How Hardware and Software Engineers Differ

Kevin Thompson, Ph.D. www.kevinthompsonphd.com

In my various careers in physics, software engineering, and Agile consulting, I have had many opportunities to observe hardware and software engineers at work. I find the similarities and differences between these populations to be fascinating.

Let's start with the similarities. What all engineers have in common is the core reality of engineering: They make things, mostly for other people to use. In particular, they focus on making things that are useful in some fashion, as opposed to things intended primarily aesthetic purposes. Sometimes these things are sold to customers, and sometimes they are used internally by an organization, but they all have useful functionality. (Yes, I know the boundary between engineering and art can blur, and some art requires engineering knowledge. I still consider the distinction significant.)

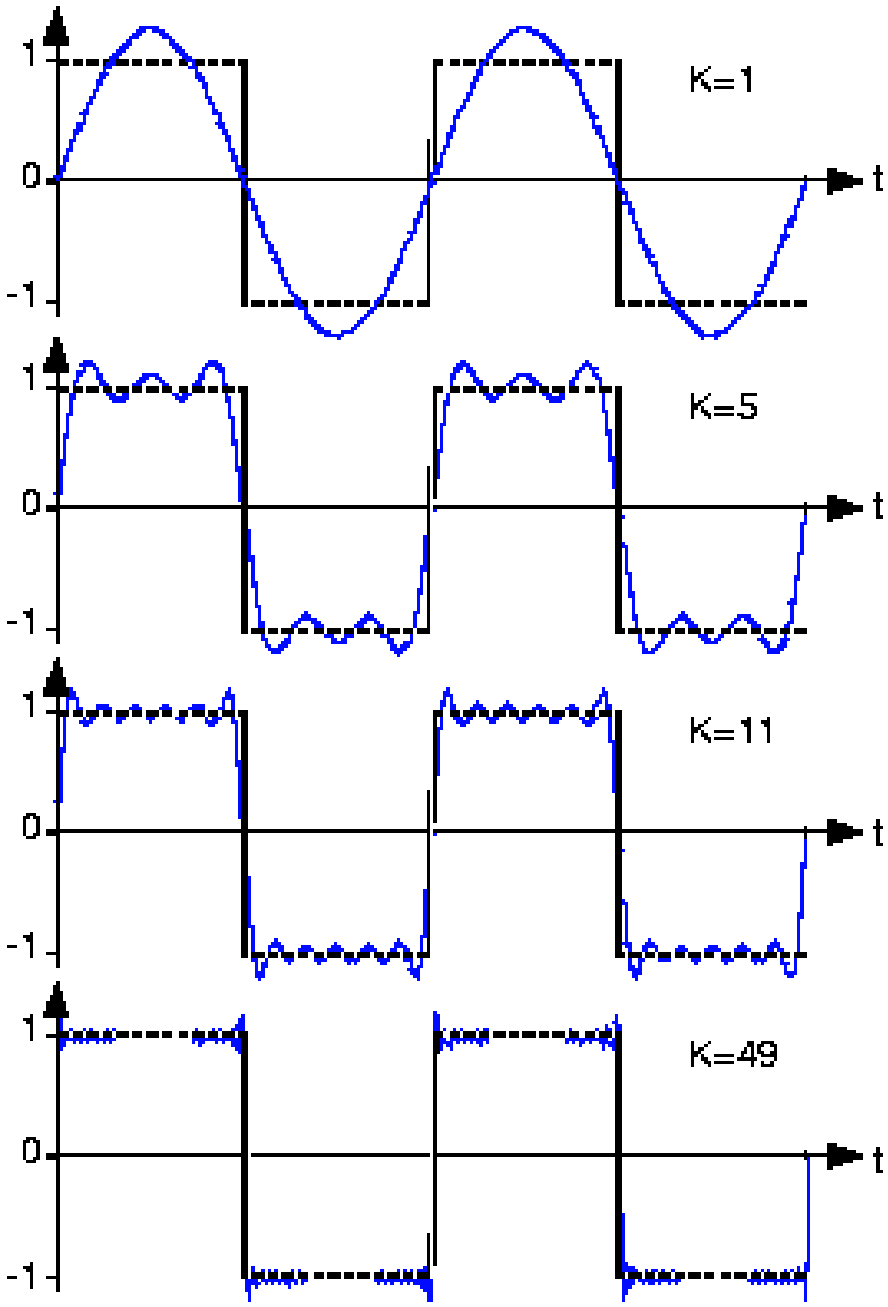
Differences between software and hardware engineering are driven by the differences between software and hardware products. Software products are built in a man-made world governed by binary logic (or Boolean Algebra, for the cognoscenti). The connection to physical reality exists, in the form of performance constraints, but is otherwise distant. (Note that I am talking about the software itself, not the nature of its application. Software that simulates or interfaces to the physical world does have to address physical characteristics but always does so in a form that reduces to binary logic at the most basic level.)

Hardware products are built for the physical world. All hardware products are constrained by physics, in very direct ways. Various portions of a physical product occupy space, have chemical and mechanical characteristics, consume power, require cooling, and so forth. Even digital signals that are intended to represent binary values must be mapped to a finite voltage range in the underlying analog world. For example, we might intend to represent the binary values of 0 and 1 by voltages of -1 and 1 volt. The actual signal for a square wave of frequency *f* is representable as a Fourier series, which can be written

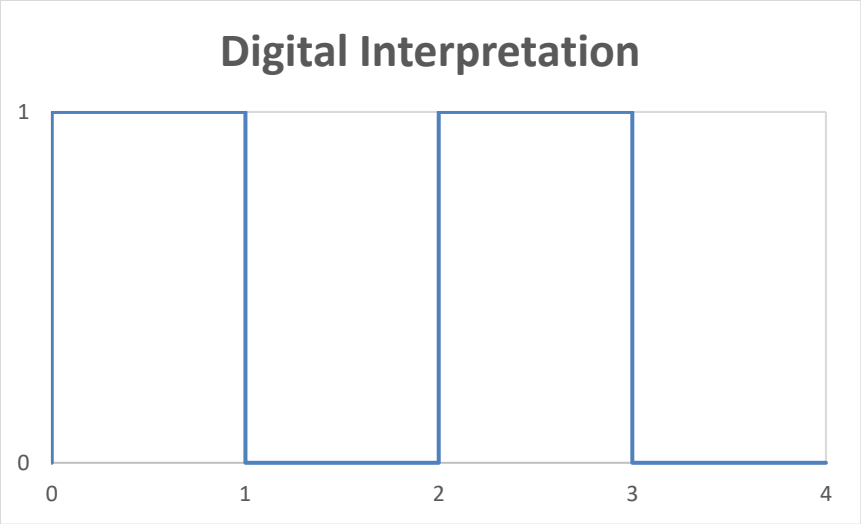
$$F(t) = \frac{4}{\pi} \sum_{n=1}^{K} \frac{\sin\left(2\pi(2n-1)ft\right)}{2n-1}$$

where K should be infinity. What we actually observe, as K gets bigger, is a series of approximate square waves that look like these:

Analog Representations of Step Functions



The overshoots at the corners never go away, no matter how large K becomes, meaning they are present even when K is set to infinity. Thus, real signals in real devices never exhibit perfect step functions. Some noise is always present, and the conversion of analog signals to appropriate digital signals can be difficult and subject to error. Extraction of the intended binary values requires making assumptions about voltage ranges and thresholds that must be managed in order to yield binary values.



To put it another way, the nature of physical reality is not digital. It obeys the laws of physics, not the mathematics of binary logic.

The differences between hardware and software engineers do not end with the differences between binary logic and physics but certainly reflect those differences. Both types of engineering require a mix of skills, but the mixes have different characteristics.

A team that develops a Web application might require knowledge of Java, Quality Assurance, test automation, and database development. A team that develops a camera might require knowledge of optics, mechanical engineering, electrical engineering, and firmware development.

At first look, the teams have a similar need for multiple skills and the ability to collaborate to develop their products. It is also usually the case that multiple teams of modest size (say, 3—9 people) must likewise collaborate to develop various portions of the product.

These similarities are real, but there are important differences as well. A key difference has to do with the concept of *generalizing specialist*.

A generalizing specialist is someone who has deep technical skills in one area, but can do work of mild difficulty in other areas relevant to his or her team. For example, the Java developer working on the Web application might be able to do development work on databases, as long as the complexity of the work is not high.

It is common in software development to have a team of generalizing specialists. This is a good thing, as it provides a degree of robustness in the face of resource loss (such as avoiding skill gaps when one team member goes on vacation, or moves to another job). The mechanics of planning work and allocating work to people is also simplified when the people are generalizing specialists.

It is not so common to have generalizing specialists in teams that focus on hardware development. The reason is that the individual disciplines (mechanical engineering, electrical engineering, firmware development) are very different, and each takes a great deal of time to learn. A software developer is

likely to learn enough about databases to be able to do simple work in that area. In contrast, it would be difficult for most electrical engineers to learn enough about mechanical engineering to do work in that area.

The relative lack of generalizing specialists in hardware development does not mean that Agile techniques cannot be used, but it does affect the details of how work is planned and carried out.

A Scrum team that develops software may have enough generalizing specialists that the team members can plan the next Sprint without spending much time thinking about which person is needed for each kind of work. A Scrum team that develops hardware probably does have to think about the practicality of partitioning work across people with different skills. This shift tends to make Sprint Planning meetings somewhat longer for hardware teams than for software teams.

To summarize, hardware and software development teams differ not just in their skills, but in their degree of specialization. This does not mean that Scrum, for example, cannot be used. (In fact, Scrum works well for both.) It does mean that it is important to address the different constraints that apply to these different worlds.