Introduction to Abstraction

By Brian Harvey

In this course, there's both a practical component, using Snap! in the lab, and a "big ideas" component, in lecture and discussion. One of the things we want you to take away from this course is to know that there's more to computer science than just writing computer programs, although of course we use those other things to help in writing programs. So we'll talk about some aspects of the social context of computing, some of the theoretical explorations of the limits of computation, and some important moments in the history of computer science -- for example, you'll learn in a few weeks how one of the first computer scientists more or less singlehandedly won World War II for the good guys.

This week's big idea: Abstraction. Arguably the central idea of all of computer science. Computer programming is easy, as long as the programs are small. What's hard isn't the programming, but the keeping track of details in a huge program. The solution is chunking, or layering -- two metaphors for abstraction.

The classic example is thinking about a car. Cars are made of nuts, bolts, metal rods, big metal blocks, rubber or paper gaskets, plastic containers for fluids, rivets, wires, and so on. (Each piece of metal is further made of atoms, which are made of electrons, protons, and neutrons, which are made of quarks, and so on down.) But if you're trying to repair a car, you don't think in those terms; if you did, you'd never find where the problem is. Instead you think about the engine, the alternator, the fuel injectors, the brakes, the transmission, and so on. That's abstraction.

The march of technological progress is, at least in part, a march toward greater and greater abstraction. Each step reduces the extent to which people have to think about details. Sticking with cars as the example, in the early days, every driver had to be at least something of a mechanic, knowing how to deal with the rather frequent failures of the machinery. Before automatic transmissions. Only people with some understanding of gear ratios could drive. (In the really early days, they couldn't downshift without mastering the skill of double-clutching.)

The automatic transmission made possible an enormous abstraction. All the complexity of the machinery that makes a car work was hidden under the surface of a very simple model: You push the pedal on the right and the car speeds up; you push the pedal on the left and it slows down. Suddenly just about anyone could drive a car.

Of course, the widespread use of cars has turned out to be a mixed blessing. Cars are one of the main causes of pollution and global warming. Computers, too, have their downsides, which we'll be discussing later. Many historians of science stay away from the word "progress," which I used two paragraphs back, because of its implicit suggestion that the development of new technology is always good. But before we can criticize technology we should understand something about how it works, and abstraction is a very powerful organizing idea to describe the mechanism.

Those two pedals, the gas pedal and the brake, are an /interface/, also known as an abstraction barrier. On the driver's side of the abstraction, what matters is the /behavior/ provided by this interface. Push this one to speed up, that one to slow down. Once that interface became standardized, further technical development has dramatically changed what happens up in the engine compartment. Originally, the gas pedal mechanically pushed a lever controlling a valve that

determined the rate at which gasoline could flow into the engine. More gasoline, bigger explosions inside the engine, more power, so more speed. Today, the gas pedal doesn't really do anything mechanically, except provide an input to a computer inside the car, whose job is to control the fuel injection system. Your input is combined with other information about the car's environment to operate smaller valves, one per cylinder, that control the gas/air mixture more precisely.

The brake pedal has had a similar history. Originally, your foot directly provided the power to push the brake pads against the wheels. Then a new mechanism was developed, preserving the interface -- push here to slow down -- but now using the pressure from your foot to operate a hydraulic system that does the hard work of pressing the pads against the wheels. But there was one important difference. The first "power brakes," like the modern gas pedal, completely eliminated the mechanical linkage between the pedal and the actual brakes. But after a few accidents in which people couldn't stop their cars because the engine died, this design was modified. Today you have "power assisted brakes," which means that your foot both operates a hydraulic cylinder and /also/ directly puts pressure on the brakes. If the engine fails, you have to push a lot harder to stop the car, but at least it's possible. The latest development, anti-lock brakes, actually lets a computer in the car override your pressure on the brake pedal if you are in danger of putting the car into a skid by trying to stop too abruptly.

But the point is that drivers who aren't particularly interested in cars don't have to know any of this. All they have to know is that you push the pedal on the right to speed up, and the one on the left to slow down! This /interface/ has survived through several generations of underlying technology, because it's a good interface -- simple but expressive. Car engineers could have made each generation of new technology more visible to drivers, with lots of knobs and switches and readouts, but they wisely refrained, and stayed with the abstraction -- the interface -- developed a century ago.

How does abstraction work in computer programming? We'll be revisiting this question all semester. But, for a starting point, think about the blocks in the Snap! menu. Each block names a simple intention, comparable to "speed up" or "slow down" in a car. But the mechanisms that make those blocks do their job are actually doing very complicated detail work. For example, "move 10 steps," above the abstraction barrier, just tells the sprite to move over a little. But the sprite isn't a real thing; it's a collection of colored dots drawn on a computer screen. "Move 10 steps" really means to erase each of those dots, then redraw them all in a different position. And the "glide" block is even more complicated; it has to erase and redraw many times, moving just a tiny bit each time.

But if the abstraction is working well, you're not thinking about tiny dots of light at all, while working with Snap!. You're moving Alonzo!

Snap! lets you, the user, build your own blocks -- your own abstractions. You can take a complicated sequence of actions, wrap them up, and present them to another user -- or to yourself thinking at a higher level of abstraction -- as a new interface, allowing you to command a new behavior without having to know anything about the detailed implementation.

About the Author: Brian Harvey is a Senior Lecturer SOE at UC Berkeley and is retired as of July 1, 2013. He continues to work on a variety of education-related projects at the University.

Brian has taught many of the lower division computer science courses at Berkeley, as well as one called the Social Implications of Computing. He is the faculty advisor of the Computer Science Undergraduate Association and the Open Computing Facility. Brian is also interested in the use of computers in pre-college education as he previously was a high school teacher and was involved in

the development of the Logo programming language. More recently he has been helping develop Snap!, a visual programming language.