colleague Dan Cooke has an interesting view: a Universal Turing machine corresponds to computer hardware, while an individual Turing machine defined to solve a particular problem corresponds to software.

What is software without programming?

Steve: My original focus on the medium on which the software hap-

pens to be deployed seems to have been a red herring. I think software engineering applies broadly to creating complex "instructions," regardless of the target media. Years ago, people argued that the need for software engineering had passed because Fortran had been invented. People didn't have to write programs anymore; they could just write down formulas! Thirty years later, we now think of Fortran programming as comparatively low-level software engineering—"writing down formulas" is harder than it looks.

As the years go by, we see the same argument repeated time and time again. The early claims about creating programs using Visual Basic were reminiscent of the early press on Fortran—"No more writing code! Just drag-and-drop buttons and dialogs!" And today more new programs are being written in Visual Basic than any other language. Ten years from now, we'll probably see programming environments that are much higher-level than Visual Basic; people working in those environments will benefit from software engineering.

Martin: This is an important point. Often people talk about things "without programming" (one of my alarm-bell phrases). You find this in phrases like, "Buy our ERP system and you can customize it for your business without programming." So you end up with people who are experts in customizing XYZ's ERP system. Are they doing software engineering? I would say yes, even though their language is putting configuration information into tables. They still need to know about debugging and testing.

Indeed, whenever we talk about writing tools so that "users can configure the software without programming," we run into the same problems. If you can enter programs through wizards, you still have to be able to debug and test the results.

Wolfgang: At the risk of being simplistic, I would define software as follows: "Software is a set of instructions that are interpreted (or executed) by a computer." I did not say "electronic computer" because it could well be a chemical computer or one that operates with light. That delegates the problem to defining the term "computer"—and that's a hardware problem!