Programming language and
API designers can learn a lot from the
field of human-factors design.

KEN ARNOLD, INDEPENDENT CONSULTANT

Programmers are People, Too

I would like to start out this article with an odd, yet surprisingly uncontroversial assertion, which is this: programmers are human.

I wish to use this as a premise to explore how to improve the programmer's lot. So, please, no matter your opinion on the subject, grant me this assumption for the sake of argument.

One could go many places with this premise, but what concerns me here is the following consequence: if programmers are human, then the study of human factors ought to shed some light on how to design the most basic tools programmers use: APIs, programming languages, and the like.

Just to be clear, when I say *human factors*, I am referring to the design practices for *all* things people use, not just human-computer interaction (HCI). People who design doorknobs, car dashboards, graphical layouts, and martini shakers have a long head start on HCI, and good HCI practitioners look to this experience to inform their own work. Now it's our turn.

The human-factors folks have been doing research and establishing approaches and rules of thumb for years, but rarely if ever do we directly apply them to our API designs. This is a major failing. When we discuss design principles, we should leverage all the tools that can be made to serve. Tools that help us understand how to make something more usable with fewer errors and more comfort—the basic focus of human-factors research—should be embraced and adapted. I

firmly believe that importing the decades of this research into how we approach API and other software designs can make a rapid and impressive improvement in productivity and reduced bug rates.

FIRST EXAMPLE: PROGRESSIVE DISCLOSURE

As API designers we may talk about a trade-off between simplicity and power, but we can look to human factors to find tools to address it. One such tool is *progressive disclosure*: rather than put expert-level features at the same level as basic ones, you put them behind a door marked “expert.” Consider the car, where most folks can work with the controls in the cabin but, with a few exceptions (such as washer fluid), they never need to open the hood. You know the engine is there, but most people just leave that part alone. A few experts do open it up and adjust the car with these expert-level controls, and if you want to do it, you know where to look. But if operating a car presented you with the entire engine as a control structure, most humans would find driving dauntingly complex, even if they just had to learn not to look at “all them wires and tubes and valves and things.”

You see progressive disclosure fairly often in GUI designs, typically as an Advanced or Expert button. This might expose settings for Web proxies in a browser or a rarely needed configuration for adjusting color balances on a printer.

We could do the same thing with APIs. For example, the Java Swing `JButton` class—the basic

Programmers are People, Too

GUI button—has well over 100 methods. But if you think about GUI buttons, there are only a half-dozen things you typically care about most of the time. When presented with this massive complexity, how are you to tell where to start? Should you be adjusting the preferred or minimum size? What happens if you change the text? Do you need to fire those things called “change listeners,” and if so, which kind?

To start getting this nonsense under control, you can start by breaking the `JButton` methods down into three major groups:

- The basic methods needed by most of us (set text/icon, add a callback listener for when the button is pressed).
- The expert methods for folks who want to get in and tweak (the font, color, border style).
- The integration methods needed by folks who write GUI widgets in Swing, who are a very rare breed of programmer (sizes, firing off change listeners, re-parenting).

Now we could use progressive disclosure to help reduce the complexity of that `JButton` class: put the expert stuff in an object returned by a `getExpertKnobs()` method, and the graphics subsystem hooks in an object returned by a `getIntegrationHooks()` method, and you would be left with a button API that had just a handful of methods—the basic methods we all need.

The overall system would be a bit more complex if you simply count methods and types, but you should not fall into the trap of thinking all methods are the same. If you

look at what is presented to the programmer, the system would go from well over 100 methods to consider when using a button to fewer than 10. All the power would be present and available, but only when you wanted it. For the most part, you could ignore it completely because they would be behind “doors” labeled, “You probably don’t need to look here.”

If one were sufficiently brutal in exiling functionality to behind the doors, the API would almost be self-explanatory to the nonexpert user.

(To be fair, Swing is not unique in its complexity. This malaise seems to infect most GUI

frameworks of note, including the Microsoft Foundation Classes, the X11 toolkits, and so on.)

What progressive disclosure does here is reduce the *surface area*, a term I use to talk about how much users have to understand about something before feeling confident they can use it. The more methods in an API, the more a user has to read through and understand before knowing, for example, if changing the label is something as simple as calling `setText` or if one of those other methods affects this. The larger the surface area, the harder it is to learn to use the tool well for even simple applications.

SECOND EXAMPLE: THE AUDIENCE

Another human-factors approach that could be applied to programmer tools is to consider the *audience* (sometimes called the *user model*). A good UI design has a notion of audience: What kinds of people are we designing for? What do they know, not know, and expect? If a UI has a clear and consistent audience, it is easier for users to make predictions about what the system will do. This reduces unpleasant surprises and means that the users can guess how to do things without reading the documentation (which they hardly ever do). Sounds like a pretty good feature in a programming language or API, doesn’t it?

This can work even if the user is not a part of the targeted audience. If you present a consistent model of some known kind of user, other kinds of users can potentially adapt, consciously or otherwise thinking via the role of the target user.

If you apply this question to C++, for example, you will find major problems. Sometimes C++ believes that programmers like to have the compiler do obvious things on their behalf and correct the compiler when it’s wrong. C++ will, for example, generate a default copy constructor that can create a copy of your type of object. Yes, the default copy constructor can be wrong, but it is right fairly often, so the language gives you a hand. If it’s wrong, any C++ programmer should know enough to provide a replacement that does work properly.

At other times C++ has a different model of the user: it believes that it should do nothing that stands a chance of being wrong, no matter how many times an obvious assumption would be right. One example is what *doesn’t* happen if you define how to check if two objects are equivalent (overriding the `==` operator). It seems obvious that knowing if two objects are equivalent would let you know if they are *not* equivalent. That is, knowing how to test `x == y`, then `x != y` can be thought of as `!(x == y)`.

But C++ does not define `!=` for you because it is possible in some odd cases that this is not true, though the

number of such cases must be infinitesimal. It is surely wrong in many fewer cases than the default copy constructor is wrong.

So is C++ a language that helps you with obvious things that are usually correct, or one that is more concerned with formal correctness? As an example, take a guess: if you define a constructor that can create a new `Foo` object from a `Bar` object, does C++ use it to automatically define how `foo = (Foo) bar` will work? Put another way, does it override the `=` operator for assigning a `Bar` to a `Foo`? Is your gut instinct different for these two questions?

This kind of inconsistency makes a system harder to learn and harder to use correctly. Instead of a consistent, comprehensible model of the audience that the user can grasp, you have instead a large collection of special cases where you can neither predict nor use some rule of thumb as a hint.

WHERE TO GO WITH THIS?

When you use a GUI that has profound inconsistencies about what level of expertise and control you have, or presents you with a dialog box with 100 options when you just want to change the display font, is it fair to say that the designer made a mistake? Then why not judge the API or language designer the same way?

The problem is that these basic rules of thumb and experiences in human factors are not part of the design discourse about the primary tools we build for ourselves: programming languages and APIs. We ought to change this. If we can learn from human factors to do things better, we will be able to write code with fewer bugs. And because it will be easier, we can spend less time learning *how* to do stuff and more time *doing* it. The study of human factors is for *humans*, not for GUIs. And (see above) programmers are humans.

So let's proceed from the following theorem: An API or programming language is a user interface to the programming model that is being presented to the user (the programmer).

Let's look at several typical rules of thumb for human factors and see how they might be applied.

Similar things should look similar. If two things mean the same or very similar things, they should be presented in ways that express that similarity. In a GUI this would mean that no matter how many ways and reasons there are to open files, the basic interaction for opening files should be the same. For an API this might mean that if there are multiple things that can be started and stopped, then the same terms should be used for each kind of

thing. It would be important to pick, for example, “start” and “stop” and use them for all starting and stopping, rather than have some places where you end execution using “end” or “terminate” or “close” or “destroy.”

As a negative example, C has two rather different variable declaration mechanisms: one used for declaring function parameters (which is comma-separated), and another used for all other variables (which is semicolon-terminated). Although we've all learned this, it is one of those small things that you can stumble over. Everywhere but in function parameters you can declare that `x` and `y` have the same type via “`double x, y;`” but in method parameters it must be “`double x, double y;`”, which loses the connection between the types of variables. In the first form you can change coordinates to be held in `float` variables naturally, whereas in the second form you must change the types of `x` and `y` independently, as if it somehow might be reasonable to change only `x`.

Use forcing functions to prevent errors. You could think of this as the “This button turns on the bathroom light, and that button launches global thermonuclear war. Don't mix them up!” rule.

A *forcing function* makes the user do something that prevents (or makes unlikely) some type of mistake. Many cars, for example, will not let you take the key out of the ignition unless the car is in a correct gear. This makes it very unlikely that you will leave the car in neutral when you leave your car, only to watch it slip down the hill and into your mother-in-law's new Lexus.

In programming you can apply this to many areas. In many languages you can make it impossible to write certain kinds of incorrect code. In C++ the presence of `const` is intended for exactly this purpose. If the compiler simply won't let you make a modifying call on an object, and you hand out only `const` references to your objects, then users can't make the mistake of modifying something they shouldn't. (Well, at least not casually—you can cast away `const`, but this is clearly suspicious behavior, so you've at least made them do something that raises alarm bells. This is still a forcing function.)

Another example is quite common: if you don't have the right token, you can't perform a particular action. Any time you see a function that requires a certain type of handle that you can only get somewhere specific, you are being forced to do something first. In Java, for example, you can open with a `File` object, but you can close only the resulting `FileInputStream` object, not the `File` object. This makes it impossible to write code that closes a file without opening it first.

Think from the user in. Interestingly, the forcing func-

Programmers are People, Too

tion principle also demonstrates the difficulty of actually applying human-factors principles. Reading a file is a very different category of action than destroying existing files, which risks much more damage. The normal forcing function approach would be to make it more awkward to destroy a file than to read one. Maybe we could make destroying files a multistep process or use longer function names that would never be accidentally typed.

Programmers, however, are not just humans, but notoriously lazy and problem-solving humans. In the face of such a design, programmers would write single methods that combine all the steps, or create shortcuts for long method names, or otherwise remove the “problem” of awkward access. I know I would. So in applying this principle we are limited by both our “materials” (programming languages) and our audience.

In other words, your audience has particular features: habits, assumptions, knowledge (correct and wrong), and customs. Programmers are human. They also are particular kinds of humans. And the user of some particular API is yet further specifiable.

The one thing they are almost certainly *not*, however, is you. You are thinking about how to solve the problem, the merits of various approaches, the detailed trade-offs between one algorithm and another, the literature on doing the work involved, and so on. By the time you are done figuring out what to do, you are likely one of the most expert folks in the world on the task you are trying to help people with.

And your users? They just want it to happen. They will have varying degrees of expertise, but even the most expert will have one reason to use your system: so they can think about it *as little as possible*. If you have done your job well, their use of your code will be only as large as you make them make it. Remember, if possible, your users would want you to provide a single command: *dwim* (do what I mean).

So you must think like your user: think *in* to the problem from their desires and viewpoints rather than *out* from your sophisticated understandings of solutions and mechanisms. Your design should ask, “What does the user

want to do?” instead of “How can I present Whilfolze’s 3rd Equation to optimize applications of Guilemorting’s Principle?” If Whilfolze and Guilemorting have useful things to say about solving the user’s actual problem, you should apply their insights instead of making the user tell you how to apply them. The user should say “solve this,” and your code should use Whilfolze and/or Guilemorting if that’s a good thing to do.

Consider the difference between a car designer asking, “How does the user control the car?” vs. “How can the user adjust the fuel intake, injectors, cylinders, spark plugs, fans, differentials, etc.?” The first question is much more likely to produce a design usable by car-ignoramuses like me because the question it asks is one I will ask. Approaching the design as a way to fiddle with the complex car parameters will almost certainly produce a more complex design with more alternatives and features presented to me. I don’t want that. If I want to become a car expert, I will open the hood. I just want the thing to work.

One way to approach the problem is this: write the pseudocode your users would want to write, and then make it work with as few additions as possible.

The primary questions are: What problems are users trying to solve? What kinds of things do users have in hand when they want to solve the problem? How does the user think of the problem? What must the user tell me and what can I deduce for myself? This starts with a good definition of who your users are, of course, which is a task most designs seem to ignore.

Remember that users have a notorious history of thinking they know how something should be done and being wrong. One classic example is the `register` keyword of C. The idea was that the user (in this case a programmer) would tell the compiler which local variables should be stored in fast-access processor registers because the programmer knew what was critical. It turned out that compilers were almost always smarter about register allocation than users could ever be. The `register` keyword quickly became advisory, and by now I suspect that all C/C++ compilers just ignore it, snickering.

DIVING HEADFIRST INTO HUMAN FACTORS

Good computer design has been a topic since computers were a Lovelace-analyzed gleam in Babbage’s eyes. Experience has improved our understanding of what makes good design. We have even reached into some human-factors-influenced fields for ways to describe our understanding, such as using architecture’s

notion of pattern languages for our own design patterns (and anti-patterns).

What we have rarely done, however, is reach into the field of human factors for its insights and apply them to what we design. We may be the last set of folks to realize that our users actually are humans, and so directly learn from what is known of how to design things for humans. API and language designers should dive headfirst into the field of human factors and drag its lessons back into what they do. And we programmers—we users—should demand it. ☹

FURTHER READING

- Norman, D. A. 2002. *The Design of Everyday Things*. Basic Books. Possibly the best book on designing for people. Consider the humble door—how many ways can a designer screw *that* up?
- Tufte, E. R. 2001. *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press. An excellent work on the human factors of information display. By analogy, most of what Tufte shows in his book can be applied to API and programming language design. As a human interaction tool, an API conveys information about what I can do and how to do it.
- Raskin, J. 2000. *The Humane Interface*. Addison-Wesley

Professional. Thinking about human-computer interfaces from a human and humane standpoint. A lot of excellent, unique thinking. (*Unique* means you may find things you absolutely hate, but you will think about human-centric design in new and careful ways as you decide *why* he's wrong when you hate it.)

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

KEN ARNOLD, a freelance consultant, was the original lead architect of JavaSpaces. He is a leading expert in object-oriented design and implementation, and is an author of several books and articles on Jini, Java, and design principles. Before working at Sun, Arnold was part of the original Hewlett-Packard architectural team designing CORBA, several user interface and Unix projects at Apollo Computers, and molecular graphics at the University of California, San Francisco. In olden days, he was part of the 4BSD team at U.C. Berkeley, where he created the curses library package for terminal-independent screen-oriented programs, and was co-author, with Mike Toy and Glen Wichman, of the computer game Rogue. He received his A.B. in computer science from U.C. Berkeley in 1985.

© 2005 ACM 1542-7730/05/0600 \$5.00