The Linguistic Command Line

Aza Raskin
Humanized | aza@humanized.com

I'm a Mac baby. Twenty-three years ago, I was born. So was the Macintosh.

Looking back, I haven't fundamentally changed since my lower-than-your-kneecap days. I'm just a larger, differently proportioned version of my younger self. Unfortunately, the same thing is also true for the GUI: It's matured, but hasn't fundamentally changed in the past 23 years. We are still stuck juggling windows in a time-wasting dance to find the application we need to get a task done.

Look at a screenshot of the original Macintosh, and compare it with one of the latest versions of OSX. With the exception of some new gadgets and some smoke and mirrors, much of the interface is still the same. Sure, there's Spotlight, a way to find the stuff we lose in our morass of folders and badly named files, and Expose, a kludge to help us wade through our windows. But these are quick-fix patches on a sinking metaphor. They keep us in an unsatisfying marriage to our windows and applications.

When we want to get something done, we still have to slog to the application that does it, dragging our content screaming and kicking behind us. Everywhere we look, our tasks are needlessly compartmentalized, and we are left schlepping hither and thither. Take the example of writing and posting a presentation to a website: A simple task like this requires Photoshop to edit the images, Excel to create a spreadsheet, PowerPoint to compile the presentation, TextEdit to create the appropriate Web pages, an FTP client to upload it to the internet, and Firefox to view it. Most of our time is spent just in moving content from one application to another. Then there is the frustration from errors caused by the cognitive overhead required to switch applications, each of which has its own idiosyncrasies. The same keyboard shortcut, Control-D, changes your font in Word, but creates a bookmark in Firefox. Trying to remember whether the methods and shortcuts we've learned in one application work in another is a game of chance. Compartmentalization of tasks via uncoordinated applications frustrates our habits and wastes our time. We shouldn't need to think about which application we are using to know how to spell check, look up word definitions, change font size and undertake other common tasks.

Applications are the cause of another computer woe: software bloat. Although bloat is partially due to sloppy coding induced by ever increasing computing power, compartmentalization forces code redundancy. Tasks rarely fall completely within a single compartment: Word has an underpowered drawing package, CAD packages have underpowered text-layout engines, and even Google has a calculator. Thus we arrive at the modern monolithic application mired in mediocre implementations of subtasks. My computer has eight copies of spell check; each features a different version of the English language, most lack a decent interface, and less than half recognize my name. When application compartmentalization is removed, so is the unnecessary code overlap: Disk and memory footprint drops, development time decreases, and usability and reliability increase.

Applications should take a lesson from services on the Internet, or even old command-line utilities. Instead of reimplementing common pieces of functionality, applications should outsource that functionality to an OS-level service, some other local service, or an Internet service. As a user, imagine if you never needed to teach your name to yet another spell check, or—as a developer—implement another instance of spell check.

My father, Jef Raskin, was a pioneer in early interface development. His work included developing the first Macintosh at Apple, and inventing "click and drag" and other ubiquitous interface metaphors. He's the reason why we use the word "font" for what should more correctly be called "typeface." Toward the end of his career, he outlined many of his ideas—both radically different and radically better—in his book, *The Humane Interface*. The challenge between task complexity and selection simplicity was included in his call for change. "By applying the concept that a system should not be more complex than your present needs, and by allowing the system to increase its power incrementally, the dream of providing products that are truly simple initially can be achieved, without their being made to merely look simple, and without impairing their flexibility," he wrote.

My father also discussed the conflict between seamless user tasks and divided applications: "Instead of a computer's software being viewed as an operating system and a set of applications, then, the humane interface views the software as a set of commands." That is, functionality should be learned on an as-needed basis, and be available anywhere in the system, regardless of the dividing lines between the individual applications. Applications interfere with the idea of as-needed functionality. The learning curve for each application can be overcome with use, but if we need to use any additional piece of functionality not provided in our main application, we must learn an entire *other* application that provides it. This makes a simple task such as editing a document with pictures unnecessarily difficult.

I've tried to follow through on these ideas in my own work and to design an interface system that works beyond the boundaries of an individual application. The challenge is that the current software economy is tied to the concept of applications. Disparate applications aren't going to disappear. Providing services, however, allows us to granulate that software economy. If you don't need all of the functionality of Photoshop, you can just buy the photo-editing features you need as a service. Instead of arguing for the abolition of applications, we can champion services with a universal way of accessing them. That way, we can snap our fingers and have the functionality we need, regardless of the application we happen to be using. This shift also reframes the interface challenge, which then becomes this: If all functionality is available to us anywhere, at any time, how do we tell the computer which particular piece of functionality we want?

We're going to need a universal way to access those thousands of possible services we might want to perform on our selected data—from calculating the sum of the values, to performing a Google search on the text, to changing the size at which it is displayed. Current paradigms for accessing this functionality don't scale to how we really work—i.e., *across* applications, not *within* them.

Imagine grafting together the endless menus for PowerPoint, Mathematica, Firefox, and Photoshop. Certainly, there would be some overlap, but the result would still be a Medusa's head of seething submenus. It would be laborious to find anything in such a monstrosity, and inefficient to manually select a menu even if we knew where to find it. We headed toward graphical menus initially because they made all options visible, by allowing recognition of an option instead of forcing the recollection of an option. Jef Raskin and the rest of the Macintosh team found that menus worked well. In hindsight, they worked well because of the limited number of available options.

With increasing scale, the menu metaphor falls short. While the recognition solution can work in a single application with a restricted set of options, it fails when we look at real tasks that cross application boundaries. For example, the argument that menus provide visibility and findability breaks down when applied at a large scale because they become slow to learn and use. Similarly, keyboard shortcuts—patches meant to increase the speed of menus—also do not scale. The keyboard features a finite number of keys and even fewer mnemonic matchings of keys to functionality.

Icons fare worse than menus and keyboard shortcuts. The abstract concepts inherent to detailed functionality are difficult to represent visually. Microsoft Word attempts to use icons to represent some of the basic functionality of text processing, but this method doesn't work out well. Can you figure out what each of these icons does?

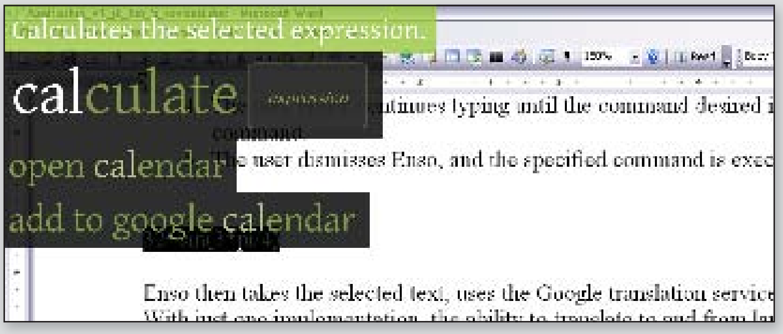
Even if you can recognize a few of the examples, your recognition is learned. Those icons, no matter how self-evident Microsoft would like them to be, require language (in the form of tooltips) to actually explain what they do. If simple text-formatting operations fail so greatly, how can icons be designed to express the full range of functionality that services provide? And how would we page through that giant lexicon of icons to quickly to find the one we're looking for?

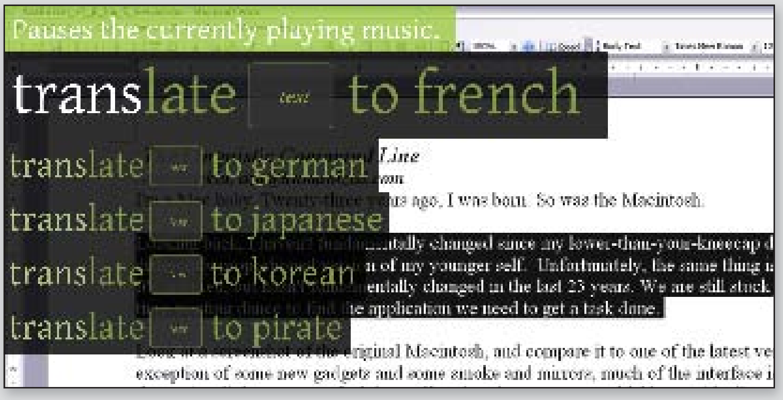
The "window, icon, menu, pointing-device" or WIMP paradigm, has its limits, and these limits are now growing clearer as the complexity of modern computing unfolds.

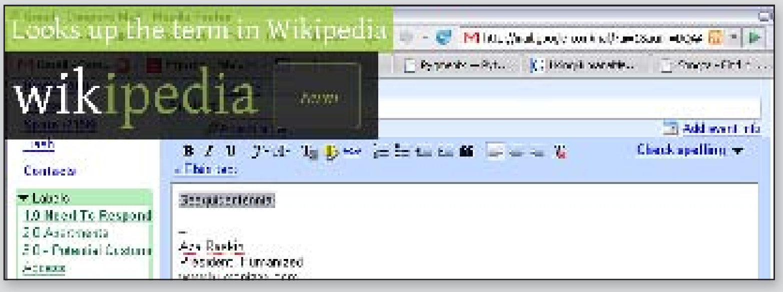
The Linguistic Command Line. Pictionary is a game in which one tries to represent objects, places, and abstract thoughts in image form. It's a hard game, and there is no reverse game. Because information density is drastically greater in pictures than in prose, a picture is indeed worth a thousand words, but only when they describe a concrete visual like a graph or a portrait. In the realm of the abstract, pictures fail. How would you pictorially represent Marxism? You could try a picture of Marx, but that doesn't distinguish between the person and the school of thought (and requires your viewer to know what Marx looks like; otherwise it's just a guy with a beard). Words can capture abstractions that pictures cannot because language has an immense amount of descriptive and differentiating power. Abstract thoughts are exactly represented by the words that give them names. It is this power that comes to the rescue in specifying functionality.

Standard GUIs, with their drop-down menus, check buttons, and tree-lists, cannot compare to the range of options that a text interface effortlessly provides. With just five alphanumeric characters, we can choose one out of 100,000,000 possible sequences. And choosing any one sequence is, in approximation, as fast as choosing any other (typing five characters takes roughly one second). It's difficult to come up with a non-text-based interface that can perform as well.

Using language to access functionality brings to mind the old-form command line, which is still one of the most powerful interface paradigms we have for controlling our computers. Although command lines are hard to learn







Linguistic Command Line Interfaces

Two current programs attempt to deliver linguistic command-line interface to users: Humanized's Enso, and Blacktree's excellent Quicksilver. Enso uses a more natural-language syntax, and works like this:

1. At any time, the user presses an activation key to call up a text-entry area.
2. Next, the user begins typing what they want to do. For instance, "translate to Japanese."
3. As the user types, Enso autocompletes to the most likely command, and related suggestions appear below the typed text.
4. The user either continues typing until the command desired is specified, or arrows to a preferred command.
5. The user dismisses Enso, and the specified command is executed.

Enso then takes the selected text, uses the Google translation service, and places the results back into the text. With just one implementation, the ability to translate to and from languages is available anywhere on the computer, always with the same interface, and accessible in a few mnemonic keystrokes. Enso uses copy and paste as the graphical equivalent to standard out and standard in, allowing it to speak to almost any application in an implementation-agnostic manner. Because of the power of language, adding a large number of commands scales well. It's always easy to get to the functionality desired.

and difficult to troubleshoot when things go wrong, nothing is intrinsically hard or difficult about using language to tell the computer what to do. The hard part of the old command lines was memorizing command names as unfathomable as the origin of Stonehenge. Worse, remembering command-line options is like bobbing for apples in a cement mixer. I still have to ask my coworkers which flags are needed for untarring a gzipped file. (It's "tar-xfvz." Gee, how could I forget?)

But maybe this confusion isn't the fault of command-line interfaces in general; maybe it's just the command lines we're used to. If commands were memorable, and their syntax forgiving, perhaps we wouldn't be so scared to reconsider these interface paradigms. Perhaps the linguistic command line is the future of computing.

The move back toward using language for selection started with Web searching. Google placed the capstone when its name became the household verb for "typing to find what you want." In fact, googling is almost always faster than wading through a bookmark menu or a categorical listing, which again indicates that something is wrong with using menus as a mechanism for accessing large quantities of data. After the Web, search came back to the desktop. OSX, Linux, and now Vista have integrated desktop searches that make searching the computer as convenient as searching the Web. Now, with a few memorable keystrokes, we can find what we are looking for. This stands in stark contrast to racking our brains to figure out where, in the jumble of files and folders, we put a document. This bears repeating: It is often easier to use a desktop search than to find something placed in the computer for safekeeping.

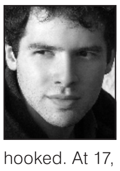
Other places on the Internet harness the power of language to good effect. The quick-add features of 30boxes.com and Google Calendar are my favorite examples: They forgo the clunky and time-consuming forms of the standard database, and opt instead for the utter simplicity of typing an event's information—for instance, "Sunday dinner at 7:30 p.m. with Asa Jasa." The quick-add feature doesn't even feel like an interface, which is the highest compliment an interface can get. The better an interface is, the less it's noticed.

Even Microsoft Word has a nice example of a domain-specific, linguistic command line hiding in its print dialog. When choosing which pages to print, you can simply enter the pages you want as text—e.g., "1-4, 7, 15-20." This means to print pages 1 through 4, page 7, and pages 15 through 20. Imagine how difficult this type of input would be to design as a more standard GUI interface.

Now imagine using a drop-down menu to select the one website—out of the 100 million websites in existence—to visit. Ludicrous! How do we actually surf to a site? By typing an address into the address bar, aided by an autocomplete that quickly enables us to visit addresses we have previously visited. When we want to go to the mail "application," we type in "gmail.com"; when we want to open a news "application," we type in "nytimes.com." On the old Unix command lines, we would type "pine" and "rn. " The address bar is just a primitive command line, a command line that our grandmother can—and does—use.

Because natural language processing is still far off on the horizon, the full linguistic command line—one that provides access to all functionality at any time—must rely on structured syntax and autocomplete to guide the user to known commands. That is, the linguistic command line needs to help the user get to the right command, instead of letting the user blithely type in a vacuum.

Just as the GUI has grown in the past two decades, so will the linguistic command line. We are seeing it in its infancy. Enso is one example of a linguistic command line: It allows users to issue commands to an operating-system service regardless of the application they're using (see sidebar). Other approaches to the problem exist, and finding the best ones will take time. Regardless of how we finally tackle the problem, it's time for a new, human-centric command line to make a comeback for language-based interfaces—a command line that finally lets us just do want we want to do, when we want to do it, wherever we are. How humane.



ABOUT THE AUTHOR Aza gave his first talk on user interface at age 10 with his father at the local San Francisco Bay Area chapter of SIGCHI and was hooked. At 17, he was speaking and consulting internationally; at 19, he co-authored a physics textbook because he was too young to buy alcohol; at 21, he started drinking alcohol and co-founded Humanized. Aza enjoys playing the French Horn, which has taken him all over the world, and puttering in his lab, which has given him a greater respect for physics.