

Please email comments to [bret@worrydream.com](mailto:bret@worrydream.com).

This draft was released March 15, 2006. A couple examples are missing, and a few are out-of-date. I would appreciate any feedback on the content, presentation, or what exactly I should do with this thing I wrote.

**Firefox and IE** cannot print this webpage. If you like reading on paper, please [download the PDF](#).

---

# Magic Ink

INFORMATION SOFTWARE AND THE GRAPHICAL INTERFACE

by Bret Victor

## Abstract

The ubiquity of frustrating, unhelpful software interfaces has motivated decades of research into “Human-Computer Interaction.” In this paper, I suggest that the long-standing focus on “interaction” may be misguided. For a majority subset of software, called “information software,” I argue that interactivity is actually a curse for users and a crutch for designers, and users’ goals can be better satisfied through other means.

Information software design can be seen as the design of *context-sensitive information graphics*. I demonstrate the crucial role of information graphic design, and present three approaches to context-sensitivity, of which interactivity is the last resort. After discussing the cultural changes necessary for these design ideas to take root, I address their implementation. I outline a tool which may allow designers to create data-dependent graphics with no engineering assistance, and also outline a platform which may allow an unprecedented level of implicit context-sharing between independent programs. I conclude by asserting that the principles of information software design will become critical as technology improves.

Although this paper presents a number of concrete design and engineering ideas, the larger intent is to introduce a “unified theory” of information software design, and provide inspiration and direction for progressive designers who suspect that the world of software isn’t as flat as they’ve been told.

## Scope and terminology

“**Software**,” as used here, refers to user-facing personal desktop software, whether on a native or web platform. “**Software design**” describes all appearance and behaviors visible to a user; it approaches software as a *product*. “**Software engineering**” implements the design on a computer; it approaches software as a *technology*. These are contentious definitions; hopefully, this paper itself will prove far more contentious.

computer; it approaches software as a *technology*. These are contentious definitions; hopefully, this paper itself will prove far more contentious.

## Contents

### *What is software?*

**Of software and sorcery.** *Is “interaction design” the cure for frustrating software, or the disease itself?*

**What is software design?** *Software is not a new and mysterious medium, but a fusion of two old ones.*

**What is software for?** *People turn to software to learn, to create, and to communicate.*

**Manipulation software design is hard.** *Creating software for creating is tricky business.*

**Most software is information software.** *People spend more time learning than creating.*

### *Graphic design*

**Information software design is graphic design.** *People learn by looking. Looks are all that matters.*

**Demonstration: Showing the data.** *Redesigning Amazon as an information graphic.*

**Demonstration: Arranging the data.** *Redesigning Yahoo! Movies as an information graphic.*

### *Context-sensitivity*

**Context-sensitive information graphics.** *Software trumps print by showing only what’s relevant.*

**Inferring context from the environment.** *The outside world can suggest what’s relevant.*

**Inferring context from history.** *Memories of the past can suggest what’s relevant.*

### *Interactivity*

**Interactivity considered harmful.** *The user can suggest what’s relevant, but only as a last resort.*

**Reducing interaction.** *Approaches to easing the pain.*

**How did we get here?** *The popular focus on interactivity is a vestige of another era.*

### *Intermission*

**Case study: Train schedules.** *Designing a trip planner as an information graphic.*

**Demonstration: Trip planning redux.** *Redesigning Southwest Airlines as an information graphic.*

### *Changing the world*

**Designing the information software revolution.** *Five steps from artifice to art form.*

**Designing a design tool.** *Dynamic graphics without the programming.*

**Engineering inference from history.** *How software can learn from the past.*

**Engineering inference from the environment.** *A platform for implicit communication between software.*

**Information and the world of tomorrow.** *Why all this matters.*



## Of software and sorcery

*A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can **answer questions**. It can **affect the world** by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells.*

*—Abelson and Sussman, Structure and Interpretation of Computer Programs (1984)*

Merlin had it easy—raising Stonehenge was a mere engineering challenge. He slung some weighty stones, to be sure, but their placement had only to please a subterranean audience whose interest in the matter was rapidly decomposing. The dead are notoriously unpicky.

Today's software magicians carry a burden heavier than 13-foot monoliths—communication with the living. They often approach this challenge like Geppetto's fairy—attempting to instill the spark of life into a mechanical contraption, to create a Real Boy. Instead, their vivified creations often resemble those of Frankenstein—helpless, unhelpful, maddeningly stupid, and prone to accidental destruction.

This is a software crisis, and it isn't news. For decades, the usability pundits have devoted vim and vitriol to a crusade against frustrating interfaces. Reasoning that the cure for unfriendly software is to make software friendlier, they have rallied under the banner of "interaction design," spreading the gospel of friendly, usable interactivity to all who would listen.

Yet, software has remained frustrating, and as the importance of software to society has grown, so too has the crisis. The crusade marches on, with believers rarely questioning the sacred premise—that software must be interactive in the first place. That software is meant to be "used."

I suggest that the root of the software crisis is an identity crisis—an unclear understanding of what the medium actually *is*, and what it's *for*. Perhaps the spark of life is misdirected magic.

## What is software design?

A person experiences modern software almost exclusively through two channels:

- She reads and interprets pictures on a screen.

A good introduction to the crisis is Alan Cooper's [The Inmates Are Running The Asylum](#) (1999). Essential concepts of interactive design are presented in Don Norman's [Design Of Everyday Things](#) (2002), Jef Raskin's [The Humane Interface](#) (2000), and Cooper's [About Face](#) (2003).

- She points and pushes at things represented on the screen, using a mouse as a proxy finger.

Thus, software design involves the design of two types of artifact:

- Pictures.
- Things to push.

These are not brave new realms of human endeavor. We share the blood of cavemen who pushed spears into mammoths and drew pictures of them in the living room. By now, these two activities have evolved into well-established design disciplines: graphic design and industrial design.

**Graphic design** is the art of conveying a message on a two-dimensional surface. This is a broad field, because people have such a variety of messages to convey—identity, social status, emotion, persuasion, and so on. Most relevant to software is a branch that Edward Tufte calls *information design*—the use of pictures to express knowledge of interest to the reader.\* Some products of conventional information graphic design include bus schedules, telephone books, newspapers, maps, and shopping catalogs. A good graphic designer understands how to arrange information on the page so the reader can ask and answer questions, make comparisons, and draw conclusions.

\* Edward Tufte, [The Visual Display of Quantitative Information](#) (2001).

When the software designer defines the visual representation of her program, when she describes the pictures that the user will interpret, she is doing graphic design, whether she realizes this or not.

**Industrial design** is the art of arranging and shaping a physical product so it can be manipulated by a person. This too is a broad field, because people work with such a variety of objects—cutlery to chairs, cell phones to cars. A good industrial designer understands the capabilities and limitations of the human body in manipulating physical objects, and of the human mind in comprehending mechanical models. A camera designer, for example, shapes her product to fit the human hand. She places buttons such that they can be manipulated with index fingers while the camera rests on the thumbs, and weights the buttons so they can be easily pressed in this position, but won't trigger on accident. Just as importantly, she designs an understandable *mapping* from physical features to functions—pressing a button snaps a picture, pulling a lever advances the film, opening a door reveals the film, opening another door reveals the battery.

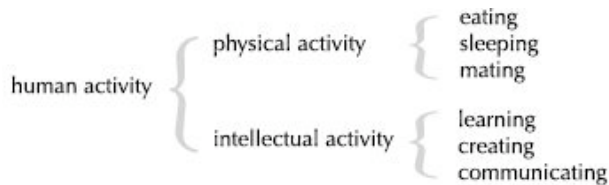
Although software is the archetypical non-physical product, modern software interfaces have evolved overtly mechanical metaphors. Buttons are pushed, sliders are slid, windows are dragged, icons are dropped, panels extend and retract. People are encouraged to consider software a machine—when a button is pressed, invisible gears grind and whir, and some internal or external state is changed. Manipulation of machines is the domain of industrial design.

When the software designer defines the interactive aspects of her program, when she places these pseudo-mechanical affordances and describes their behavior, she is doing a virtual form of industrial design. Whether she realizes it or not.

The software designer can thus approach her art as a fusion of graphic design and industrial design. Now, let's consider how a user approaches software, and more importantly, *why*.

## What is software for?

Software is for people. To derive what software should do, we have to start with what *people* do. Consider the following taxonomy of human activity:\*



At the present, software can't do much for physical needs—if your avatar eats a sandwich, you remain hungry. But people are increasingly shifting their intellectual activities to the virtual world of the computer. This suggests three general reasons why a person will turn to software:

- To learn.
- To create.
- To communicate.

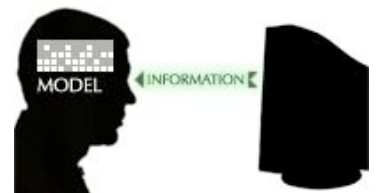
I propose that software can be classified according to which of these needs it serves. I will call these categories *information* software, *manipulation* software, and *communication* software.

**Information software** serves the human urge to learn. A person uses information software to **construct and manipulate a model that is internal to the mind**—a mental representation of information. Good information software encourages the user to ask and answer questions, make comparisons, and draw conclusions. A person would use recipe software, for example, to decide what to cook for dinner. She would learn about various dishes (where “learning” could be as informal as a quick skim for something tasty that contains ingredients on hand), compare her options, and make her decision. In effect, she is constructing an internal understanding of culinary possibilities, and mentally prodding this model to reveal the optimal choice. It's the same effect she would hope to achieve by consulting a recipe *book*.

**Manipulation software** serves the human urge to create. A person uses manipulation software to **construct and manipulate a model external to herself**—a virtual object represented within the computer, or a remote physical object. Some examples include software for drawing, writing, music composition, architectural design, engineering design, and robot control. Manipulation software can be considered a virtual *tool*—like a paintbrush or typewriter or bandsaw, it is used as an interface between creator and artifact.

**Communication software** serves the human urge to communicate. A person uses

\* There are any number of ways of breaking down the spectrum of human activity. I don't claim that the subdivision given here is *definitive*, but that it's *useful*. Consider it a set of basis vectors into the space of human activity. Different basis sets are helpful for reasoning about different problems, but they all describe the same space.



raw mechanics, communication can be thought of as *creating* a response to information *learned*—that is, the external model manipulated by the speaker is the internal model learned by the listener. Thus, this paper will simply treat communication software as manipulation software and information software glued together, and mention it no further.\* This design approach is widespread—email software typically has separate reading and writing modes; messageboards similarly segregate browsing and posting.

## Manipulation software design is hard

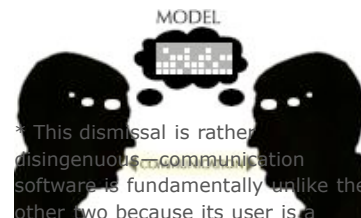
Manipulation software generally displays a representation of an object—the model—which the user directly manipulates with pseudo-mechanical affordances. Because manipulation is the domain of industrial design, manipulation software emphasizes industrial design aspects.

Consider a tool for laying out a small newspaper. The user will spend most of her time performing a number of pseudo-physical operations—writing, drawing, cutting, moving, rotating, stretching, cropping, layering—within a virtual space. The primary design challenge, just as with any industrial design, is to provide affordances that make these mechanical operations *available*, *understandable*, and *comfortable*. However, in a physical space, each operation would use a specialized tool. Designing a “mega-tool” that cleanly incorporates all operations (and flattens them into two dimensions, and uses only the gestures “click” and “drag”) is a significant challenge indeed.

Although manipulation is the focus, good manipulation software must provide superb visualization as well. This establishes the feedback loop that is critical for all creative activity—the manipulator must see the effects of her manipulation. Thus, manipulation software design is also a significant graphic design challenge.

For example, the newspaper editor needs to see what a page looks like—close-up, from a distance, and in relation to other pages—and how it *would* look in a variety of other configurations. She wants to see misspelled words, lines that are poorly justified or hyphenated, and widows and orphans. She wants to see columns that are short or overlong, and how they can be corrected by changing column width or leading. She wants to know what stories and ads are still on the table, their sizes, and how they can be fit in. She wants to know how recently and how often stories about a given topic have run, and how readers have responded. She wants to know past response to a given ad, as a function of the topics or authors of the stories it was coupled with. Finally, the presentation of all this information must not distract the editor from the primary task of manipulating the layout.

Furthermore, the industrial and graphic designs in manipulation software must be in intimate synergy, since it is the graphic design which describes how the object can be manipulated—the mechanical affordances are graphical constructs. Even more graphically challenging is manipulation of abstract objects, such as music or financial data, where the graphical representation must show not only what can be done with it, but *what it is* in the first place.\*



\* This dismissal is rather disingenuous—communication software is fundamentally unlike the other two because its user is a *group*, and a group as a whole can have different goals than any of its constituents individually. The considerations of social software design are well beyond the scope of this paper, but see [Clay Shirky's](#) essays, particularly [Social Software and the Politics of Groups](#) (2003).

\* As opposed to painting software, for instance, where the graphical representation can be the artifact itself. This is not a pipe, but it's close enough.

Because of these intertwined design challenges, the design of excellent manipulation software is *unbelievably* difficult, and mustn't be underestimated. Fortunately, for an enormous class of software, manipulation is not only largely unnecessary, but best avoided.

## Most software is information software

J.C.R. Licklider once examined how he spent his research time:

*In the spring and summer of 1957... I tried to keep track of what one moderately technical person [myself] actually did during the hours he regarded as devoted to work... About 85 per cent of my "thinking" time was spent getting into a position to think, to make a decision, to learn something I needed to know. Much more time went into finding or obtaining information than into digesting it. Hours went into the plotting of graphs, and other hours into instructing an assistant how to plot. When the graphs were finished, the relations were obvious at once, but the plotting had to be done in order to make them so... Throughout the period I examined, in short, my "thinking" time was devoted mainly to activities that were essentially clerical or mechanical: searching, calculating, plotting, transforming, determining the logical or dynamic consequences of a set of assumptions or hypotheses, preparing the way for a decision or an insight.\**

For Licklider and other early visionaries such as Vanevar Bush and Doug Engelbart,\* the ideal of the then-hypothetical personal computer was a brain supplement, enhancing human memory and amplifying human reasoning through data visualization and automated analysis. Their primary concern was how a machine could help a person *find* and *understand* relevant knowledge. Although they were generally discussing scientific and professional work, their prescience fully applies in the modern home.

Most of the time, a person sits down at her personal computer not to create, but to *read, observe, study, explore, make cognitive connections, and ultimately come to an understanding*. This person is not seeking to make her mark upon the world, but to rearrange her own neurons. The computer becomes a medium for asking questions, making comparisons, and drawing conclusions—that is, for *learning*.

People turn to software to learn the meaning of words, learn which countries were bombed today, and learn to cook a paella. They decide which music to play, which photos to print, and what to do tonight, tomorrow, and Tuesday at 2:00. They keep track of a dozen simultaneous conversations in private correspondence, and maybe hundreds in public arenas. They browse for a book for Mom, a coat for Dad, and a car for Junior. They look for an apartment to live in, and a bed for that apartment, and perhaps a companion for the bed. They ask when the movie is playing, and how to drive to the theater, and where to eat before the movie, and where to get cash before they eat. They ask for numbers, from simple sums to financial projections. They ask about money, from stock quote histories to bank account balances. They ask why their car

\* J.C.R. Licklider, "[Man-Computer Symbiosis](#)" (1960).

\* See Bush's paper "[As We May Think](#)" (1945) and Engelbart's paper "[Augmenting Human Intellect](#)" (1962).

isn't working and how to fix it, why their child is sick and how to fix her. They no longer sit on the porch speculating about the weather—they ask software.

Much current software fulfilling these needs presents mechanical metaphors and objects to manipulate, but this is deceiving. People using this software do not *care* about these artificial objects; they care about seeing information and understanding choices—manipulating a model in their heads.

For example, consider calendar or datebook software. Many current designs center around manipulating a database of “appointments,” but is this really what a calendar is for? To me, it is about combining, correlating, and visualizing a vast collection of information. I want to understand what I have planned for tonight, what my friends have planned, what’s going on downtown, what’s showing when at the movie theater, how late the pizza place is open, and which days they are closed. I want to see my pattern of working late before milestones, and how that extrapolates to future milestones. I want to see how all of this information interrelates, make connections, and ultimately make a decision about what to do when. Entering a dentist appointment is just a tedious minor detail, and would even be *unnecessary* if the software could figure it out from my dentist’s confirmation email. My goal in using calendar software to ask and answer questions about what to do when, compare my options, and come to a decision.

Consider personal finance software. Entering and classifying my expenses is, again, tedious and unnecessary manipulation—my credit card already tracks these details. I use the software to *understand* my financial situation and my spending habits. How much of my paycheck goes to rent? How much to Burrito Shack? If I give up extra guacamole on my daily burrito, will I be able to buy a new laptop? What is my pattern of Christmas spending, and will I have to cut back if I don’t take any jobs for a month? If I buy a hybrid car, how much will I save on gas? I want to ask and answer questions, compare my options, and let it guide my spending decisions.

Consider an online retailer, such as Amazon or Netflix. The entire purpose of the website—the pictures, ratings, reviews, and suggestions—is to let me find, understand, and compare their offerings. The experience is about building a decision inside my head. In the end, I manipulate a shopping cart, but that is merely to put my mental process to effect, to reify the decision. At the best retailers, this manipulation is made as brief as possible.

Even consider reading email. Most current designs revolve around the manipulation of individual messages—reading them one-by-one, searching them, sorting them, filing them, deleting them. But the *purpose* of reading email has nothing to do with the messages themselves. I read email to keep a complex set of mental understandings up-to-date—the statuses of personal conversations, of projects at work, of invitations and appointments and business transactions and packages in the mail. That this information happens to be parceled out in timestamped chunks of text is an implementation detail of the communication process. It is not necessarily a good way to *present* the information to a learner.



Similar arguments can be made for most software. Ignore the structure of current designs, and ask only, “Why is a person using this?” Abstracted, the answer almost always is, “To learn.”

So far, this categorization has just been an exercise in philosophy. But this philosophy suggests a very practical approach to software design.



## Information software design is graphic design

It might seem like I’m demanding a lot from my software. But it’s not deep magic—no simulations of complex phenomena, no effects on the external world, certainly no sentience or spark of life. I’m asking for software to display a complex set of data in a way that I can understand it and reason about it. This is a well-established problem; it’s the *raison d’être* of information graphic design. My demands are perfect examples of graphic design challenges.

A well-designed information graphic can almost *compel* the viewer to ask and answer questions, make comparisons, and draw conclusions. It does so by exploiting the capabilities of the human eye: instantaneous and effortless movement, high bandwidth and capacity for parallel processing, intrinsic pattern recognition and correlation, a macro/micro duality that can skim a whole page or focus on the tiniest detail. Meanwhile, a graphic sidesteps human shortcomings: the one-dimensional, uncontrollable auditory system, the relatively sluggish motor system, the mind’s limited capacity to comprehend hidden mechanisms. A graphic presents no mechanisms to comprehend or manipulate—it plugs directly into the mind’s spatial reasoning centers.

For example, consider this train timetable:

	Southbound	Northbound
Dictionopolis	6:00 7:05 7:15 8:40 9:12 11:38 2:24 3:52 6:00* 6:02 7:15**	
Forest of Sight	7:02 8:20 9:45 12:42 3:24 7:00* 8:15**	7:45 10:36 1:15 4:55 6:12 7:36
Valley of Sound	7:50 9:08 10:36 1:30 4:12 7:50*	6:55 9:48 12:30 4:10 5:25 6:48
Digitopolis	8:50 10:12 11:30 2:30 5:12 6:00 8:12 8:50 10:20	11:35 3:12 4:25 4:55 5:50 7:10
Mtns of Ignorance	11:00 12:15 1:40 4:36 7:24	9:24 1:00 2:15 3:40 6:38*
Castle in the Air		6:00 8:24 12:00 1:15 2:45 5:40*

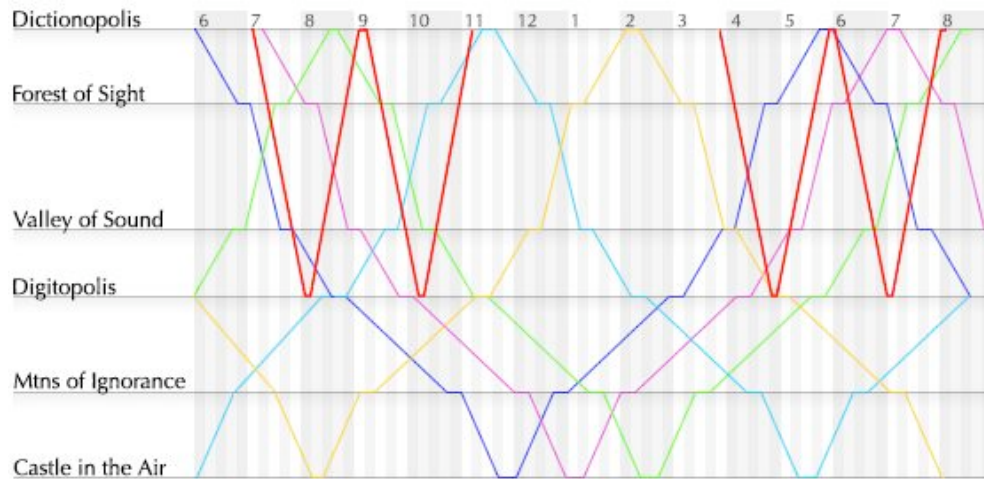
Red indicates an express train, which stops only at Dictionopolis and Digitopolis.

\* Line ends at Digitopolis.

\*\* Line ends at Valley of Sound.

This design may be adequate for commuters, whose questions mostly concern when trains arrive at stations. But train system operators have a different set of questions: Where exactly are the trains at any given time? How fast are they moving? Where do two trains cross? (They better not be on the same track at that point!) Where are the trains at the start of the day, and

where do they end up at night? If a train is delayed, how do all these answers change? Like some of the software questions above, these questions seem very difficult to answer. But consider this revised timetable design:



Each train is represented by a distinctly-colored line, with distance along the track plotted vertically and time horizontally. The slope of the line represents the train’s direction and speed; horizontal sections are stops. This graphic incorporates no more *data* than the previous one, yet all of the operators’ questions are answered at a glance. Important features such as crossings are *emphasized* simply because the eye is naturally drawn toward line intersections. Footnotes are unnecessary; the exceptions are no longer exceptional when seen in context. Should a train be delayed, all revised stops and crossings can be “calculated” simply by drawing a new line.\*

Compared to excellent ink-and-paper designs, most current software communicates deplorably. This is a problem of surface, but not a superficial problem. The main cause, I believe, is that many software designers feel they are designing a machine. Their foremost concern is behavior—what the software *does*. They start by asking: What functions must the software perform? What commands must it accept? What parameters can be adjusted? (In the case of websites: What pages must there be? How are they linked together? What are the dynamic features?) These designers start by specifying *functionality*, but the essence of information software is the *presentation*.

**I suggest that the design of information software should be approached initially and primarily as a graphic design project.** The foremost concern should be appearance—what and how information is presented. The designer should ask: What is relevant information? What questions will the viewer ask? What situations will she want to compare? What decision is she trying to make? How can the data be presented most effectively? How can the visual vocabulary and techniques of graphic design be employed to direct the user’s eyes to the solution? The designer must start by considering what the software *looks like*, because the user is using it to learn, and she learns by looking at it.

\* Graphical train timetables date from the late 1800s. For the origin of this and other classic graphical forms, see Howard Wainer’s book [Graphic Discovery](#) (2005).

It must be mentioned that there is a radically alternative approach for information software—*games*. Playing is essentially learning *through* structured manipulation—exploration and practice instead of pedagogic presentation. Despite the enormous potential for mainstream software, accidents of history and fashion have relegated games to the entertainment bin, and the stigma of immaturity is tough to overcome. (The situation is similar for graphic novels.) Raph Koster’s [Theory of Fun for Game Design](#) (2004) and

Instead of dismissing ink-and-paper design as a relic of a previous century, the software designer should consider it a *baseline*. If information software can't present its data at least as well as a piece of paper, how have we progressed?

James Paul Gee's [What Video Games Have To Teach Us About Learning and Literacy](#) (2003) deal directly with games as learning tools. Salen and Zimmerman's [Rules of Play](#) (2003) and Chris Crawford's [Art of Interactive Design](#) (2003) and [Chris Crawford on Game Design](#) (2003) discuss learning through play in a broader context.

## Demonstration: Showing the data

Edward Tufte's first rule of statistical graphic design is, "*Show the data.*" All information graphics, statistical or not, must present the viewer with enough information to answer her questions. It seems that many software designers, in their focus on functionality, forget to actually present the data.

Consider the information presented when searching a popular online bookstore.\*

\* Based on [amazon.com](#) as of January 2006.

### Showing results in: "bread"



**Waiting For Good Dough** by Samuel Biscuit (**Hardcover** - Jul 3, 1953)

Books: [See all 3764 items](#)

**Buy new:** \$40.00 **Used & new** from \$23.86 Usually ships in 24 hours

**Excerpt from page 2:** "... unless your **bread** machine uses pneumatic pistons or is ..."

[See more references](#) to **bread** in this book.

**Surprise me!** [See a random page](#) in this book.



**The Bun Also Rises** by Ernest Hemingwaffle (**Hardcover** - Jun 3, 1926)

Books: [See all 3764 items](#)

**Buy new:** \$21.80 **Used & new** from \$9.86 Usually ships in 24 hours



**Yeast of Eden** by John Sconebeck (**Paperback** - Sep 3, 1952)

Books: [See all 3764 items](#)

**Buy new:** \$20.00 **\$16.50** **Used & new** from \$6.00 Usually ships in 24 hours

**Excerpt from page 1:** "... wards off vampires. But **bread** should never be ..."

[See more references](#) to **bread** in this book.

**Surprise me!** [See a random page](#) in this book.

There are a number of graphic design criticisms one could make—the uniform text size and weight results in a solid, oppressive mass; the abundance of saturated primary colors gives a distracting, carnival-like appearance; the text is spread all over the page, giving the eye no well-defined path to follow. However, the most egregious problem is simply that there is *not enough information to make any sort of decision*.

The user's goal is to find the best book about some particular topic. Given that the books shown are presumably related to this topic, what questions does the user have?

- Is the book *appropriate*? That is, what is it about, and do I care?
- Is the book *good*? That is, what did other people think of it, and do I trust them?

The answers will be used to compare the available books, and decide upon one to follow up on and possibly buy.

Unfortunately, these questions are completely unaddressed by the information provided. To see relevant information, the user must click on each listing individually. That is, she must navigate by *hand* instead of by *eye*, and must use her memory to compare information across *time* instead of *space*.

The problem is that this graphic was designed as an *index* into a set of webpages, but is used as a *catalog* for comparing a set of books. The purpose of this graphic should not be to return a list of query matches, but to help the user *learn* about books related to her topic of interest.

Consider this redesign:



	<p><b>Waiting for Good Dough</b> by <b>Samuel Biscuit</b>. 180 pages, hardcover</p> <p><b>Bread machine cookbook.</b> An introductory guide to setting up and using a bread machine, with 80 recipes from basic to exotic. The author, a renowned breadmaster at New York's Biscuit Bakery, guides the novice baker step-by-step, pointing out pitfalls and opportunities for improvisation.</p> <p><b>I Love My New Bread Machine</b>, "just what I needed to get going" <b>Well-written but Simplistic</b>, "doesn't cover many of the fancier breads" <b>Every Bread Machine Should Come With This</b>, "the foccaccia is delicious"</p> <p>★★★★☆ New: <b>\$40.00</b>, 7 used &amp; new from <b>\$23.86</b></p>	<p><b>CONTENTS:</b></p> <table border="0"> <tr> <td>Bread basics</td> <td>Whole wheat breads</td> </tr> <tr> <td>Choosing a bread machine</td> <td>Rye breads</td> </tr> <tr> <td>Getting to know your machine</td> <td>Multigrain breads</td> </tr> <tr> <td>Your first loaf</td> <td>Country breads</td> </tr> <tr> <td>Variations on a theme</td> <td>Sourdough breads</td> </tr> <tr> <td>Tips and tricks</td> <td>And for dessert...</td> </tr> <tr> <td>White breads</td> <td>Appendix: Crumbs and croutons</td> </tr> <tr> <td>Egg breads</td> <td>Index</td> </tr> </table> <p><b>RELATED BOOKS:</b></p> <p>★★★★☆ <b>The Bread Machine Book</b> by Amanda Crumbles      ★★★★★ <b>The Bread Batch of Courage</b> by Stephen Crepe      ★★★★★ <b>Knead for Speed</b> by Pam Cake</p>	Bread basics	Whole wheat breads	Choosing a bread machine	Rye breads	Getting to know your machine	Multigrain breads	Your first loaf	Country breads	Variations on a theme	Sourdough breads	Tips and tricks	And for dessert...	White breads	Appendix: Crumbs and croutons	Egg breads	Index
Bread basics	Whole wheat breads																	
Choosing a bread machine	Rye breads																	
Getting to know your machine	Multigrain breads																	
Your first loaf	Country breads																	
Variations on a theme	Sourdough breads																	
Tips and tricks	And for dessert...																	
White breads	Appendix: Crumbs and croutons																	
Egg breads	Index																	
	<p><b>The Bun Also Rises</b> by <b>Ernest Hemingwaffle</b>. 570 pages, hardcover</p> <p><b>Pastry recipes.</b> Over 650 recipes make up this classic reference, beloved by pastry chefs and home cooks throughout the world. The only cookbook endorsed by the National Pastry Alliance, this guide is filled with mouthwatering ideas for restaurant service or entertaining at home.</p> <p><b>Inside Tips from a Pro</b>, "an excellent reference for a serious home chef" <b>Belongs in Everyone's Collection</b>, "I find myself referring to it again and again" <b>Superb and Authoritative</b>, "an encyclopedia of pastry-making"</p> <p>★★★★☆ New: <b>\$21.80</b>, 14 used &amp; new from <b>\$9.86</b></p>	<p><b>CONTENTS:</b></p> <table border="0"> <tr> <td>Welcome to my kitchen!</td> <td>Tarts</td> </tr> <tr> <td>Basic doughs</td> <td>Pies and cobblers</td> </tr> <tr> <td>Yeast breads</td> <td>Pound cakes</td> </tr> <tr> <td>Flatbreads</td> <td>Sponge cakes</td> </tr> <tr> <td>Breakfast breads</td> <td>Custards and puddings</td> </tr> <tr> <td>Croissants and turnovers</td> <td>Sauces and syrups</td> </tr> <tr> <td>Buns and muffins</td> <td>Index</td> </tr> <tr> <td>Cookies</td> <td></td> </tr> </table> <p><b>RELATED BOOKS:</b></p> <p>★★★★☆ <b>Lord of the Pies</b> by William Goldenbrown      ★★★★★ <b>Quit Playing Games With My Tart</b> by Sally Lunn      ★★★★★ <b>Hedda Cobbler</b> by Henrik Ibscane</p>	Welcome to my kitchen!	Tarts	Basic doughs	Pies and cobblers	Yeast breads	Pound cakes	Flatbreads	Sponge cakes	Breakfast breads	Custards and puddings	Croissants and turnovers	Sauces and syrups	Buns and muffins	Index	Cookies	
Welcome to my kitchen!	Tarts																	
Basic doughs	Pies and cobblers																	
Yeast breads	Pound cakes																	
Flatbreads	Sponge cakes																	
Breakfast breads	Custards and puddings																	
Croissants and turnovers	Sauces and syrups																	
Buns and muffins	Index																	
Cookies																		
	<p><b>Yeast of Eden</b> by <b>John Sconebeck</b>. 270 pages, paperback</p> <p><b>History of bread.</b> This well-researched and beautifully-illustrated book covers bread's 6000-year history and role in world cultures. The author, a professor at the American Culinary Academy, vividly describes bread's surprising and pivotal roles in politics, religion, and technology, from ancient Egypt to today.</p> <p><b>Thorough but Dry</b>, "I was struggling through the dry, academic writing style" <b>Dull, Dull, Dull</b>, "easily the worst book I've read on the history of bread" <b>Comprehensive Account of the Rise of Bread</b>, "I found it fascinating"</p> <p>★★★★☆ New: <b>\$16.50</b>, 2 used &amp; new from <b>\$6.00</b></p>	<p><b>CONTENTS:</b></p> <table border="0"> <tr> <td>The Pharaohs</td> <td>McCormick's revolution</td> </tr> <tr> <td>Ancient Isreal</td> <td>America's wheat empire</td> </tr> <tr> <td>Ancient Greece</td> <td>World War I</td> </tr> <tr> <td>Ancient Rome</td> <td>The modern farmer</td> </tr> <tr> <td>The Middle Ages</td> <td>The bread of the future</td> </tr> <tr> <td>The early Americas</td> <td>Bibliography</td> </tr> <tr> <td>The French Revolution</td> <td>Index</td> </tr> <tr> <td>The rise of science</td> <td></td> </tr> </table> <p><b>RELATED BOOKS:</b></p> <p>★★★★☆ <b>One Hundred Years of Solid Food</b> by G. Gordita Marquez      ★★★★★ <b>Decline and Fall of the Ramen Empire</b> by Yoshi Noya      ★★★★★ <b>Gums, Germs, and Strudel</b> by Jared Diner</p>	The Pharaohs	McCormick's revolution	Ancient Isreal	America's wheat empire	Ancient Greece	World War I	Ancient Rome	The modern farmer	The Middle Ages	The bread of the future	The early Americas	Bibliography	The French Revolution	Index	The rise of science	
The Pharaohs	McCormick's revolution																	
Ancient Isreal	America's wheat empire																	
Ancient Greece	World War I																	
Ancient Rome	The modern farmer																	
The Middle Ages	The bread of the future																	
The early Americas	Bibliography																	
The French Revolution	Index																	
The rise of science																		

Is a book appropriate? A synopsis and table of contents give an overview of the book's contents.

Is a book good? A rating and reviews indicate popular opinion. Because all of this information is on a single page, it can be compared by eye, with no need for memory.

The standard 5-star rating system is information-weak—it gives only an *average*. It can be enhanced with whiskers underneath that indicate the *distribution* of ratings. This allows the viewer to differentiate between a book that was unanimously judged middling ★★★★★ and



one that was loved and hated  —these are both 3-star ratings, but have very different meanings. The viewer can also see whether a highly-rated book got *any* bad reviews; in a sea of praise, criticism often makes enlightening reading. As a whole, the whiskers give a visual indication of the number of ratings, which reflects the trustworthiness of the average. The whiskers are unobtrusive, and can easily be ignored by viewers who don't care about distribution.




Text weight and color is used to emphasize important information and call it out when skimming. Text in grey can be read when focused upon, but disappears as background texture when skimming. All critical information is contained in a column with the **width of an eyespan**, with a picture to the left and supplementary information to the right. The viewer can thus run her eye vertically down this column; when she spots something interesting, she will slow down and explore horizontally.

The user wants to see books related to a topic in her *head*. But ideas in the head are nebulous things, and may not translate perfectly to a concrete search term. For this reason, a mini-list of related books is provided for each book. This is similar to a “related words” section in a thesaurus listing—it allows the user to correct a near miss, or veer off in a tangential but intriguing direction.

Conventional software designers will worry about functionality—how does the user interact with this graphic? Clearly, other than the “related books” listing, a click *anywhere* in a book's section should reveal details and purchasing options. What else could the user mean by clicking? It's analogous to pulling the book off a physical shelf.

This is a significant redesign over the original; yet, I consider it a conservative one. A more ambitious design could surely show even *more* data, perhaps allowing the user to browse within the book or fully explore the space of related books. A world of possibilities opens up with a simple change of mindset. This is not a list of search results—it is an information graphic. It is for *learning*.



RELATED BOOKS:  
 One Hundred Years of Solid Food  
 Decline and Fall of the Ramen Era  
 Gums, Germs, and Strudel by Jan

## Demonstration: Arranging the data

Just as important as *what* data is shown is *where* it is shown. Unlike the words in a paragraph, the elements in a graphic can be deliberately *placed* to encourage spatial reasoning. Unfortunately, most software graphics are arranged to maximize aesthetics, not to bring out useful relationships in the data. (That is, when any skilled thought is given to appearance at all.)

Consider this excerpt of a graphic for browsing nearby movie showings:\*

\* Based on [movies.yahoo.com](http://movies.yahoo.com) as of January 2006.

Old Joe's Showhouse [ <a href="#">Add to My Favorite Theaters</a> ]	
17 Main St., Lilliput <a href="#">Theater Info</a>   <a href="#">Map It</a>	
<a href="#">Grapes of Khan, The</a> Rated PG-13, 1 hr 22 min <b>Showtimes:</b> 11:00, 4:15, 5:20, 7:45, 9:55, 10:10	<a href="#">Rainman Forever</a> Rated R, 1 hr 33 min <b>Showtimes:</b> 11:55, 3:00, 7:15
<a href="#">Rent and Rentability</a> Rated PG-13, 1 hr 43 min <b>Showtimes:</b> 1:15, 5:15, 9:30	

Little-End Cinemas [ <a href="#">Add to My Favorite Theaters</a> ]	
47 Main St., Lilliput <a href="#">Theater Info</a>   <a href="#">Map It</a>	
<a href="#">Die Hard With More Intensity</a> Rated R, 1 hr 47 min <b>Showtimes:</b> 11:55, 1:15, 2:30, 3:50, 5:05, 6:25, 7:40, 9:05 10:15	<a href="#">Hairy Plumber and the Goomba of Doom</a> Rated PG, 2 hr 10 min <b>Showtimes:</b> 11:30, 1:35, 3:40, 5:45, 7:50, 10:00
<a href="#">Rainman Forever</a> Rated R, 1 hr 33 min <b>Showtimes:</b> 2:15, 4:45, 7:15, 9:35	<a href="#">Rent and Rentability</a> Rated PG, 1 hr 43 min <b>Showtimes:</b> 7:00, 9:30, 11:30

If a person is in the mood for a movie, what questions might she have?

- What movies are showing today, at which times?
- What movies are showing around a *particular* time?
- Where are they showing?
- What are they about?
- Are they good?

The user will use the answers to compare the available movie showings and decide upon one to go see.

Although the above graphic clearly has an information deficiency (What are these movies about? Are they good?), the worst problem is that the data is not arranged in any useful manner. Understanding which movies are playing when involves scanning a pageful of theaters, extracting movies of interest and mentally merging their showtimes. A viewer's eye might leap erratically around the screen as she compares showtimes of a given movie at six theaters, trying to find the one that best fits her dinner plans.

The primary question is, "What *movies* are showing today, at which *times*?" Given the two spatial dimensions available to us, this should suggest a graphic with *movies* along one axis and *times* along the other.

Consider this redesign:



### ★★★★★ Rainman Forever

**Action:** An autistic man fights crime on the streets of Gotham.  
Uwe Boll, Dustin Hoffman, Jim Carrey, Jet Li  
"137 interminable minutes. I counted them." (Ebert)  
"He's an excellent driver in a terrible movie." (Boston Sun)  
"A flop. Definitely a flop.." (SF Chronicle)



### ★★★★★ Die Hard With More Intensity

**Drama:** A streetwise cop confronts loneliness in Tokyo.  
Sofia Coppola, Bill Murray, Bruce Willis, Jet Li  
"Extraordinarily powerful ... A masterpiece of cinema." (Ebert)  
"Beautiful and haunting." (filmscritic.com)  
"Moves slow but packs a punch." (Boston Sun)



### ★★★★★ Hairy Plumber and the Goomba of Doom

**Fantasy:** Mario and Luigi attend a school of wizardry.  
Steven Spielberg, Daniel Radcliffe, Emma Watson, Jet Li  
"Not as good as the others, but still a visual treat." (Ebert)  
"The boys are back and better than ever." (filmscritic.com)  
"Fans of the series won't be disappointed." (Boston Sun)



### ★★★★★ Rent and Rentability

**Romance:** Dissimilar sisters seek husbands in the East Villiage.  
Ang Lee, Emma Thompson, Kate Winslet, Jet Li  
"A poor adaptation of the Broadway hit." (Ebert)  
"A real tear-jerker. Keep your hanky handy!" (filmscritic.com)



### ★★★★★ The Little Schemer

**Adventure:** An elephant journeys to find Lambda the Ultimate.  
Friedman & Felleisen, Car, Cdr, Cons, Cond  
"Cons is magnificent! ... Add this movie to your list!" (Ebert)

Old Joe's Showhouse	11:55		3:00			7:15				
AMC Lilliput		1:25	2:45	4:20	6:15	7:30	9:25	10:35		
UA Easy Street		12:45	3:00	5:00	7:20	9:00	11:00			
Gulliver Theater		1:40		4:25		7:20		10:00		
Little-End Cinemas		2:15	4:45	7:15	9:35					
AMC Lilliput	11:45	12:40	2:00	3:30	5:00	6:30	7:30	8:30	9:50	11:30
UA Easy Street		1:00	2:30	4:30	7:00	8:50	10:00			
Landmark Quinbus				4:30		8:00	10:00	11:50		
Little-End Cinemas	11:55	1:15	2:30	3:50	5:05	6:25	7:40	9:05	10:15	
AMC Lilliput	12:00		2:25	4:45	7:05	9:25				
UA Easy Street	12:40		3:55		7:15	10:20	12:15			
Landmark Quinbus	12:05		2:45							
Gulliver Theater						8:30	10:30			
Little-End Cinemas	11:30	1:35	3:40	5:45	7:50	10:00				
Old Joe's Showhouse		1:15		5:15		9:30				
Gulliver Theater		12:35	2:20	4:55		8:20	11:10			
Little-End Cinemas					7:00	9:30	11:30			
PLT Arthouse					7:30	9:30				

As with the bookstore redesign, enough information is given about each movie to determine its content and quality, although films have enough external marketing that the intent is more to *remind* than *introduce*. Text weight is again employed to make critical information stand out and supplementary information disappear until focused upon.

More interesting is the chart on the right, which plots movie showings by time. To find all movie showings around a particular time, the viewer simply scans her eye vertically down the page. If she is only interested in a particular movie, she looks only within that movie's range. The current time is indicated by shading the past, providing a springboard for the viewer's eye; in this example, it is about 4:45.

The original design grouped movies by theater; this redesign groups theaters by movie.\* The assumption is that the viewer would rather see a particular movie at any theater than any movie at a particular theater. However, to ease correlation of the various movies offered at a given theater, each theater is color-coded. If the viewer prefers to avoid the Gulliver Theater because of sticky floors, the consistent yellow background may help her filter out its showtimes.

No theater addresses are shown. The viewer is likely to be familiar with the theaters in her area, and if she isn't, a simple address would be useless without a map or directions. Presumably, a mouse click or hover over a theater's name would reveal this information, or perhaps it could be displayed elsewhere on the page.

This demonstration and the previous one have attempted to illustrate the power of approaching information software as *graphic design*, instead of as styling the regurgitation of a database. To design excellent software, however, this mindset is necessary but insufficient. Something major is missing.

\* I assume that Yahoo! simply mimicked the newspapers, and newspapers arrange by theater for business reasons.

Very little in the above designs is software-specific. For the most part, the designs would work almost as well on *paper*. Modern magic shouldn't merely match our ancient technology—it should surpass it. We've seen how graphic design can improve software, but *how can software improve graphic design?*

The answer lies with *context*.



## Context-sensitive information graphics

Print has one supreme flaw: ink is indelible. An ink-and-paper design is static—it must display all its data, all the time. However, a reader typically only *cares* about a subset relevant to her current situation. The designer is faced with the challenge of organizing the data so that hopefully mutually-relevant subsets are grouped together, and the reader has the challenge of visually or physically navigating through the entire data space to find the group of interest.

For example, a rider consulting a bus schedule must comb through a matrix of times and stations to find the single relevant data point—the time of the next bus.\* Any driver who's been lost in an unfamiliar city knows the frustration of locating the immediate vicinity on a roadmap. And a reader consulting an encyclopedia must not only find the right entry on the page and the right page in the book, but even the right book on the shelf! These are consequences of static graphics. Because ink is permanent, the reader must navigate through lots of paper.

\* And then, she must consult her watch and do some arithmetic to calculate the information she actually cares about—how long she will be waiting.

The modern computer system provides the first visual medium in history to overcome this restriction. Software can:

- **infer the context** in which its data is needed,
- **winnow the data** to exclude the irrelevant, and
- **generate a graphic** which directly addresses the present needs.

Liberating us from the permanence of publication is the undersung crux of the computer—the dynamic display screen. Its pixels are magic ink—capable of absorbing their context and reflecting a unique story for every reader. And the components surrounding the display—CPU, storage, network, input devices—are its *peripherals for inferring context*.

Information software design, then, is the design of *context-sensitive information graphics*. Unlike conventional graphics, which must be suitable for any reader in any situation, a context-sensitive graphic incorporates *who* the user is and *what* exactly the user wants to



learn at the moment. Context allows software to winnow its data space to the subset of information that the user cares about, and present the data in such a way that the user's current questions can best be answered.

All information software consists of context-sensitive graphics, whether the designer realizes it or not. For example, the list of query results from an internet search engine is a context-sensitive information graphic. The software's data space consists of all the websites in the world. This is winnowed down to a dozen, using context that is inferred entirely from the search term contributed by the user.\* Despite its enormous data space, this software restricts itself to a meager scrap of context, impersonal and imprecise.

\* Clicking a "next" button contributes further context—dissatisfaction with the first set of results.

There are, in fact, three sources from which software can infer context:

- **Environment** involves sensing the current state of the world.
- **History** involves remembering the past.
- **Interaction** involves soliciting input from the user.

## Inferring context from the environment

A person determines her surroundings through the five human senses. Software doesn't operate in a vacuum, either; through connections to hardware and other software, it can sense much about the user's situation. Some examples of context clues in the software's environment include:

- **Date and time.** Time is one of fundamental dimensions along which we organize our lives, and in any data space with a temporal dimension, "now" is almost always the prime landmark. Because users often seek information on demand, information related to "now" or "soon" is often the most relevant. Fortunately, every general-purpose computer knows when "now" is. A person using a software bus schedule, for example, should never have to hunt for the next bus.
- **Geographical location.** Similarly, the most interesting spatial landmark is usually "here." Unfortunately, this currently can be harder to determine automatically, but the payoff is enormous.\* Obviously, a software roadmap needs to know the user's location, but so does the bus schedule, as well as business listings, transportation planners, travel guides, and much other information software.
- **Physical environment.** Given a time and location, many details of the physical environment, such as the weather, are just a network connection away. Consider a travel guide that suggests parks when sunny and museums when rainy.
- **Other information software,** such as open websites. By reading some information, the user is indicating a topic of interest. All other information software should take heed. Consider a person reading the website of an upcoming stage play. When she opens her calendar, the available showings should be marked. When she opens a map, she should see directions to the playhouse. When she opens a restaurant guide, she

\* I believe that location is such vital context, Powerbooks should come with GPS receivers pre-installed, with an easy software API. Developers would then write software to take advantage of it, and other computer makers would follow suit. Someday, a computer without GPS might seem as silly as a computer without a clock.

should see listings nearby, and unless the play offers matinees, they shouldn't be lunch joints.

- **Documents created with manipulation software.** *Creating* some information indicates an even stronger topic of interest. Consider a person who requests information about “cats” while writing a paper. If the paper’s title is “Types and Treatment of Animal Cancer,” the information should skew toward feline medical data. The title “History of Egypt” indicates interest in ancient feline worship instead. And if the paper contains terms related to building construction, “cats” probably refers to the decidedly non-feline Caterpillar heavy machinery.\*
- **Email.** Names, addresses, and phone numbers in recent email clearly constitute valuable hints. A recipient who opens a calendar should find the sender’s schedule juxtaposed with her own. When she opens a map, addresses in the email should be marked. But beyond that, recent correspondence can indicate current activities, and an email archive as a whole can describe the user’s characteristics and interests. Consider a person who requests information about “racing.” The fields of running, bicycles, and cars have distinct sets of terminology; if one set regularly shows up in the person’s conversations, “racing” isn’t so ambiguous.

All software lives within an environment, rich with evidence of context. Using software that doesn’t look outside itself is like conversing with a blind person—constantly describing what is plainly visible.\*

On the other hand, the power of the environment is multiplied when it is correlated with the past—that is, when the software makes use of *history*.

## Inferring context from history

A human doesn’t just use her senses to recognize her situation; she also uses memories of past situations. Software, too, can use its memory to understand the present. The current context, or a good approximation, can often be *predicted* from a history of past environments and interactions.

**Last-value predictors** represent the simplest form of prediction. They simply predict the current context to be the same as the previous one. This is reasonable in many situations where the user’s context is fairly static, changing slowly over the short term. For example, if yesterday, the user looked for one-bedroom apartments in North Berkeley, she is probably still interested in one-bedroom apartments in North Berkeley today. If nothing else, the software should present this information immediately, without asking for details.

Last-value prediction is frequently thought of and implemented as manipulation of explicit state—that is, the context is a persistent object that remains as is unless changed by the user, so the software always appears as the user left it.\* This stateful conceptual model mimics physical reality, and can be comfortable if the user cares enough about the software’s state to

\* This example is from Budzik and Hammond’s paper [User Interactions with Everyday Applications as Context for Just-in-time Information Access](#) (2000).

\* Some of the suggestions given here may seem daunting (or infeasible) to an engineer. Implementation will be discussed later in the paper.

\* The engineering challenge then becomes merely persisting across invocations of the program. Often, not even this is bothered with.

keep her own mental state in sync. However, this is often not the case with information software, especially software that is consulted intermittently. (If you put down a newspaper for a few hours, you won't be distressed to find it on a different page when you return. You probably won't even notice. On the other hand, you would be delighted if you often came back to find it on exactly the page you wanted to read.) By thinking of this as context prediction instead of state maintenance, the door is opened to more sophisticated predictors.

**Learning predictors** attempt a deeper understanding of the user. They construct a model to explain past contexts, and use the inferred relationships to predict the current context.

One simple approach to learning is to discover a common attribute of recent contexts, and narrow the current context along that attribute's dimension. For example, in a music library, as the user chooses several bluegrass songs in a row, the software can graphically emphasize other songs in this genre. With further confidence, it might consider de-emphasizing or omitting songs outside of the genre. As another example, consider a user who requests information about "Lightwave," then about "Bryce," then "Blender." These terms have many meanings individually, but as a group they are clearly names of 3D rendering software packages. A subsequent search for "Maya," another 3D package, should not display information about the ancient civilization. In fact, information about Maya could be presented automatically.

Another simple approach is to establish the user's velocity through the data space. If a person asks a travel guide about the Grand Canyon on one day, and Las Vegas the next day, the following day the software might suggest attractions around Los Angeles.\*

\* A better travel guide would suggest skipping Los Angeles.

In general, the problem is one of inferring a *pattern* that explains the user's interests as a function of the environment, and extrapolating along the pattern to classify the current environment. As an example of general pattern modeling, consider a person who, as a byproduct of traveling to work, always checks the train schedule from Berkeley to San Francisco in the morning, and San Francisco to Berkeley in the evening. If the software can discover and model this pattern, it can present the appropriate information at each time without the user having to request it. When she looks in the morning, she sees by default the San Francisco-bound schedule; in the evening, the Berkeley-bound schedule.\*

\* Again, this may sound daunting to an engineer. Implementation will be discussed later in the paper.

Large histories can enable some very sophisticated predictors, especially if it is possible to reach into the environment and correlate with other users' histories. For example, by asking their users to rate each movie they return, Netflix is able to infer some enormously valuable context—each user's *taste*. This allows them to winnow an enormous dataset (their catalog of movies) down to a dozen data points (movies the user hasn't seen, which were enjoyed by people with similar taste), which can be presented in a single, navigation-free graphic. The winnowing is impressively on-target—two-thirds of users' selections come from recommendations.\* TiVo similarly uses a collaborative predictor to infer which television programs the user would be interested in. These are presented on a "suggestions" page, and recorded automatically when possible.\*

\* Laurie J. Flynn, [Like This? You'll Hate That](#). *New York Times*, Jan. 23, 2006.

\* For technical details, see Ali and van Stam's paper [TiVo: Making Show Recommendations Using a](#)

Amazon, iTunes, and an increasing number of other online retailers are currently incorporating similar schemes. However, with the exception of the lowly junk-mail filter, non-retail information software that learns from history is still rare. Typically, users can only hope for last-value prediction, if that. Most software wakes up each day with a fresh case of amnesia.

Unfortunately, software that doesn't learn from history dooms users to repeat it. And repeat it they will—tediously explaining their context, mouse click by mouse click, keystroke by keystroke, wasted hour by wasted hour. This is called *interactivity*.



## Interactivity considered harmful

Chris Crawford defines interaction as a three-phase reciprocal process, isomorphic to a conversation: an interactant **listens** to her partner, **thinks** about what was said, and **speaks** a response. Her partner then does the same.\* For *manipulation* software, interaction is perfectly suitable: the user views a visual representation of the model, considers what to manipulate next, and performs a manipulation. The software, in turn, inputs the user's manipulation request, updates the model, and displays the updated representation. With good feedback and an effective means of "speaking" to the software, this process can cycle smoothly and rapidly. It mimics the experience of working with a physical tool.

Information software, by contrast, mimics the experience of *reading*, not working. It is used for achieving an understanding—constructing a model within the mind. Thus, the user must *listen* to the software and *think* about what it says... but any manipulation happens mentally.\* The only reason to complete the full interaction cycle and *speak* is to explicitly provide some context that the software can't otherwise infer—that is, to indicate a relevant subset of information. **For information software, all interaction is essentially navigation around a data space.**

For example, Amazon's data space consists of their catalog of items. For a yellow pages directory, the data space contains all business listings; for a movie guide, all showtimes and movie information; for a flight planner, trips to and from all airports. In all of these cases, every interaction, every click and keystroke, search term and menu selection, simply serves to adjust the user's view into the data space. This is simply *navigation*.

Alan Cooper defines *excise* in this context as a cognitive or physical penalty for using a tool—effort demanded by the tool that is not directly in pursuit of a goal. For example, filling a gas

\* See Crawford's book [The Art of Interactive Design](#) (2003), or his essay [Fundamentals of Interactivity](#) (1993).

\* Except possibly for signaling a decision, such as clicking a "buy" button, but that *concludes*, not *constitutes*, a session.

tank is done to support the car, not the goal of arriving at a destination. Cooper goes on to assert that software navigation is nothing but excise:

*...the most important thing to realize about navigation is that, in almost all cases, it represents pure excise, or something close to it. Except in games where the goal is to navigate successfully through a maze of obstacles, navigation through software does not meet user goals, needs, or desires. Unnecessary or difficult navigation thus becomes a major frustration to users. In fact, it is the authors' opinion that poorly designed navigation presents the **number-one problem** in the design of any software application or system...\**

\* Alan Cooper and Robert Reimann, [About Face](#) (2003), p143.

If all interaction is navigation, and navigation is the number-one software problem, interactivity is looking pretty bad already. However, when compared with the other two sources of context, interactivity has even *worse* problems than simply being a frustrating waste of time:

- The user has to already know *what* she wants in order to ask for it. Software that infers from history and the environment can proactively offer potentially relevant information that the user wouldn't otherwise know to ask for. Purely interactive software forces the user to make the first move.
- The user has to know *how* to ask. That is, she must learn to manipulate a machine. Donald Norman's concept of determining a user's "mental model" has become widespread in the software usability community, and is now considered a core design challenge.\* However, Norman described this concept in the context of *mechanical* devices. It only applies to software if the software actually contains hidden mechanisms that the user must model. A low-interaction, non-mechanical information graphic relieves both user and designer from struggling with mental models.
- Navigation implies state. Software that can be navigated is software in which the user can get lost. The more navigation, the more corners to get stuck in. The more manipulable state, the more ways to wander into a "bad mode." State is the primary reason people fear computers—stateful things can be *broken*.\*

\* See Donald Norman's book [The Design of Everyday Things](#) (2002), p9.

Beyond these cognitive problems are *physical* disadvantages of interaction. The hand is much slower than the eye. Licklider described spending hours plotting graphs and seconds understanding them. A user who must manually request information is in a similar situation—given the mismatch between mousing and reading speeds, most of her time may be spent navigating, not learning. Further, the user might prefer to learn information while using her hands for other purposes, such as writing or eating or stroking a cat. Each time software demands the user's hands, this activity must be interrupted. Finally, the growing prevalence of computer-related repetitive stress injuries suggests that indiscriminate interactivity may be considerably harmful in a literal, physical sense.

\* The only state kept by a book is which page it is open to, which is why "getting lost in a book" describes a *pleasurable* experience!

Unless it is enjoyable or educational in and of itself, interaction is an essentially *negative* aspect of information software. There is a net positive benefit if it significantly expands the range of questions the user can ask, or improves the ease of locating answers, but there may be other roads to that benefit. As suggested by the above redesigns of the train timetable, bookstore, and movie listings, many questions can be answered simply through clever, information-rich graphic design. Interaction should be used *judiciously* and *sparingly*, only when the environment and history provide insufficient context to construct an acceptable graphic.

It is unfortunate that the communities concerned with human factors of electronic artifacts have latched onto the term “interaction.”\* **For information software, the real issue is context-sensitivity.** Interaction is merely one means of achieving that. And as long as “speaking” is constrained to awkwardly pushing metaphors with a mouse, interaction should be the last resort.

The working designer might protest that interaction is unavoidable in practice, and may even consider my ideal of interaction-free software to be a scoff-worthy fantasy. This is only because the alternatives have been unrecognized and underdeveloped. I believe that with the invention of new context-sensitive graphical forms and research into obtaining and using environment and history, the clicking and dragging that characterizes modern information retrieval will be made to seem laughably archaic. But every condonation of “interactivity,” from the annals of academia to the corporate buzzvocabulary, postpones this future.

\* Most professional communities and academic programs use the term Human-Computer Interaction, or HCI; the ACM special-interest group is CHI, the converse. Many practitioners, following Cooper, refer to their profession as “interaction design.”

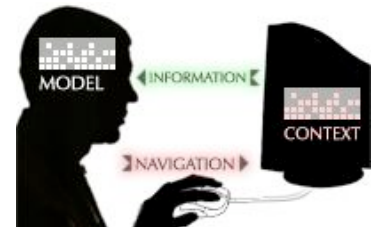
## Reducing interaction

When the user is forced to interact, the software assumes the form of manipulation software. The external model, manipulated through navigation, is the software’s model of the context. However, unlike genuine manipulation software, the user does not *care* about this model—it is merely a means to the end of seeing relevant information.

The designer’s goal is to let the user adequately shape the context model with as little manipulation as possible. Assuming that graphic design, history, and the environment have been taken as far as they will go, there are a few techniques that can lessen the impact of the remaining interaction:

- **Graphical manipulation** domains present the context model in an appropriate, informative setting.
- **Relative navigation** lets the user *correct* the model, not *construct* it.
- **Tight feedback loops** let the user stop manipulating when she’s close enough.

**Graphical manipulation.** Command-line systems are criticized for forcing the user to learn the computer’s language. Modern GUIs may be easier to use, but they are not much different in that respect. The GUI language consists of a grammar of menus, buttons, and checkboxes, each labeled with a vocabulary of generally decontextualized short phrases. The user “speaks”



by selecting from a tiny, discrete vocabulary within an entirely fixed grammatical structure—a bizarre pidgin unlike any human language, unexpressive and unnatural.\*

\* One might wonder what Sapir and Whorf would conclude.

As an alternative, consider a child describing his toy at “Show and Tell”:\*

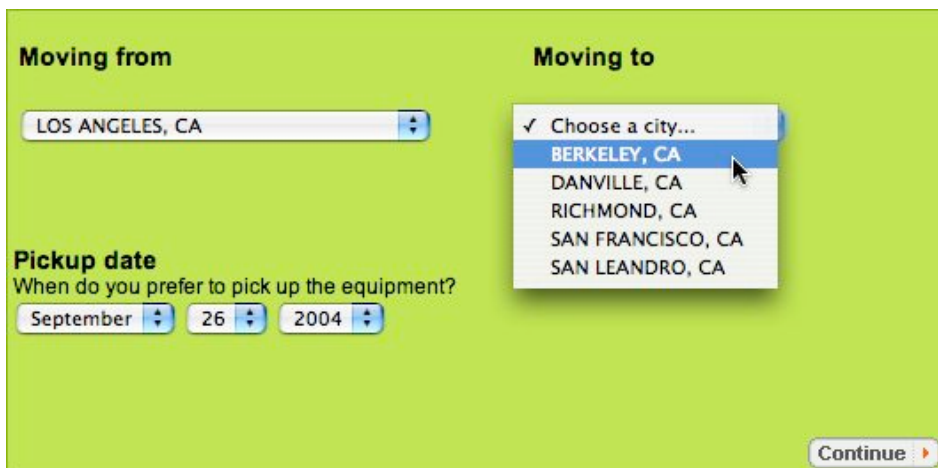
\* From Scott McCloud’s book Understanding Comics (1994), p138.



Because the child’s “telling” skills are underdeveloped, he communicates complex concepts through *showing*. Similarly, a GUT’s stunted grammar makes telling tedious, but software’s dynamic display is ideal for showing. A user can specify context by pointing somewhere on an information graphic and saying, “There!”

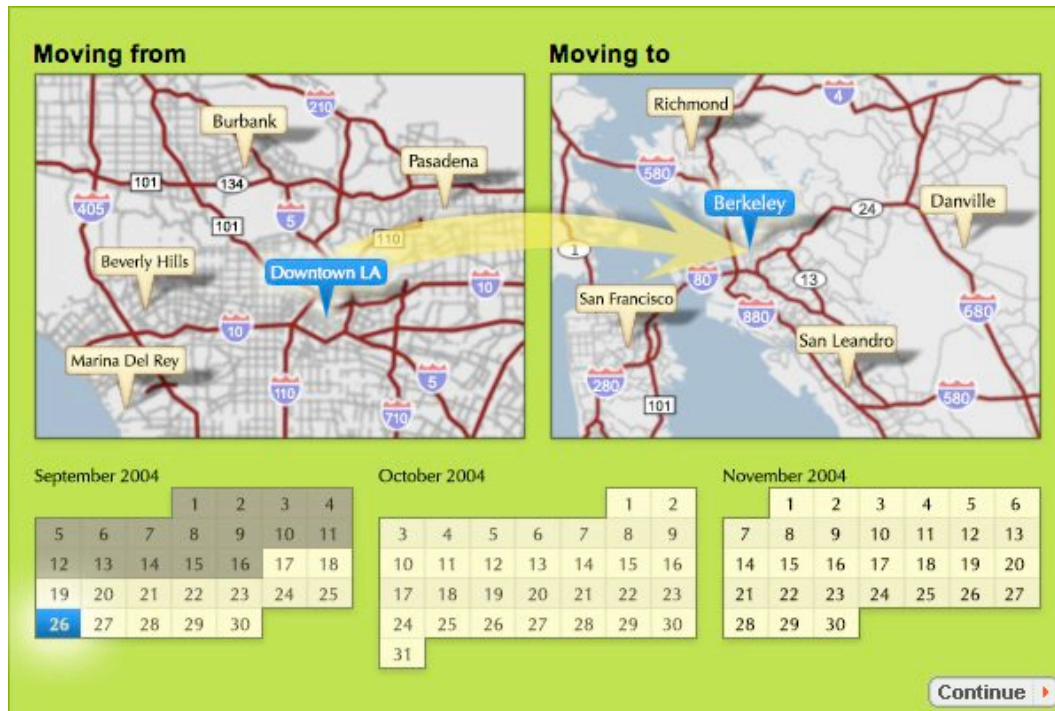
Two of the most fundamental context dimensions are *where* and *when*. For millennia, people have described these concepts with specialized information graphics. But much modern software abandons this tradition, as seen on the website of a popular moving company:\*

\* Based on uhaul.com as of January 2006.





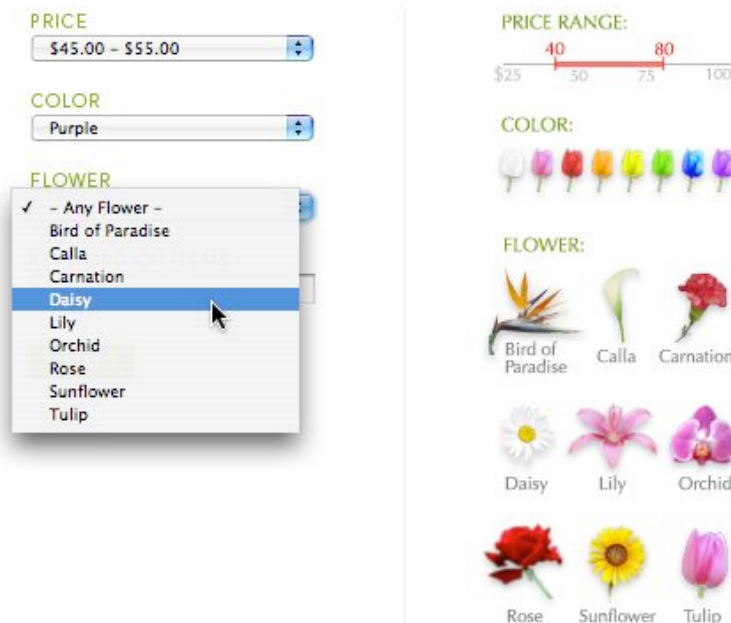
These drop-down menus are awkward and uninformative. Geographical locations belong on maps, and dates belong on calendars. Consider this redesign:



Even this is not ideal. Locations and dates should be chosen from the user's own map and calendar. But until platforms that enable such a thing are widespread, software can at least provide temporary ones.

As an example of more application-specific context, a prominent online flower shop lets the user narrow the view via a set of drop-down menus.\* Compare it with a simple visually-oriented redesign:

\* Based on [teleflora.com](http://teleflora.com) as of January 2006.



Many types of context can be naturally expressed in some informative graphical domain, relieving the user from manipulating information-free general-purpose controls. Several more examples will be given in the case study below.



**Relative navigation.** If the software properly infers as much as possible from history and the environment, it should be able to produce at least a reasonable *starting point* for the context model. Most of the user's interaction will then consist of *correcting* (or confirming) the software's predictions. This is generally less stressful than constructing the entire context from scratch.

For example, Google Maps offers both absolute navigation (typing in an address) and relative navigation (panning and zooming the current map). However, it initially displays by default the entire continent; this effectively demands that the user type in an absolute location to get started.\* A better design might start at the last place the user looked (last-value prediction), with a nearby list of locations predicted by history (recently visited or manually bookmarked) and the environment (addresses extracted from email, open websites, and calendar software). A reasonable starting point would almost always be a click away, and from there, the user could use relative navigation (dragging and zooming) or simply "navigate" by eye if the graphic is detailed enough.

An even better design would recognize the prediction list as information software in its own right, and would take steps to *show the data* (such as annotating the predictions with driving times to and from common locations, taking current traffic conditions into account) and *arrange the data* (perhaps spatially arranging the predictions on their own map). This might answer most of the user's questions without any interaction at all.

**Tight feedback loops.** Salen and Zimmerman offer a game design perspective on a principle that is crucial for all interactive software:

*If you shoot an asteroid while playing a computer game and the asteroid does not change in any way, you are not going to know if you actually hit it or not. If you do not receive feedback that indicates you are on the right track, the action you took will have very little meaning. On the other hand, if you shoot an asteroid and you hear the sound of impact, or the asteroid shudders violently, or it explodes (or all three!) then the game has effectively communicated the outcome of your action.\**

This principle is universal. If the user clicks a checkbox and *nothing happens*, her action is rendered ambiguous or even meaningless. She cannot evaluate a response and let it guide her next action. In terms of Crawford's conversation metaphor, the software is failing to speak back—she is shouting into the wind.

For information software in particular, all interaction specifies context. Thus, **each interaction can and should result in a discernible change to a context-sensitive information graphic.** Providing immediate feedback reduces the amount of manipulation the user must do before either reaching an adequate view or recognizing a wrong turn and backtracking.

Any web form with a "submit" button or dialog box with an "accept" button fails this point. Google Maps offers reasonable feedback during relative navigation, but none during absolute navigation, such as typing in an address. Even a simple predictive auto-complete would be

\* The user can manually specify an initial location, but she will presumably set this to her home. Ironically, her own neighborhood is the *least* likely place she'll need mapped.

Conceptually, a prediction list might itself be considered relative navigation, as a set of "shortcuts" through the data space.

\* Katie Salen and Eric Zimmerman, *Rules of Play* (2003), p35.

helpful, but consider the possibilities suggested by Ben Fry's [zipdecode](#) applet. (First click "zoom" in the lower-right, then type in numbers.) Imagine honing in on familiar areas simply by typing the first few digits of a zip code—type "9" to immediately zoom into the US west coast, followed by "4" to zoom into the SF bay area and then "5" for the east bay. Because of the immediate feedback, the user can stop typing when she gets close enough, and use relative navigation from there.

## How did we get here?

Much current software is interaction-heavy and information-weak. I can think of a few reasons for this.

First, our current UI paradigm was invented in a different technological era. The initial Macintosh, for example, had no network, no mass storage, and little inter-program communication. Thus, it knew little of its *environment* beyond the date and time, and memory was too precious to record significant *history*. Interaction was *all it had*, so that's what its designers used. And because the computer didn't have much to inform anyone of, most of the software at the time was manipulation software—magic versions of the typewriter, easel, and ledger-book. Twenty years and an internet explosion later, software has much more to say, but an inadequate language with which to say it.\*

A second reason why modern software is dominated by mechanical metaphors is that, for the people who create software, the computer is a machine. The programmer lives in manipulation mode; she drives her computer as if it were a car. Thus, she inadvertently produces software that must be operated like a machine, even if it is *used* as a newspaper or book. Worse, the people who design platforms and GUI toolkits are even more prone to this perspective, since they work at a lower level. The application software designer is then almost forced into a mechanical model by the design environment.\*

Even software that starts out information-rich and interaction-simple tends to accumulate wasteful manipulation as features are added over successive versions. It's easier on both the designer and the programmer to plug in another menu item and dialog box than to redesign a dynamic graphic, and sometimes it's justified as a less jarring change for the user. After ten versions, the software can grow into a monstrosity, with the user spending more time pulling down menus than studying and learning information.

Software doesn't have to be this way, but the solution will require a significant re-thinking of both the design process and the engineering platforms. After a detailed case study of one recent design, I will discuss what's needed to usher in the information software revolution.

\* Make no mistake, I revere GUI pioneers such as Alan Kay and Bill Atkinson, but they were inventing rules for a different game. Today, their windows and menus are like buggy whips on a car. (Although Alan Kay clearly *foresaw* today's technological environment, even in the mid-'70s. See "A Simple Vision of the Future" in his fascinating [Early History of Smalltalk](#) (1993).)


\* Apple's Interface Builder, for example, makes it simple to place buttons, sliders, and blocks of text. Dynamic graphics, the cornerstone of information software, must be tediously programmed with low-level constructs.



## Case study: Train schedules

I recently created a program for planning trips on BART, the San Francisco bay area subway system, in the form of a “Dashboard widget” (mini-application) for the Apple Macintosh. The widget went through five iterations over the course of five months. There were no bugs in any released version; the iterations added improvements based on user feedback.



The design has clearly been successful. Even though the target audience is fairly small (SF bay area public transportation riders with the latest Mac OS and knowledge of how to customize it), hundreds of users have sent in wildly enthusiastic praise,\* and the widget was given a rare perfect  rating in Macworld magazine. If you are unfamiliar with the widget, you can watch a one-minute demo movie:



As information software, the widget was approached primarily as a graphic design project. I will discuss how its design exemplifies the viewpoints in this paper, and also point out where it falls short and could be improved.\* I will also compare it to the trip planner on the official BART website, which follows the typical mechanical paradigm of drop-down menus, “submit” button, and table of results.

\* “Amazing” and “awesome” seem to be the top adjectives.

\* The widget originally inspired this paper, not vice-versa. Thus, the widget does not reflect new ideas conceived while writing this. (Yet!)

The BART widget was designed around three classical forms of graphical communication: the timeline, the map, and the sentence.

## Showing the data

Information software allows the user to ask and answer questions, make comparisons, and draw conclusions. In the case of trip planning, some questions are:

- When is the next train leaving? How long is that from now?
- When is that train arriving? How long is that from now?
- Which line is that train on?
- Does that trip have a transfer? Is so, when, where, and for how long?
- What about the train after that? And after that?
- How frequently do the trains come?
- What about trains around 7:00 am on Tuesday?

Users use the answers to compare the available trips, and draw a conclusion about which to take. Naturally, it must be possible for that conclusion to take the form of a plan: “Which train will I take? I will take the 7:32 train.” However, the plan then becomes a mental burden on the user. A good design would also allow for a series of quick boolean conclusions over time: “Should I start walking to the station *now*? No... no... no... okay, let’s go.”

The choice of graphical representation depends on what sort of data space is left after context-based winnowing. What context can be inferred?

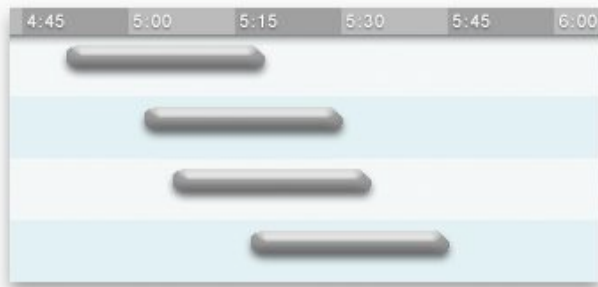
The user is expecting to leave around a particular time; thus, the graphic can exclude trips outside of some narrow time window. Furthermore, the most common time is “soon”; thus, the software can initially assume that the time window is “the near future.” Also, notice that all of the questions implicitly refer to a single route—a particular origin and destination pair. That is, the user wants to compare trips along the time dimension, but not the space dimensions. Thus, the graphic need only concern itself with a single route, which we last-value predict to be “the same as last time.”\*

After winnowing the data, we are left with a handful of trips—ordered, overlapping spans of time. We need a graphical construct that allows the viewer to compare the start, end, and length of each span. A natural choice is a *time bar graph*, which allows for important qualitative comparisons at a glance: When does each span start and end? How long is each span? How close together are they?

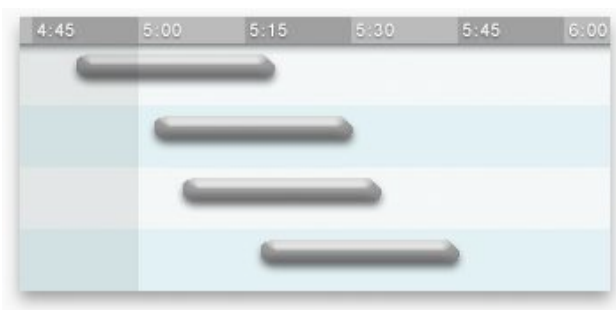
\* A learning predictor for the route is presented later in the paper.

The time bar graph may have been invented by proto-chemist Joseph Priestly in 1765 to compare the [lifespans](#) of various historical figures. Priestly’s chart inspired William Playfair to invent the modern statistical bar graph. Howard Wainer claims to have uncovered a bar graph from 3000 years earlier, plotting population changes in the tribes of Isreal after

the exodus. See [Graphic Discovery](#) (2005), p18.



The most important context, the current time, can be emphasized by shading the past. The most important data point, the next train, can be emphasized by keeping it in a constant location, the second row. This answers the most important qualitative questions: Is the next train coming soon? Did I just miss a train? For an experienced viewer, the conclusive question, “Should I start walking to the station *now*?”, can be answered literally at a glance.



The graphic can then be unobtrusively annotated with quantitative information, so closer inspection answers all of the questions precisely:



Transfers can be regarded as segmentation of the overall trip. The question that must be answered exactly is *where* to transfer. The questions of when and how long should be answered qualitatively; the exact times would be irrelevant clutter.\*

\* A better design would probably place the transfer station name closer to the graphical representation of the transfer, instead of over to the side.



**And that's about it.** Although there clearly is more to the widget than this, most of the “user experience” is represented by the picture above. That is, this software is normally “used” by simply *looking* at it, with no interaction whatsoever. In contradiction to the premise of interaction design, this software is at its best when acting *non-interactively*.

Accordingly, all interactive mechanisms—the buttons and bookmarks list—are hidden when the mouse pointer is outside the widget. Unless the user deliberately wants to interact with it, the widget appears as a pure information graphic with no manipulative clutter.\*

Of course, if the predicted context is wrong, the user must interact to correct it. This involves navigation in the usual two dimensions, *time* and *space*.

\* Tufte uses the term “administrative debris.”

## Navigating through time

The widget initially assumes a time window of “the near future.”\* There are two cases in which this context is incorrect:

- The user wants to see even later trips.
- The user wants to plan for some other time entirely.

**Relative navigation.** To see earlier or later trips, the user can simply drag the graphic around. A cursor change suggests this, as well as a brief message when the widget is first started.\* The mouse scrollwheel and keyboard arrow keys also serve to navigate through time. The “underlying” graphic is infinite—the user can scroll forever. Thus, a GUI scrollbar would be inappropriate.

**Absolute navigation.** To plan around an arbitrary time, the user clicks a button to reveal the hours of the day, from morning to night, laid out linearly. The user can then click anywhere on the mechanism to jump to that time.



The mechanism’s labeling is intentionally vague, so the user will click approximately in the right area, and then continue to drag left or right until the correct information is displayed on the chart of train schedules. This forces the user to keep her eyes on the information graphic, instead of wasting effort precisely manipulating the navigation mechanism.\* Unlike the time of

\* This window changes over time, of course. The widget naturally stays in sync, always displaying relevant information. A button to manually “refresh” the display would be almost obscenely mechanical.



\* This is the same concept suggested by the Google Maps


day, the predicted date (today) is probably close—few people plan subway trips weeks in advance. Thus, the date control is relative.

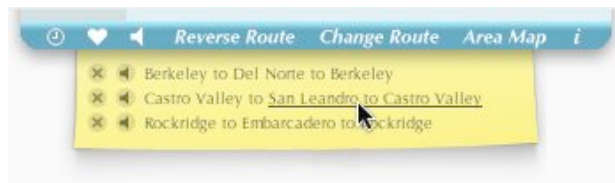
## Navigating through space

The assumed context includes where the user is coming from and where she is going. The assumption is “the same as last time”; that is, this appears as explicit state. There are three cases for which the context is incorrect.

The most common case is that the user is making a round trip, and wants to come home. The “reverse route” button serves this case.\*



The second case is that the user is making a common trip, and knows exactly where she wants to go. The bookmarks feature serves this case. When the user clicks the heart button , the trip is added to a bookmarks list. From then on, that trip and its reverse can be selected with a click. No manipulation is needed to bring up the bookmarks list—it slides out when the mouse is over the widget.\*



The most interesting case is the least common, but the most stressful for the user—selection of an unfamiliar station. The user needs information to decide which station to travel to; thus, this can be approached as an information software problem in itself. Some questions the user might have:

- Where are the stations?
- What are the lines?
- What order are the stations on a particular line?
- Which stations are near a particular area?

These questions involve orientation and navigation in a physical two-dimensional space. The standard graphical device for this situation is the map. The map allows the user to ask and answer the above questions, make comparisons among the available stations, and decide which station she’s looking for.

prediction list above. Instead of precise, tedious absolute navigation, offer quick ballpark navigation, followed by relative navigation in a tight feedback loop.

\* A better design could probably eliminate this interaction with a predictor as described above (and implemented below), or a graphic that somehow incorporates both directions at once.

\* A better design might further reduce interaction by annotating each bookmarked trip with its next depart time. In many cases, that would eliminate the need to even click on the bookmark. Another improvement would be to automatically infer “bookmarks” from recent trips or environmental clues.

This map courtesy of [newmediasoup](http://newmediasoup.com).

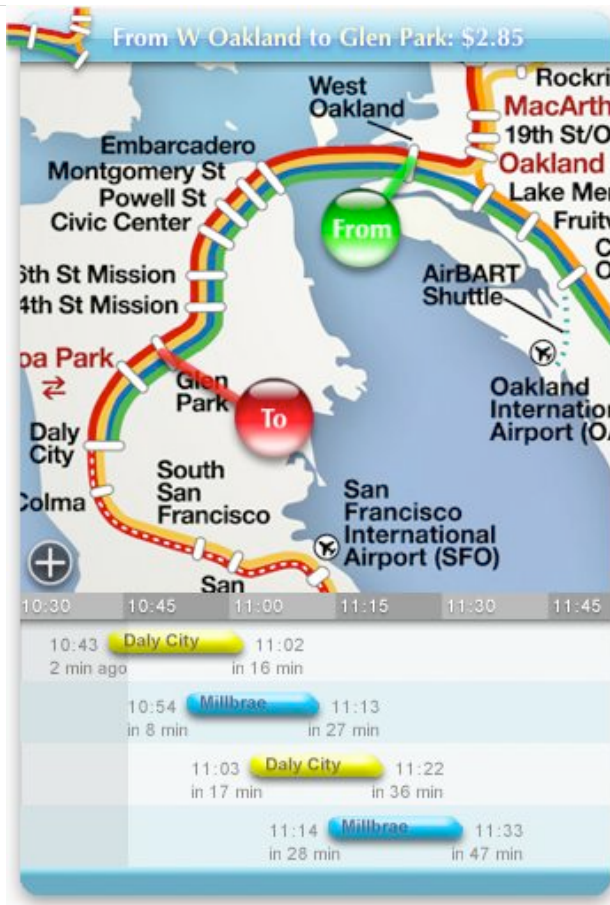




Once the user has decided, she must indicate her selection to the software. This manipulation can be done in the same graphical domain as the information. “From” and “To” markers appear directly on the map; these are dragged to the desired station. Instead of having to name the station, the user effectively points at the map and says, “There!” Although it is less important in this case, the feedback loop remains tight; the train chart updates as the markers are moved.\*

\* Widgets are expected to have a small screen footprint, which is why the map can be hidden. Ideally, the map would always be visible. A better design might then overlay dynamic information on the map, such as the positions of the trains and arrival times at stations.





## Configuring notifications

Instead of the user continually asking “Should I start walking to the station now?”, she might prefer the software to notify her directly: “Start walking to the station now!” Audio works well for infrequent, asynchronous notifications. The widget can speak announcements of upcoming trains. (Hear a [sample](#).)

The design challenge is allowing the user to express if and when she wants announcements. For example, if the user is about ready to go home and it’s a twelve-minute walk to the BART station, she would want the software to announce trains departing in twelve minutes. But if she’s meeting a friend at the station, she would want to hear about trains *arriving* in twelve minutes. Normally, of course, she doesn’t want to hear anything at all.

A typical design would use a preference dialog or form that the user would manipulate to tell the software what to do. However, an information design approach starts with the converse—the *software* must explain to the *user* what it will do. It must graphically express the current configuration.

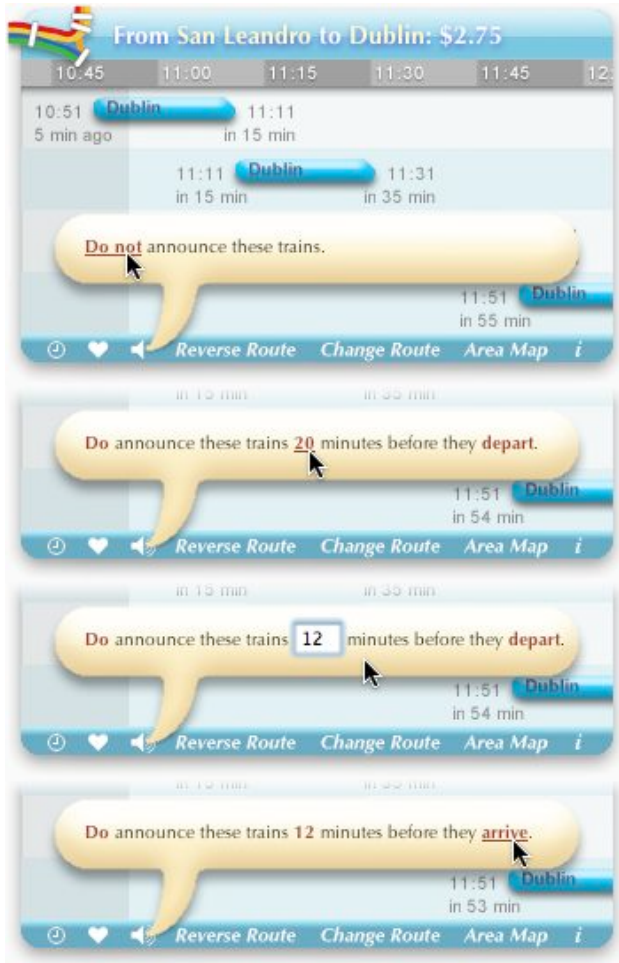
For presenting abstract, non-comparative information such as this, an excellent graphical element is simply a concise sentence.\*

Vocal announcements were originally a semi-hidden Easter Egg, but they got enough of a user response that they were moved up to first-class feature.

\* Chris Crawford discusses the relative merits of pictorial and textual representation in his essay

Announce trains twelve minutes before they arrive.

As with the map, once the information graphic is established, manipulation can be incorporated. In this case, some words are colored red, and the user can click on these words to change them.\*



The user always sees the software presenting information, instead of herself instructing the software. If the information presented is wrong, the user corrects it in place. There is no “OK” or confirmation button—the sentence *always* represents the current configuration. The graphic fades out when the mouse is clicked outside of it or the mouse leaves the widget.

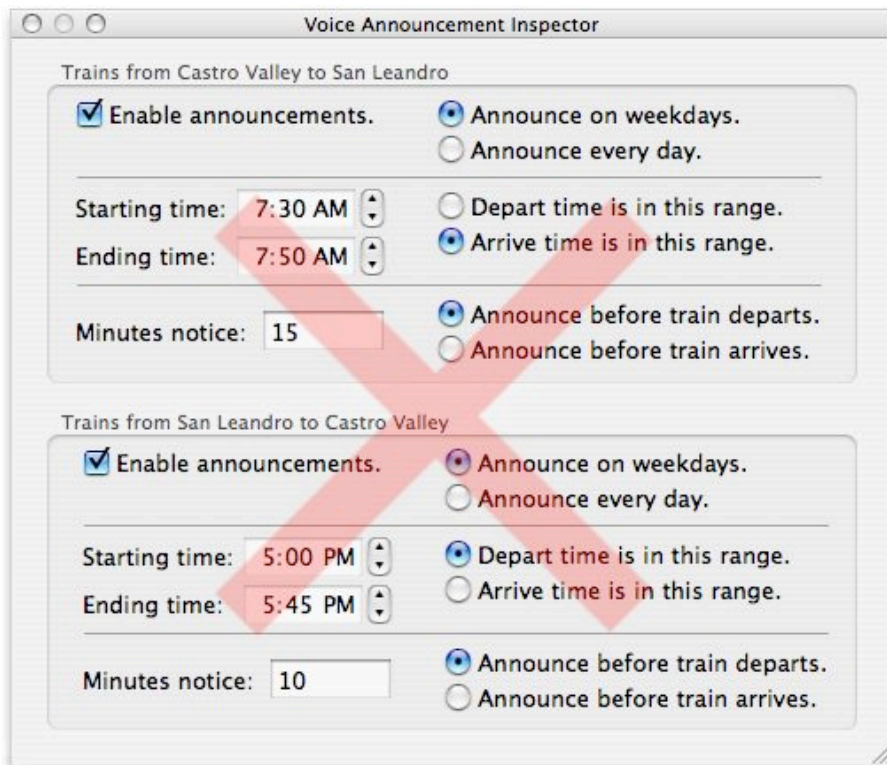
This approach scales well to more complex configuration. The widget allows spoken announcements to be associated with a bookmark and a particular time. This is useful for daily trips, such as to and from work. The user thinks, “It takes me 15 minutes to drive to BART, it takes ten minutes to walk from the station to work, and I have to be at work by 8.” This graphic represents her thought precisely, as well the trip home:



[Representation Versus Depiction](#) (1994).

\* Numerical and time parameters transform into edit controls when clicked—the idea for this was inspired by Jeremy Ruston’s wonderful [TiddlyWiki](#).



Sentence-based configuration scales so well because parameters are given meaning by the surrounding textual context, which can itself consist of other parameters. A typical configuration dialog box attempts to express each parameter in isolation, resulting in intimidating (or bewildering) verbosity:\*



Some additional graphical touches help bring the design together. The sentence is contained within a cartoon speech bubble which, beyond simply looking cute, implies that the activity pertains to speech, and points via the tail to the button which spawned it and the trip to which it refers. More importantly, if a voice announcement is activated, the button's icon changes to an active speaker.   This avoids a "hidden mode" problem by providing a clear visual indication of where the voice is coming from and how to turn it off.

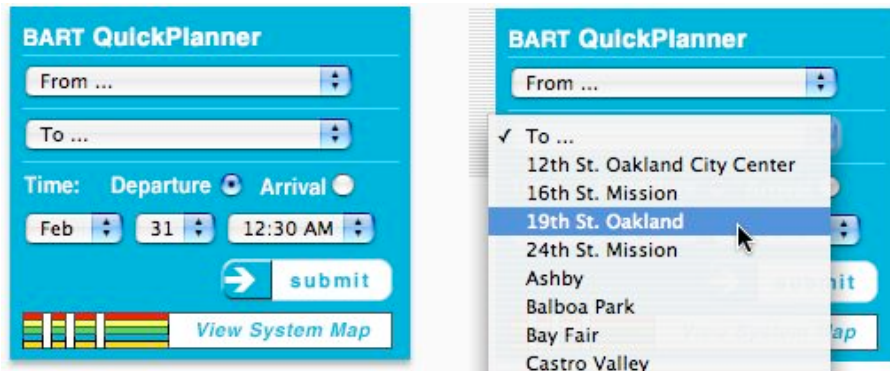
\* Some people claim that no interface can be fundamentally more "intuitive" than another, because intuition is simply a result of familiarity. But surely these people were parsing and producing complete sentences long before they could manage a dialog box. The human brain actually does have some hard-wiring.



## Comparison

The trip planner on the official BART website refuses to divulge any information whatsoever without a sequence of menu selections and a button-push.\*

\* Based on [bart.gov](http://bart.gov) as of January 2006.



Because the BART system is two-dimensional, no linear arrangement of the stations can convey useful information. Instead, the stations are listed alphabetically, but because many stations go by several names (“Berkeley” or “Downtown Berkeley”? “Oakland City Center / 12th St.,” “City Center / 12 St.,” or “12th St”?) the selection is difficult even for those familiar with the system. The user can click a link to see a map, but the map graphic is static; the selection must be made through drop-down menus. Information and navigation are completely segregated, and the feedback loop is enormous.

The results screen shows *no* useful information at a glance:

Your Schedule		Fare: \$3.75	
Depart: <a href="#">Richmond</a>		Date: Jan 31, 2006	
Arrive: <a href="#">Concord</a>		Departing Around: 7:30 PM	
Depart	Board	Arrive	Notes
Richmond at 7:04p	Fremont train	MacArthur at 7:23p	<a href="#">Transfer</a> 🚲
MacArthur at 7:28p	Pittsburg/Bay Point train	Concord at 7:54p	🚲
Depart	Board	Arrive	Notes
Richmond at 7:19p	Fremont train	MacArthur at 7:38p	<a href="#">Transfer</a> 🚲
MacArthur at 7:43p	Pittsburg/Bay Point train	Concord at 8:09p	🚲
Depart	Board	Arrive	Notes
Richmond at 7:35p	Fremont train	MacArthur at 7:54p	<a href="#">Transfer</a> 🚲
MacArthur at 7:58p	Pittsburg/Bay Point train	Concord at 8:24p	🚲
Depart	Board	Arrive	Notes
Richmond at 7:55p	Fremont train	MacArthur at 8:14p	<a href="#">Transfer</a> 🚲
MacArthur at 8:16p	Pittsburg/Bay Point train	Concord at 8:41p	🚲
Depart	Board	Arrive	Notes
Richmond at 8:15p	Fremont train	MacArthur at 8:34p	<a href="#">Transfer</a> 🚲
MacArthur at 8:38p	Pittsburg/Bay Point train	Concord at 9:01p	🚲

[Later Times>>](#)

The starting and ending stations, always the same, clutter the results. Transfers are treated as two separate trips, and the relevant times (the start and end of the entire trip) are in opposite corners, with distracting clutter in between. Not only does the information not stay in sync with the current time, there is no relative time information at all. Other than a “later times” link (which leads to a page with only an “earlier times” link!) navigation through time or space requires hitting the back button and working a drop-down menu.

For all its interactivity, the information here is sparse, poorly presented, and hard to get to. Yet, this sort of design is so typical of software on all platforms, it has almost become an accepted norm. For many people, this is “how computers work.”

## Conclusion

Ironically, the BART widget appears so fresh because its underlying ideas are so *old*. The time bar graph was invented about 250 years ago. The map and the written sentence are both about 5000 years old. They are beautiful, venerable forms of visual communication. The bugs have been worked out. They are universally, intuitively understood.



The pull-down menu, the checkbox, and the bureaucracy-inspired text entry form were invented 25 years ago, desperation devices to counter inadequate technology. They were created for a world that no longer exists.

Twenty-five years from now, no one will be clicking on drop-down menus, but everyone will still be pointing at maps and correcting each others' sentences. It's fundamental. Good information software reflects how humans, not computers, deal with information.

## Demonstration: Trip planning redux

BART's official planner is somewhat of a straw man, since BART has little competitive pressure to provide a quality website. The airline industry, on the other hand, has every incentive to give customers a smooth decision-making experience. However, planning a trip through the sky is almost identical to planning one underground.\*

First, a mechanical, information-free configuration screen:

\* This example is based on [southwest.com](http://southwest.com) as of January 2006, but I checked ten other airline websites and found them (almost eerily) similar.

**Where are you traveling?**  
(See a [map](#) of cities Southwest Airlines serves.)

Depart:	Arrive:	Return:
Oakland, CA - OAK	Ft. Lauderdale, FL - FLL	None
Oklahoma City, OK - OKC	Ft. Myers, FL - RSW	Depart City/Round Trip
Omaha, NE - OMA	Harlingen, TX - HRL	Albuquerque, NM - ABQ
Ontario, CA - ONT	Hartford, CT - BDL	Austin, TX - AUS
Orange County, CA - SNA	Houston Hobby, TX - HOU	Baltimore, MD - BWI

**When are you traveling?**  
(We are currently accepting reservations through January 23, 2006.)

Depart Date:	Depart Time:	Return Date:	Return Time:
September 15	<input type="radio"/> Before Noon	September 22	<input type="radio"/> Before Noon
October 16	<input type="radio"/> Noon - 6pm	October 23	<input type="radio"/> Noon - 6pm
November 17	<input type="radio"/> After 6pm	November 24	<input type="radio"/> After 6pm
December 18	<input checked="" type="radio"/> Anytime	December 25	<input checked="" type="radio"/> Anytime
January 19		January 26	
20		27	

Followed by a table of textual results:

### Select Departing Flight - Oakland, CA to Hartford, CT (Monday, October 17 2005)

Depart Time:  Depart Date:

These fares do not include government fees and taxes. More Fares →

Flights	Departs	Arrives	Stops	Refundable Anytime \$299	Special Fares \$260 - \$275	Restricted Fares \$230 - \$245	Advance Purchase \$200 - \$260	Fun Fares \$170 - \$185
379/487	6:00am	5:35pm	LAX/2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
357/2789	7:40am	5:50pm	LAS/1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Unavailable
1668/1812	8:55am	9:05pm	LAS/2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Unavailable
1111/590	12:25pm	11:05pm	MDW/1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

### Select Returning Flight - Hartford, CT to Oakland, CA (Tuesday, October 25 2005)

Return Time:  Return Date:

These fares do not include government fees and taxes. More Fares →

Flights	Departs	Arrives	Stops	Refundable Anytime \$299	Special Fares \$260 - \$275	Restricted Fares \$230 - \$245	Advance Purchase \$200 - \$260	Fun Fares \$170 - \$185
824/913	6:30am	12:10pm	MDW/1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
711	6:50am	3:00pm	3	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Unavailable
2049/1943	7:20am	2:50pm	PHX/2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
1103	12:20pm	5:00pm	1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
770/362	1:25pm	7:35pm	BWI/2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
267	3:00pm	8:45pm	2	<input type="radio"/>	Unavailable	Unavailable	Unavailable	Unavailable
145/1709	5:45pm	11:40pm	MDW/1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

The actual information is squeezed into a few columns on the left, with most of the screen a monument to Southwest's intricately stratified pricing structure. (Additional columns to the right are not shown.)

What questions might a user have?

- What cities does this airline fly out of? Where are they?
- What flights are available on the days I want to travel?
- When do they depart and arrive?
- How long are they? (This can get confusing across time zones.)
- How many stops are there? How many transfers?

Consider this redesign:

CLICK ON THE CITIES THAT YOU ARE TRAVELING BETWEEN,  
OR CHOOSE A RECENT TRIP.



CLICK A DATE FOR A ONE-WAY TRIP,  
OR DRAG ACROSS A RANGE  
FOR A ROUND TRIP.

	S	M	Tu	W	Th	F	S
Sept.	4	5	6	7	8	9	10
	11	12	13	14	15	16	17
	18	19	20	21	22	23	24
	25	26	27	28	29	30	1
Oct.	2	3	4	5	6	7	8
	9	10	11	12	13	14	15
	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
	30	31	1	2	3	4	5
Nov.	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
	27	28	29	30	1	2	3
Dec.	4	5	6	7	8	9	10
	11	12	13	14	15	16	17
	18	19	20	21	22	23	24
	25	26	27	28	29	30	31
Jan.	1	2	3	4	5	6	7
	8	9	10	11	12	13	14

Blue shaded days indicate special fares.

CLICK A FARE FOR EACH FLIGHT

Depart  Stop  Arrive  YOU'D LIKE TO PURCHASE.



Refundable anytime fares allow for changes to travel plans without a penalty.  
Special fares generally do not require more than one to seven days advance purchase.  
Restricted fares require a round trip with 1 night stayover, and 7 days advance purchase.  
Advance purchase fares require a round trip with 1 night stayover, and 7 days advance purchase.  
Fun fares are only available for turbulent trips.  
Promotional fares are subject to the terms of the promotion.  
Internet special fares are only available for flights purchased through this website.

Refundable ticket from OAK at 8:55 AM to BDL at 9:05 PM	\$299
Fun fare from BDL at 1:25 PM to OAK at 7:35 PM	178
PFC and security fees	25
<b>Total</b>	<b>\$502</b>

VERIFY THE ITEMIZATION ABOVE AND [CLICK HERE TO PURCHASE.](#)



The times and lengths of the flights, and the count, times, and lengths of stops and transfers, can be compared visually. Trips without transfers stand out because they are entirely blue; non-stop flights would appear unbroken. Anomalies, such as the 6:50 from Hartford which arrives *later* than the 7:20, stand out literally. Times can be converted into either time zone simply by referencing the appropriate header bar.

There is some attempt to use color symbolically. On the map, the calendar, and the flight chart, green represents “home,” and yellow the destination. However, it is not critical that the user notice this.

Interaction is simplified to the point where a short, instructive sentence can describe each and every click. At the most, the user will click twice on the map, drag across the calendar, and click twice on the ticket prices, possibly with some page scrolling. Last-value prediction (automatically selecting the last route purchased, and displaying a list of recent trips) may eliminate or reduce the map clicks for many travelers. A learning predictor, capable of inferring that the user always spends the first Monday through Friday of the month in Baltimore and selecting that range on the calendar automatically, could eliminate all context-establishing interaction, leaving only the decision-conveying interaction of clicking ticket prices. Of course, since everything is on the same page and feedback loops are tight, the user can explore different dates and cities, and see the available flights immediately.

With air travel in a slump for the past few years, airlines have been desperate for any passengers they can get. Unsuccessful ones have even faced bankruptcy. With so much at stake, why hasn't *any* airline attempted to improve the ticket-buying experience through better software design?

The problem is primarily *cultural*. Asking “Why doesn't Southwest design better software?” is challenging the symptom, not the disease. The real question is, “**Does software design exist yet?**” Before we can expect better airline websites, we may need to change a worldview.



## Designing the information software revolution

Mass production of machines emerged at the start of the 20th century. Henry Ford's assembly line methods spread throughout the manufacturing world, dramatically lowering production costs and making a variety of machines affordable for the average person. But many of these products were unpleasant to interact with. Between the businessman's specifications and the

engineer's implementation, there was no *design*. Within a few decades, a new profession arose to fill the gap—industrial design.

The next revolution in the mass production of machines was software. The late 1970s saw the rise of the personal computer, a device capable of behaving as *any* machine—typewriter, adding machine, filing cabinet, arcade game—when given the right instructions.

Manufacturing a “machine” was now just a matter of printing its instructions onto a disk, and production costs plummeted. But much of this software was unpleasant to interact with.

Between the businessman's specifications and the engineer's implementation, there was no *design*. Within a couple decades, a new profession arose to fill the gap—interaction design.

The mass production of *information* has a very different history than the mass production of machines. Industrial design brought art to existing mass-produced technology, but printing brought mass-producing technology to an existing art.

Before the 15th century, books were precious and extremely rare, for each had to be copied by hand. A single book might cost as much as a farm. Books were also exquisite works of art, carefully lettered in calligraphy, lavishly illustrated and decorated. In the 1440s, Johann Gutenberg's movable type press boosted book production over a *thousand-fold*, making books affordable (and literacy worthwhile, and political awareness possible) for the average person for the first time. Fortunately, Gutenberg and contemporary printers were exceptionally devoted to the art form, and took great pains to preserve the quality of the hand-lettered page.\* The explosion of new books of all kinds, as well as the rise of the broadside (precursor to the poster and the newspaper), created a great demand for artists in the new medium, many of whom transitioned from the old medium. The art of laying out a page eventually became known as graphic design.

The next revolution in the mass production of information was the web. Unlike early printers, unfortunately, early web technologists cared little for the artistic qualities of their predecessor, but the capabilities eventually evolved to approximate the printed page on the computer screen. Publishing was now just a matter of sending bits through a wire. The explosion of websites created a great demand for artists in the new medium, many of whom transitioned from the old medium. The art of laying out a webpage became known as web design.

These parallel evolutions have produced designers for interactive machines (conventional software) and designers for static page layouts (conventional websites). From this viewpoint, the chimeric effects of convergence are almost to be expected. The emerging “interactive web” embraces a ludicrously mixed metaphor of *machines on pages*, a monstrous hybrid of virtual mechanical affordances printed on virtual paper. Information is trapped behind interactive mechanisms *and* presented in static layouts—it is the worst of both worlds.

Good context-sensitive information graphics are neither interactive nor static, neither machines nor page layouts. Design has not evolved to produce them. The culture is blind to the possibilities.

Who will draw information software? And how?

\* Gutenberg's emulation of calligraphy was so accurate, his bibles were sold as handmade manuscripts in Paris. When people noticed the quantity and similarity of the books, they did not suspect printing, but witchcraft! See Philip Meggs's superb [History of Graphic Design](#) (2005).

For related historical allegories, see Peter Drucker's fascinating essay [The Next Information Revolution](#) (1998).

same standards that they hold print. People constantly settle for ugly, clunky software, but demand informative, professionally-designed books, newspapers, magazines, and—ironically—brochures, ads, and manuals for that very software.\* Though once justified by technological limitations, this double standard is now dangerously obsolete. It is the first and largest obstacle to revolution. Without consumer demand, design appears to give no return on investment.

Prominent usability pundits have claimed that the public is becoming more discriminating, but since this claim underlies their consultancies' sales pitch, it is far from an unbiased observation. I see the opposite—as technology races ahead, people are tolerating increasingly worse design just to use it. The most beautifully-designed DVD player will go unsold if the competition costs the same and has S-Video output, or plays MP3s from memory sticks. Good design makes people happy, but feature count makes people pay.

I don't know the solution to cultivating a culture of good taste, but I believe lessons can be learned from the emergence of industrial design, about seventy years ago.\* At a time when many products competed on ornamentation, the simplified, functional creations of industrial designers were too untraditional to sell on looks alone. The salesman made inroads by directly touting the tangible benefits of good design, such as comfort and safety. He would demonstrate to a homemaker how his vacuum cleaner or iron was designed to reduce fatigue and cramping. He would demonstrate to a farmer how his machinery was designed to eliminate the finger-severing accidents that were, to that point, distressingly common. Explicitly informed of the benefits, people gradually came to demand, then expect, such conscientious design in their everyday products.

Today, software consumers demand technological features because software marketing *presents* features. Consumers ignore design because marketing ignores design. The cycle is vicious, but perhaps vulnerable too—some brilliant new software with engineering, design, and marketing all in sync may raise the bar for everyone.

**The second step** toward the information software revolution is finding people with **talent** for visual communication. Currently, almost all software is designed by people who are very comfortable with computers; their interest in technology motivated them to enter the field. This suggests an enormous exclusion of potential talent—imagine if all graphic designers had to be comfortable running a print shop!\* I believe that ideal candidates for software design are those who have achieved mastery of information graphics in other mediums. There may be multitudes of artists, currently drawing business graphics or maps or comics, who could excel at information software design if they had any idea that it was a legitimate artistic field. Recent years have brought a wealth of beautiful amateur websites, created by visually-oriented people dabbling in the only sort of software design accessible to them. But because full-fledged software is seen as an artifact of arcane technology, a product of “programmers,” these people lack the confidence to consider life beyond HTML.

**The third step** is complementing the designer's talent with **skill**. Skill is achieved through *education* and *practice*, but dearth of the former has given aspiring designers no entry point—

\* As brochures have become websites, this duality has veered into absurdity: “Let's design beautiful software to sell our ugly software!” The wrapper tastes better than the candy.

\* See the chapter “Through the Back Door” in Henry Dreyfuss's recently rereleased autobiography [Designing for People](#) (1955).

Other factors that boosted industrial design were fashion (top designers were promoted as celebrities) and price (good design often *lowered* manufacturing and materials costs). See Raymond Loewy's autobiography [Industrial Design](#) (1979). Both factors can be applied to software.

\* One might argue that the entire next generation will be comfortable with computers. But comfort with today's “computers” may prove irrelevant—who can say what a “computer” will be in twenty years? It is better to look for interest and talent in communicating with people, not with technology, since people don't change nearly as fast.

they are expected to learn the art through osmosis and guesswork. Effective education can entail any, but ideally all, of: classes, books, and examples.

- **Classes.** The renowned [Art Center College of Design](#) in Pasadena offers forty courses in industrial design. Students learn art theory, draftsmanship, and visual communication theory. They learn about form, and the visual and tactile properties and constraints of materials. They learn about cognitive and behavioral psychology, and explore how users experience products. They follow the entire production process: researching the needs of the target markets; sketching ideas and proposals; drawing detailed renderings; designing virtual 3D models; constructing physical models out of clay, plastic, and fiberglass; constructing a functional mechanical solution; designing logos and retail packaging. They learn to devise artistic solutions to problems, to think creatively and think critically, to invent concepts and critique those of others. They interact with industry representatives and do team projects under corporate sponsorship.

Art Center offers only five courses that could be somewhat related to information software.\* For the most part, students learn to make websites. There is nowhere near the breadth or depth offered to designers of physical products. Art Center clearly knows how to put together an applied arts curriculum. What's missing is the understanding of software as an applied art.

There are other schools that offer specializations in “information architecture,” “usability,” and other recently-coined areas, but these subjects approach software design from a scientific perspective, neglecting the essentially artistic aspect of visual communication and the creative and critical techniques used by art schools for centuries. Experimental analysis can be valuable, but only if an artist has created a design worth analyzing.

- **Books.** Information software design will need a body of pedagogical literature, once enough theory is developed to make pedagogy possible. Until that point, the student has little recourse—the closest established areas, information graphic design and “user interface design,” are both severely underserved.

The paucity of literature on information graphic design is bewildering. Edward Tufte's books are highly acclaimed, and deservedly so, but they almost win their titles by default. In a typical bookstore, they are lost amidst a sea of fashion rags masquerading as graphic design guides, or perhaps submerged in a “computer” section overflowing with the latest engineering fads. They have too little company to define a category.\*

The shortage of *good* books on user interface design is more understandable, since pedagogy requires a working paradigm—the status quo must be at least acceptable. Accordingly, I haven't yet found a *textbook* that is at all helpful for software design; the only books I've found worthwhile are the few that *challenge* the status quo and present fresh, progressive ideas. For the field to progress, we need less recycled platitudes and more cutting-edge research.

\* All five use “interactive” as a synonym for “software”: [Interactive Structures, Information and Interactivity, Branding and Interactivity, Interactive Design 1](#), and [Interactive Design 2](#). The intent of this example is not to malign Art Center's curriculum, but to demonstrate the lack of resources for the aspiring software designer.

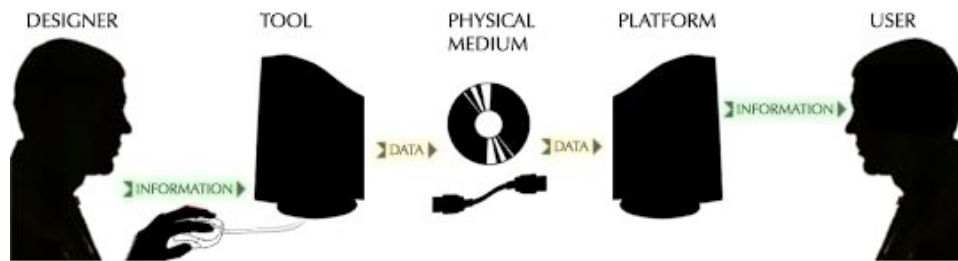
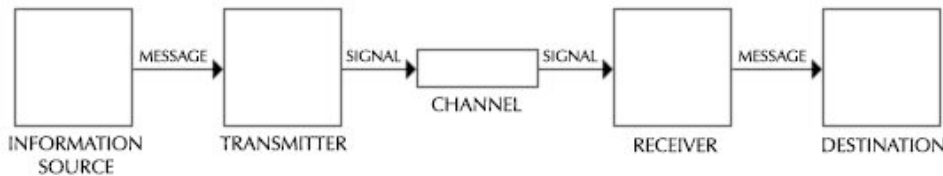
\* Their best company is probably William Cleveland's [The Elements of Graphing Data](#) and [Visualizing Data](#). I doubt you will find either in a bookstore.

The industrial design literature,

the failure of software engineering schools to provide examples of great works, expecting students to somehow derive style from first principles.\* Since software *design* isn't yet recognized as an artistic field in the first place, its situation is even worse—the very concept of a gallery of software designs will seem absurd to most people. But a corpus is crucial for the development of any artistic field. Outstanding designs must be *recognized, collected, and explicated*. Furthermore, outstanding *designers* should be recognized and encouraged to teach, instead of hidden behind a corporate label.

**The fourth step** is supplementing the designer's talent and skill with **tools and platforms**. These two terms are vague in common usage. I will define a *tool* as a communication device that a designer has control over, and a *platform* as a communication device that a recipient is expected to provide. This is best demonstrated with Claude Shannon's communication model:

Shannon's general communication model\*



Analogous model for tools and platforms

A tool encodes mental *information* into physical *data*, which can travel in a physical *medium*. A platform decodes the physical data into the mind of the recipient. Because all information transfer short of telepathy requires some medium, this model is universal. If I write you a letter, my tools are pen and paper, and your platform is knowledge of my written language. If I broadcast a radio signal, my tools are a microphone and transmitter, and your platform is a radio receiver. In general, my tools are whatever I use to make the thing I hand off to you. Your platform is whatever I'm counting on you to already have.

To deliver her message most effectively, the visual designer needs *as much control as possible* over what the viewer sees. But, by definition, the designer only has direct control over the *tool*. She is at the mercy of whatever platform implementation the recipient happens to supply. This implies that a good platform must be as *simple* and as *general* as possible.

- **Simplicity.** From a practical (and historical) standpoint, we can assume that *no complex specification will be implemented exactly*. This, in itself, is not a problem. However, multiple, decentralized implementations of a complex specification will be incorrect in different ways. A platform consisting of the union of all possible

incidentally, seems to consist primarily of photographs of chairs. I don't know what that means. This is how that works. engineering disciplines as well, though less discussed. I can think of only one exception from my own schooling in electrical engineering—David Rutledge's innovative [introduction](#) to analog electronics, taught through gradual construction and analysis of a commercial radio transceiver. Engineering study typically focuses on how something *should be done*, not how it *has been done*, to the detriment of the culture.

\* Adapted from Claude Shannon, [A Mathematical Theory of Communication](#) (1948), p2.

"I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies, and the other way is make it so complicated that there

implementations is thus arbitrarily *unreliable*—the designer can have no assurance of what a recipient actually receives. **For a platform to be reliable, it must either have a single implementation, or be so utterly simple that it can be implemented uniformly.** If we assume a practical need for open, freely implementable standards, the only option is simplicity.\*

- **Generality.** If we think of a computer as a machine that runs software, then in some sense, all data handled by a computer platform must be “software.” The data making up a JPEG image, for example, can be thought of as the encoding of a *program* that describes a picture. (This is sometimes called the “data is code” equivalence.) But the limitations of the JPEG platform result in severely lobotomized “programs”—they cannot animate, respond to context, incorporate new compression techniques, or otherwise take any advantage of the *computer* beyond what JPEG explicitly allows. A crippled platform cripples a designer’s means of expression.

In order for a designer to take full advantage of the medium, a good platform must provide safe access to *everything* that is technologically possible. A platform for information software must offer: inputs from the environment (that is, communication with other software and physical sensors), from history (that is, storage), and from the user (that is, interaction); computational resources with which to respond to inputs; and unrestricted graphical output. Anything less robs information software of its full potential. The proper way to prevent destructive behavior is a well-designed security model, not arbitrarily amputating the computer’s capabilities.

Alarming, the latest platforms forgo both of these virtues. CSS, a language for specifying visual appearance on the web, is a particularly egregious example. It is so complex that it has *never* been implemented correctly; yet, successive versions specify even more complexity. At the same time, it is so underpowered that many elementary graphic designs are impossible or prohibitively difficult, and context-sensitivity (or *anything* computational) must be addressed externally. Most CSS lore is dedicated to describing the tangles of brittle hacks needed to circumvent incompatibilities or approximate a desired appearance.

One cause of the CSS mess is the eschewing of elegant, flexible abstractions for “1000 special cases,” a detrimental approach which precludes simplicity and generality in any domain. However, the larger and more germane fault is the language’s attempt to serve as *both tool and platform*, thereby succeeding as neither.

For universal reliability, the ideal platform must be optimized for ease of implementation. For artistic expressiveness and exploration, a *tool* must be optimized for the designer’s manipulation. Thus, the tool and platform cannot be the same—we must expect a layer of translation between what the designer works with and what the platform interprets.\*

A simple and general platform shifts complexity to this translation layer—the tool’s “back end”—where the designer has control over it. If a particular tool is implemented incorrectly, the designer can work around its particular idiosyncrasies, or switch to a different tool. (It is much

are no *obvious* deficiencies.” C.A.R. Hoare, [The Emperor’s Old Clothes](#) Turing Award lecture (1980), p81.

\* POSIX, Java, and newer web standards (DOM, CSS) are some attempts at universal platforms (for various domains) that have proven too complex to implement uniformly. In each case, the power of the platform is effectively constricted to some simple, reliable *subset*, and enormous time is wasted designing around incompatibilities. By contrast, JPEG, MP3, and modern CPU instruction sets are universally dependable, because much of the complexity is placed at the encoding tool, not the decoding platform. (Almost a century ago, a similar justification was used to reject single-sideband public radio.) The complex Perl and Flash platforms are dependable only because they have centralized implementations.

\* There is a direct analog with RISC computer processors, whose simplified instruction sets are targeted at compilers, not programmers. This considerably eases implementation of the processor, although the motive in

easier for a designer to switch or upgrade tools than for a sea of users to switch or upgrade platforms.) Meanwhile, the tool’s “front end”—that which the designer interacts with—can be simple or complex, general or domain-specific, according to the designer’s needs and skill level.

this case is more performance than reliability.

The platform must make it *possible* to create information software. The tool must make it *easy*. A specific look at some tools and platforms for information software will be offered in the next few sections.

**The fifth and final step** into the information software revolution is an **environment** where experimentation, evolution, and interplay of ideas can thrive. Much like our geological environment, a creative environment can become fatally polluted by short-sighted business interests.

Before 1786, authors invariably presented quantitative data as tables of numbers. In this year, an economist named William Playfair published a book called *The Commercial and Political Atlas*.<sup>\*</sup> In order to illustrate his economic arguments, Playfair *single-handedly* invented the line graph, the bar graph, and the pie chart, and thereby the entire field of statistical graphics. Within years, his inventions had spread across Europe, transforming the landscape of visual communications and heralding an age of discoveries in data made visible. Today, children take these graphical forms for granted; they seem as obvious and fundamental as written language.

\* Remarkable recent efforts have brought this classic back into print, as [Playfair’s Commercial and Political Atlas and Statistical Breviary](#) (2005).

Imagine if Playfair had patented his invention and prosecuted his imitators, suppressing the crucial period of initial excitement and growth. Would we today be staring at tables of numbers, unable to apply our visual cortex to unlocking their patterns?

**This path is inevitable**, for it is the path of all artistic media. Books, newspapers, and the static visual arts have already completed it, or almost so. Movies, television, and published music are struggling at step five, but completion is only a matter of time. For information software as well, it is only a matter of time. But a decade or a century?

Of course, design is nothing without implementation. If information software is to consist of dynamic graphics that infer from history and the environment, it must be possible and easy to create such things. The following sections will discuss a design tool for dynamic graphics, and engineering approaches to inferring from history and the environment.

## Designing a design tool

Software tools for drawing static graphics or composing static animations have long been commonplace. But the designer who wants to create *dynamic* graphics—graphics whose properties are data-dependent—currently has two undesirable options:

- She can learn some sort of programming language. Many designers are intimidated by engineering and may lack the talent or desire to program. They are completely justified—**drawing is a visual activity, and working with textual abstractions is entirely inappropriate**. Painters, illustrators, and sculptors manipulate the artifact directly—



there is no abstraction, and visual feedback is immediate. Would we have any of our great works of art if the creators had to work with “rectangle.width = 17” instead of visible brushstrokes?\*

- Alternately, a designer can draw a series of *mockups*, snapshots of how the graphic should look for various data sets, and present these to an engineer along with a verbal description of what they mean. The engineer, who is skilled in manipulating textual abstractions, then implements the behavior with a programming language. This results in ridiculously large feedback loops—seeing the effect of a change might take a day instead of a second. It involves coordination and communication between at least two people, and requires that the designer *justify* herself—she must convince the engineer and possibly layers of management that each change is worth the engineer’s time. This is no environment for creative exploration.

There is nothing wrong with the concept of drawing mockups. It is a natural, visual way to work, and is ubiquitous across many artistic disciplines, from architecture to industrial design. The problem lies with engineering the behavior the mockups describe. But, consider what exactly the engineer does. From a set of mockups, the engineer *infers the pattern* they conform to—how the graphic changes as a function of the data—and codifies this inferred pattern in a computer program.

Is a human really necessary? Couldn’t this pattern be inferred by a software tool instead?

Going down this path leads to a computer science discipline known as “**programming by demonstration**” (PBD) or “programming by example.”\* This field is concerned with teaching behavior to a computer implicitly, through a series of examples, rather than with explicit instructions. Researchers have created systems (with varying degrees of success) for constructing interactive GUI widgets, defining parameterized graphical shapes, moving and renaming files, performing regular expression-like text transformation, and other domain-specific tasks. With these systems, the user typically performs a few iterations of a repetitive task manually, and the system then performs the rest according to an inferred generalization, perhaps asking for clarification or confirmation.

This section outlines a hypothetical but plausible tool to allow designers to create dynamic data-dependent graphics with *no conventional programming*. These dynamic graphics would serve as the user-facing visible representation of information software. In a sense, this is a tool for “drawing information software.”

The tool can be considered an extension of a conventional vector-oriented drawing program.\* Using the same drawing process as with a conventional tool, the designer draws a mockup of the graphic—how the graphic should look for some particular set of data. She then takes a *snapshot* of this graphic, and indicates the data set that it corresponds to. She then modifies the graphic to correspond to a slightly different data set, takes another snapshot, and so on. Each snapshot serves as an example. With well-chosen examples, the tool will infer how to generate a graphic for arbitrary data.

\* Early music composers typically worked in silence, with pen and paper, and did not actually hear their compositions until they were presented to musicians. Composers who couldn’t handle this abstraction were belittled. With the growing popularity of the clavier and harpsichord, and then the piano, it became acceptable for composers to hear their creations as they composed. Most of our classical masterpieces were composed in this way. Today, not only is every composer expected to work at an instrument, illiteracy is even becoming acceptable!

\* See Allen Cypher (ed.), [Watch What I Do](#) (1993, available online) and Henry Lieberman (ed.), [Your Wish Is My Command](#) (2001). Both are compendia of research projects, not textbooks.

\* Popular examples of drawing tools are Adobe Illustrator and Macromedia Flash. The necessary feature is the representation of graphical elements as objects with variable properties, rather than as arrays of pixels.

The concept of snapshots may have been introduced by David Kurlander’s [Chimera](#) (1991), which


This tool is significantly less ambitious than many in the literature, for several reasons:

used common features in a set of snapshots to infer constraints while drawing a static graphic.

- Many systems use “programming by demonstration” as a means toward *end-user programming*, typically to allow novices to automate repetitive tasks. This tool is intended for professional designers with specialized skills and training, and thus can assume a higher level of user sophistication.
- Many systems attempt to infer a full computational procedure, and have the most difficulty with computational concepts such as conditionals and iteration. As we will see, this tool mostly has to infer *mappings* from some set or numerical range to another—functions in the *mathematical* sense rather than the (imperative) *computational* sense. This may be significantly easier.
- Similarly, many systems attempt to infer stateful programs. This introduces enormous complexity, because the user must teach an output that depends on both the input and a potentially large hidden state. A graphic is stateless; its appearance is only a function of the input data. Examples require no context.

**Demonstration.** I will demonstrate how we might use this tool to design the BART widget described above.



**Train component.** We start by modeling a single train bar.  This graphic has a number of dynamic aspects: position, length, color, and label. For now, we will just handle the color and label. We draw a picture, take a snapshot, and indicate the data properties that it corresponds to:

Train snapshots:



This is what a **Train** should look like if the “Line” property is “Orange” and the “Destination” property is “Fremont.” With only a single example, of course, the tool cannot infer anything dynamic. Let’s teach it how to change the label.

Train snapshots:



Compare these two snapshots. The graphics are exactly the same, except a label has changed from “Fremont” to “Richmond.” The data is exactly the same, except the “Destination”

property has changed from “Fremont” to “Richmond.” The simplest inference is that the “Destination” property should be used as the label text. The tool will learn and use this rule, provided no other example contradicts it.

We now teach the tool how to respond to the “Line” property.

Train snapshots:



The graphics in the new snapshots are exactly the same as the orange-line Richmond-bound example, except for hue adjustments. The data is exactly the same, except for the “Line” property. Thus, the tool infers that each given value of the “Line” property corresponds to a particular hue adjustment.

At this point, the tool should understand how to draw a **Train** for any arbitrary data set, as long as “Line” is within the provided domain. (How we know that it has learned correctly will be discussed below.)

If we want to clarify the model for posterity, we can add *visual comments* simply by drawing outside the snapshots:



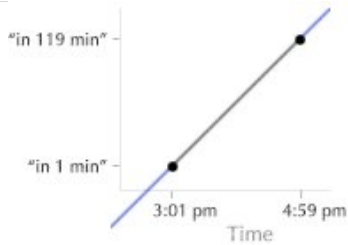
**When component.** Now, we’ll model the time-related text that appears to the left and right of a **Train**. We will use two data properties. “Now” refers to the current time, and “Time” refers to the start or end of the trip. Here are our first two snapshots:



When snapshots:

3:01 in 1 min	Now: 3:00 pm Time: 3:01 pm
4:59 in 119 min	Now: 3:00 pm Time: 4:59 pm

In these examples, “Now” stays constant while “Time” varies. The tool will easily infer that the first row corresponds to “Time” (again, as long it doesn’t contradict further examples). The second row is more problematic. The tool infers linear relations when given two points, so our examples indicate this relation:



The correct relation actually depends on “Now,” but we haven’t yet demonstrated variance with respect to “Now.” Our third snapshot does so:

When snapshots:

3:01 in 1 min	Now: 3:00 pm Time: 3:01 pm
4:59 in 119 min	Now: 3:00 pm Time: 4:59 pm
2:01 in 1 min	Now: 2:00 pm Time: 2:01 pm

The simplest non-trivial relation now depends on the difference between “Now” and “Time”, which is correct:\*

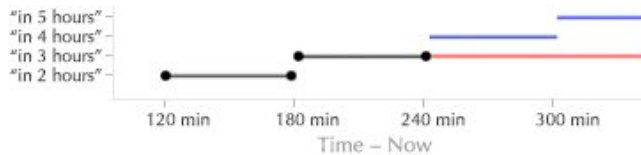


Now, let’s teach the tool how to present larger time differences:

When snapshots:

3:01 in 1 min	Now: 3:00 pm Time: 3:01 pm	5:00 in 2 hours	Now: 3:00 pm Time: 5:00 pm
4:59 in 119 min	Now: 3:00 pm Time: 4:59 pm	5:59 in 2 hours	Now: 3:00 pm Time: 5:59 pm
2:01 in 1 min	Now: 2:00 pm Time: 2:01 pm	6:00 in 3 hours	Now: 3:00 pm Time: 6:00 pm
		6:59 in 3 hours	Now: 3:00 pm Time: 6:59 pm

This gives us the following relation, with interpolation in black and two possible extrapolations in red and blue.



The blue extrapolation is desired. The tool can probably infer it, since it results in an arguably simpler relation. (The red interpretation makes “in 2 hours” a special case, whereas the blue interpretation understands it as part of a general rule.) But if the tool infers incorrectly, the designer can easily correct it. (How so will be discussed below.)

\* To understand how the tool might figure this out, let us take  $f(x,y)$  to be the number in the text label, and  $x$  and  $y$  to be our two data properties (here expressed as minutes since 3:00, although any units and origin will work). The three snapshots give us these constraints:

$$f(0,1) = 1$$

$$f(0,119) = 119$$

$$f(-60,-59) = 1$$

One of the simplest and most naturally-occurring functions of two variables is linear combination:

$$f(x,y) = ax + by + c$$

Solving for the coefficients gives us

$$a = -1, b = 1, c = 0$$

Because linear combination results in such simple coefficients, the tool will have high confidence in this inference, and will use it unless contradicted by some other example.

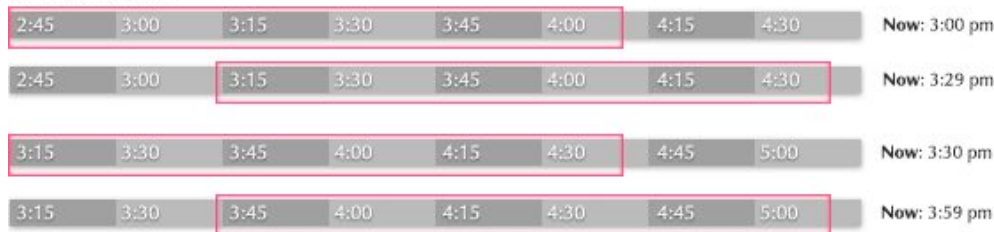
We now provide snapshots to cover earlier times.

**When snapshots:**

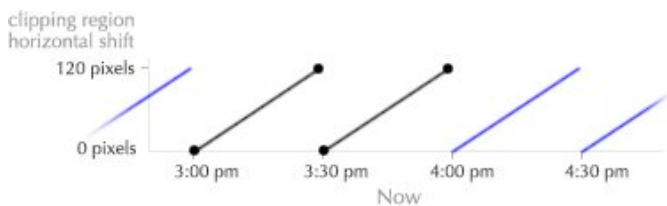
3:01 in 1 min	Now: 3:00 pm Time: 3:01 pm	5:00 in 2 hours	Now: 3:00 pm Time: 5:00 pm	2:59 1 min ago	Now: 3:00 pm Time: 2:59 pm	1:00 2 hours ago	Now: 3:00 pm Time: 1:00 pm
4:59 in 119 min	Now: 3:00 pm Time: 4:59 pm	5:59 in 2 hours	Now: 3:00 pm Time: 5:59 pm	1:01 119 min ago	Now: 3:00 pm Time: 1:01 pm	12:01 2 hours ago	Now: 3:00 pm Time: 12:01 pm
2:01 in 1 min	Now: 2:00 pm Time: 2:01 pm	6:00 in 3 hours	Now: 3:00 pm Time: 6:00 pm			12:00 3 hours ago	Now: 3:00 pm Time: 12:00 pm
		6:59 in 3 hours	Now: 3:00 pm Time: 6:59 pm			11:01 3 hours ago	Now: 3:00 pm Time: 11:01 am

**Timeline component.** Time extends infinitely; thus, the timeline is conceptually an infinitely-wide bar. Of course, only a portion of this bar is actually visible at any given instant. Dealing directly with infinite graphics will be discussed below. Here, I will demonstrate how this can be easily simulated with a normal graphic.

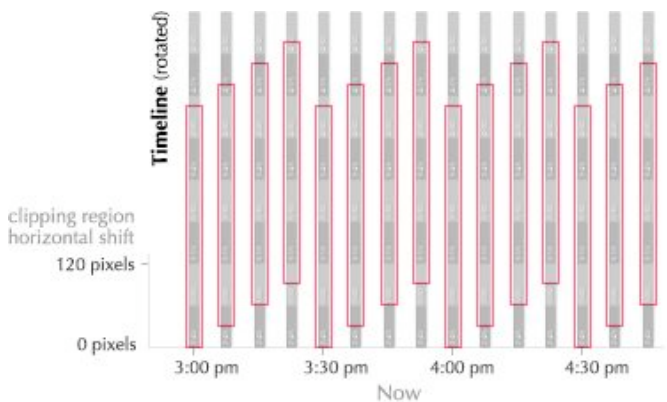
**Timeline snapshots:**



The red box indicates the clipping region of the graphic. The section within the box is the portion that will actually be visible. These snapshots differ from each other in only two aspects: the position of the clipping region and the text labels. The inferred position of the clipping region is shown below as a function of “Now,” with the interpolation in black and extrapolation in blue:

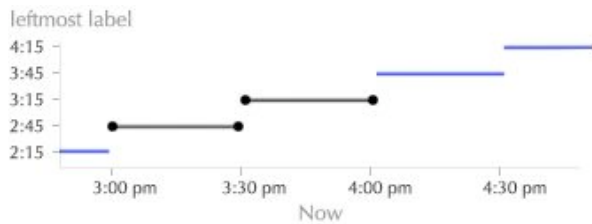


Because the graph above is somewhat abstract, it may easier to view images of the **Timeline** itself (rotated sideways) as a function of “Now”:



We can see that the clipping region slides rightward with time, snapping back to the left on the half hour. The cyclic extrapolation can either be inferred by the tool or specified by the designer, as will be explained below.

The first text label's value as a function of "Now":



That is, the text label is "rounded down" to a multiple of fifteen minutes. The rest of the labels will be inferred similarly. With a little thought, it is clear that this graphic, when cropped to the red rectangle, appears to scroll boundlessly with respect to "Now."

**Row component.** Next, we combine some of the components created above to form a compound component:



Adjacent pairs of snapshots describe how to adjust, respectively, the end point of the **Train**, the start point of the **Train**, and the clipping region:



Notice that adjustments were made within individual components. The length of the **Train** was changed, and the second **When** was right-justified.\* The tool allowed "Depart Time" to be explicitly linked to the first **When**'s "Time" property, and "Arrive Time" to the second, so these relationships did not need to be inferred. (These links are not shown here.)

\* Thus, modeled components are not black boxes, only adjustable through data properties, but can be modified at any level in the hierarchy.

**Title component.** We are almost done. We have to put the title together:

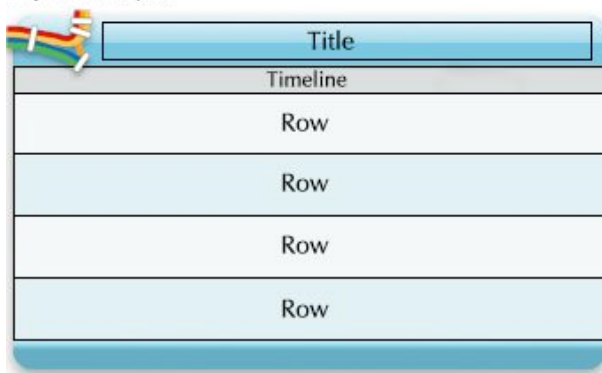


No inference is used here; we explicitly link the properties to the appropriate labels. (Again, not shown.)



**Trip Planner component.** Finally, we are ready to lay out the top-level component. We draw the background picture and place the components created above.

Trip Planner layout:



Our final graphic and its data properties look like so:

Trip Planner snapshot:



**From Station:** Embarcadero  
**To Station:** Rockridge  
**Fare:** \$2.90

**Trip 1:**  
**Depart Time:** 2:45 pm  
**Arrive Time:** 3:15 pm  
**Line:** Yellow  
**Destination:** Pittsburg

**Trip 2:**  
**Depart Time:** 3:00 pm  
**Arrive Time:** 3:30 pm  
**Line:** Yellow  
**Destination:** Pittsburg

**Trip 3:**  
**Depart Time:** 3:15 pm  
**Arrive Time:** 3:45 pm  
**Line:** Yellow  
**Destination:** Pittsburg

**Trip 4:**  
**Depart Time:** 3:30 pm  
**Arrive Time:** 4:00 pm  
**Line:** Yellow  
**Destination:** Pittsburg

No inference is used here. We explicitly link the top-level properties to the appropriate component properties.

Our dynamic graphic is complete. The final program would consist of this graphic and a data source that fills in the properties. Of course, this small example does not entirely emulate the actual BART widget, but it is easy to see how additional features can be added, simply with models and snapshots.\* It is also easy to see how a completely different design, such as the tables on the official BART website, could be composed on top of the exact same data source.

\* Except for animation, interaction, and state.

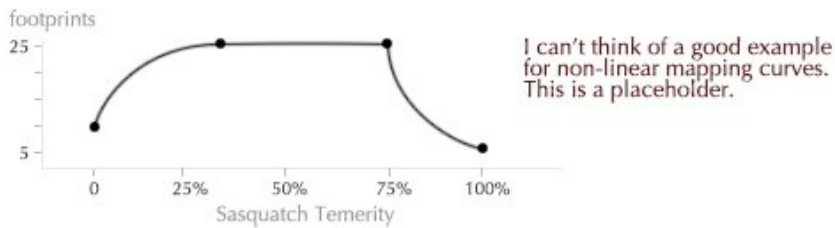
**Editing snapshots.** The essence of this process is elimination of abstraction. The designer works with concrete, visible examples.

However, this raises a concern about editing. An advantage of abstraction is that it localizes common properties, so widespread changes can be made with a single edit. What if the designer decides that a **Train** should have square corners instead of rounded? Having to individually edit each of the snapshots is unacceptable—such a burden would squelch experimentation.

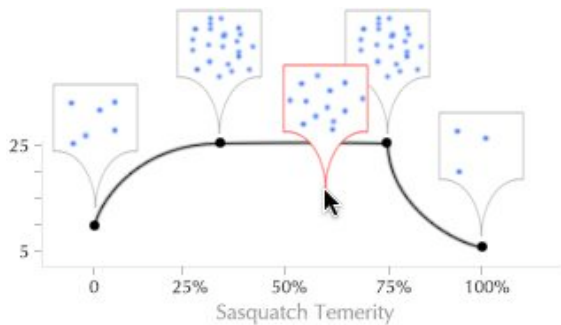


Instead, the designer simply *selects* the snapshots she wants changed, and proceeds to edit *one* of them. The changes propagate to all selected snapshots. This is possible because the tool treats the snapshots as variations on a single graphic, rather than independent graphics.

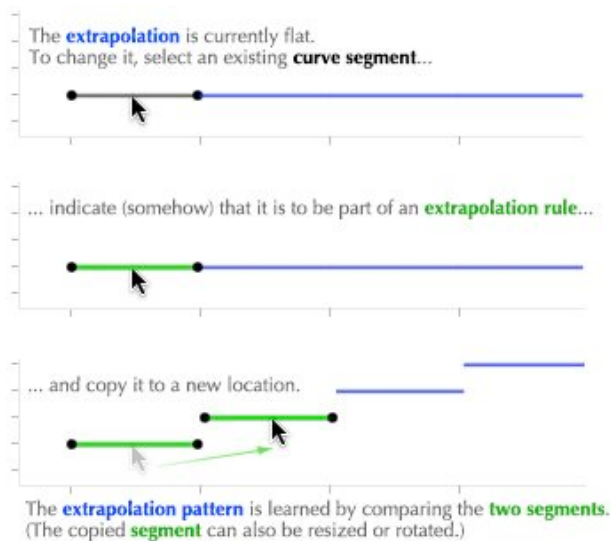
**Editing curves.** A more quantitatively-oriented designer may prefer to manipulate inferred relations directly. Mapping curves can be shown graphically, and the designer can move anchor points around, add new anchor points, and introduce curvature by stretching the interpolation curves. This allows for non-linear or nuanced behavior that would be difficult to specify purely with examples.



The curves are an abstraction, but because it is purely visual, designers may find it comfortable. To lessen the abstraction, abundant concrete examples from along the curve are shown, and a designer can point anywhere in the plane to see an example that corresponds to that point.



Curve editing may also allow for better control over extrapolation:



In the above, we are essentially using “drawing by example” to specify the extrapolation curve.

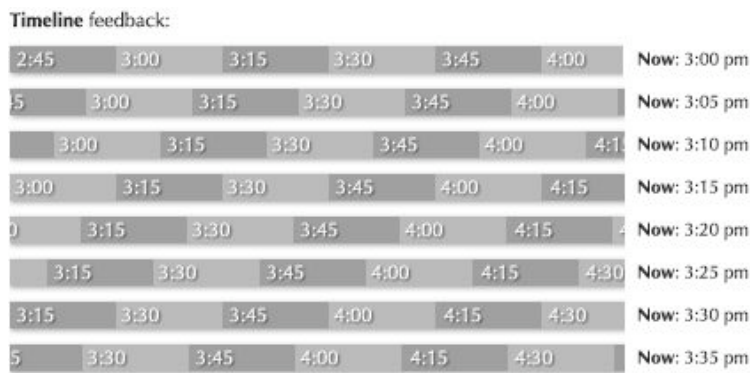
**Feedback through examples.** Conventional software engineers will be worried by the rampant ambiguity in this design process. In the demonstration above, the snapshots are visible but the inferred relations tying them together are not. How does a designer know if the tool’s understanding matches her own?

Unlike a programmer typing into a text editor, the designer does not create these snapshots in isolation. The tool provides a design environment that *actively* communicates the dynamics of the graphic.

One approach is for the tool to directly ask the designer about ambiguous cases. The tool can present the designer with a data set that would disambiguate an unclear relation, and the designer would then draw a snapshot for that particular data set. We might imagine the design process becoming inverted, driven by the tool—the designer would create a few representative examples, and then let the tool explicitly ask for all of the examples necessary to fill out the model.\*

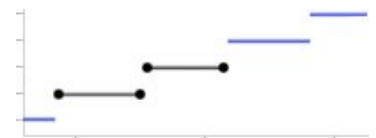
However, visual art has traditionally been composed actively, not reactively, and this approach may feel unnatural and stifling. A more natural and information-oriented approach uses *continuous peripheral feedback*. The tool adorns the screen with an array of its own examples that represent the inference it currently understands. As the designer works, she can visually verify that the inferred relations are correct:

\* There are a number of Programming By Demonstration research systems that take a similar interactive approach to disambiguation.

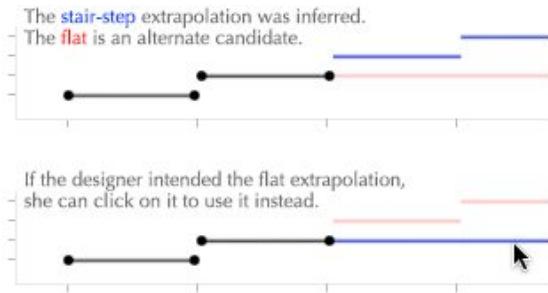


The tool can graphically emphasize feedback examples according to how little confidence it has in them. If one is incorrect, the designer creates a new snapshot simply by clicking on it and correcting it.

**Feedback through curves.** In addition to feedback through examples, the mapping curves described above also provide feedback. As the designer creates snapshots, she can see the inferred curves. If an inference is incorrect, she can either create more snapshots, or directly edit the curve (as long as the tool has correctly inferred which *variables* are involved in the mapping).



If the tool feels an extrapolation is ambiguous, it can display all of the candidate extrapolations on the curve, and the designer can select one with a click:

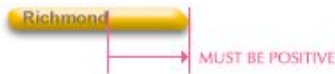


**Hints.** If necessary, the designer can add hints to encourage the tool to prefer certain inferences. There are two types of hints: dependencies and constraints.

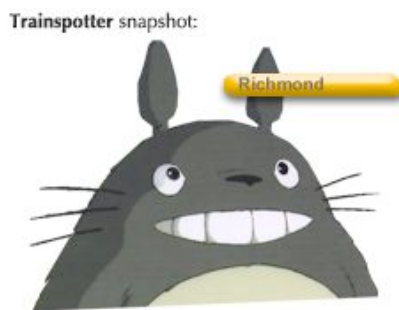
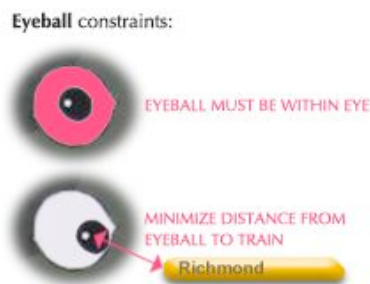
A *dependency* hint suggests that a particular graphical aspect is related to a particular data property. The specific mapping between the two must be specified through examples or curves, but this hint tells the tool which variables are involved.



A *constraint* hint suggests that a particular graphical aspect is related to some other graphical aspect.\* A “hard” constraint specifies an invariant relationship, such as two components that must always be aligned or parallel. In the example below, we ensure that the train cannot shrink smaller than the text label, by constraining the right edge of the train to lie to the right of the label:



A “soft” constraint specifies a *goal* that should be fulfilled as well as possible, given the other constraints. The example below models a character whose eyes will watch the train, wherever it goes. The eyeball is hard-constrained to lie within the eye, and soft-constrained to move as close as possible to the train.



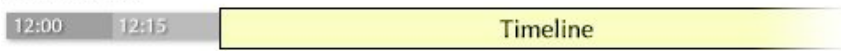
Hints may allow a designer to specify complex relationships that would be difficult to describe purely through snapshots.

**Recursion.** In the demonstration above, an infinitely-wide timeline bar was modeled by cyclicly panning over a finite graphic whose labels change on each cycle. This corresponds to the computational concept of iteration. An alternative for modeling infinite graphics is recursion. Consider this layout:\*

\* Drawing with constraints is as old as drawing on computers—both originated with Ivan Sutherland’s seminal *Sketchpad* (1963). Constrained drawing was further explored in a number of research projects (of note are David Kurlander’s *Chimera* (1991) and Michael Gleicher’s *Briar* (1993)) but has rarely appeared in general-purpose commercial tools.

\* The approach shown here is precisely how infinite data structures are represented in lazy

Timeline layout:



This **Timeline** component *contains* a **Timeline** component, shifted right by 100 pixels. The graphic now must be infinitely wide, because it is 100 pixels wider than itself. The tool draws this graphic by continuously “copying” the entire **Timeline** graphic, and “pasting” it into the yellow box:

How the tool expands the recursion:



With each paste, the yellow box shifts over by 100 pixels, and the pasting could go on forever. Now, we edit the text labels in the first paste (indicated by red arrows):



As we do so, the tool infers a linear relation between the top-level label (12:00) and the pasted one (12:30), and that relation is used to generate the labels in subsequent pastes (1:00, 1:30, etc.). That is, the tool learns to add a half hour each time it pastes. The result is a timeline whose labels increment forever.

For the final component, we need just two snapshots, to show how “Now” should pan across the graphic:

Timeline snapshots:



Some may claim that recursion is inappropriate for graphic designers. I would argue that recursion’s reputation for abstruseness is due to the textual abstractions used in mathematics and programming, and especially because expansion is rarely shown explicitly. I believe that, with training, any designer who appreciates MC Escher can learn to make powerful use of visual recursion.

**Insulation from engineering.** One of the primary benefits of this tool is the freedom it gives designers in composing the appearance of information software. The engineer’s job is to create a data source, and possibly spot-optimize the tool’s inferences if any are prohibitively

programming languages such as Haskell.

In fact, if we think of a component model as a function definition, and the placement of a component as a function call, this tool can be seen as an editor for an underlying functional “language.” Intriguing features include purity (evaluating a component has no side effects), laziness (components placed outside a clipping region need not be evaluated), and a combination of applicative evaluation (via mapping curves, whether explicit or inferred by the tool) and constraint-solving (via constraint hints). If we allow a component’s parameters (the function’s “arguments”) to themselves be dynamic graphics instead of merely text, and provide a means of graphically extracting part of a parameter and recursing on the rest, this language should be as expressive and powerful as any textual functional language.

inefficient. Unless complex behavior is necessary, the engineer is completely uninvolved with graphical presentation, to the relief of both designer and engineer.

For example, in the actual BART widget, the ending times become left-justified if the trip is too short:



Excellent software is characterized by this sort of attention to detail. However, if a designer were to *request* that this minor feature be implemented, she would probably be rebuffed by both engineer and management. This tool allows a perfectionist designer to add this feature on her own, just by taking a couple additional snapshots.\*

## Engineering inference from history

The section “Inferring context from history” presented the need for software to learn from the past. Good information software will attempt to predict current context by discovering patterns in past contexts. Although such application software is rare, there is nothing novel or exotic about the algorithms required. The computer science discipline devoted to this subject is called “**machine learning**” or “learning systems,” and several decades of research have produced a variety of algorithms for modeling and predicting behavior.\*

Consider the example presented earlier of a train trip planner that predicts the route that the user wants to see. There are typically daily or weekly patterns to a person’s local travel schedule. A planner that models these patterns could automatically present the user with appropriate information, eliminating most interaction.

As a demonstration, I implemented this behavior with the very simple probabilistic algorithm described below:\*

**History collection.** Each time the user indicates interest in a particular route, it is recorded in a history with the date and time. The user indicates interest either by explicitly switching the planner to display a route, or by looking at the planner and then looking away, indicating that the shown route is still interesting.

**Prediction.** When the user looks at the planner, each history entry “votes” for its route with a certain weight, and the route with the largest total weight is displayed. Each entry’s weight is a product of three factors, which depend on the time, the day of the week, and the age of the history entry.

- **Time.** If the time is 9:00, the user’s route at 9:10 yesterday is very relevant. What the user did at 10:00 is not quite so relevant, and her 12:00 activity is probably unrelated. Thus, each vote is weighted by a window around the time of the history entry.

Typically, if the user is preparing to catch a train, she won’t just look once at the planner and memorize the time. She will glance at the planner frequently over a span of

\* Implementation of the inferencing described here may involve algorithms that are unfamiliar to many engineers. Interested (or skeptical) engineers are encouraged to read the two machine learning books cited immediately below, as well as the two Programming By Demonstration books cited above.

\* Tom Mitchell’s book [Machine Learning](#) (1997) gives a good introductory overview to the basic algorithms. Russell and Norvig’s book [Artificial Intelligence: A Modern Approach](#) (2003) covers learning within a much broader context, but is less focused and concise.

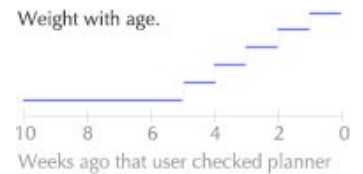
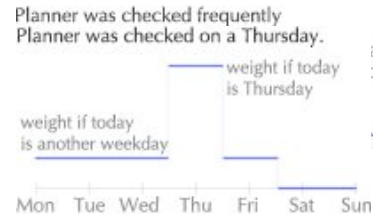
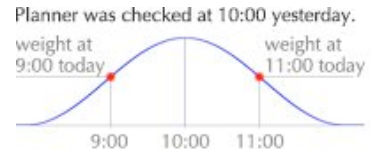
\* You can see the Lua [source code](#) for the algorithm and test simulator.

### HISTORY

Berkeley to San Francisco:	Mon, 8:25 am
San Francisco to Berkeley:	Mon, 5:05 pm
Berkeley to San Francisco:	Tue, 8:14 am
San Francisco to Palo Alto:	Tue, 1:25 pm
Palo Alto to San Francisco:	Tue, 3:06 pm
San Francisco to Berkeley:	Tue, 5:24 pm

time. Each of these looks should not count as an individual vote. Instead, the entire span of checking is coalesced into a single history entry, with a widened time window. (Also, throughout these frequent check-ups, the user sees only last-value prediction. Learning prediction is only used if some time has passed since the previous look.)

- Day of the week.** A user will typically exhibit a superposition of daily patterns, such as going to and from work, and weekly patterns, such as cello practice every Tuesday. To allow for both, history entries from a different weekday are allowed to vote, but have a smaller weight. The bleed across days allows the algorithm to learn daily patterns faster, but because other days are penalized, weekly patterns can be learned as well. Saturday and Sunday are independent from weekdays and from each other.
- Age.** Older history entries are given less weight, and eventually are forgotten. This makes the algorithm *adaptive*. If the user adopts a new pattern, such as switching jobs or joining the Thursday-night knitting circle, the algorithm is able to keep up, instead of having to be manually reset.



Finally, the most recent route is given a bonus vote. This causes the algorithm to default to last-value prediction if there is no compelling reason to do otherwise.

**Results.** I tested this algorithm with user models that simulate a variety of schedules. Various trade-offs are possible through choices of weights and window widths; the results below are intended to convey a qualitative idea of the algorithm's performance.

For a user who simply uses the planner to go to and from work, the algorithm learns the pattern flawlessly within a week. When the user switches schedules, the algorithm adapts within a couple weeks.\*

**SCHEDULE**

Mon-Fri 8:00 am: Berkeley to San Francisco  
 Mon-Fri 5:00 pm: San Francisco to Berkeley

**After 10 weeks, switches to:**

Mon-Fri 8:00 am: Berkeley to Oakland  
 Mon-Fri 5:00 pm: Oakland to Berkeley

**MISPREDICTIONS** per week, out of 10 predictions



More complicated schedules are also learned quickly and almost flawlessly.

**SCHEDULE**

Mon 8:00 am: Berkeley to Oakland  
 Mon 5:00 pm: Oakland to Berkeley  
 Tue-Fri 8:00 am: Berkeley to San Francisco  
 Thu 2:00 pm: San Francisco to Fremont  
 Thu 3:30 pm: Fremont to San Francisco  
 Tue-Fri 5:00 pm: San Francisco to Berkeley

**MISPREDICTIONS** per week, out of 12 predictions



Up to a certain level, random (unscheduled) behavior can be added without the algorithm losing the pattern.\*

\* Of course, humans won't check the planner at exactly the scheduled time, and neither does the model. The simulated times are normally distributed around the base time shown in the schedule, with a standard deviation of half an hour.

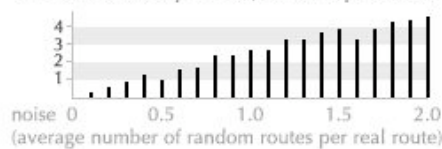
\* This graph plots mispredicted *scheduled* views. Obviously, the random views are always mispredicted.



#### SCHEDULE

Mon–Fri 8:00 am: Berkeley to San Francisco  
Mon–Fri 8:00 am: [random route]  
  
Mon–Fri 5:00 pm: San Francisco to Berkeley  
Mon–Fri 5:00 pm: [random route]

#### MISPREDICTIONS per week, out of 10 predictions



In conclusion, it appears that this algorithm would successfully be able to infer the context of a regular user, allowing relevant information to be presented with little or no interaction.

**So what?** As an ad-hoc solution to a particular problem, this algorithm seems to work quite well. **As a general solution, it is no solution at all.**

This simple, understandable example was intended to demonstrate that learning prediction is not science fiction—it is a viable and powerful approach to context inference, one that every software designer must keep on her palette. However, the best learning algorithms are considerably more complex than this one. Currently, machine learning is considered an experts-only area, where the fruits of research are restricted to specialists. Implementing learning behavior typically involves calling in an expert, not assigning it to the application programmer.

Unfortunately, an algorithm that can only be wielded by a master is almost worthless. There are far more applications than experts; if application programmers cannot make use of learning, learning applications will remain rare and exotic.

This predicament has been overcome many times before. All software today performs an intricate dance of feeding a processor primitive machine-level instructions, repolarizing tiny magnets in hard drives, transmitting bits reliably across wires, and lighting up specific pixels on a screen. The algorithms behind these operations are unimaginably complex, requiring years of study. Yet, even novice programmers have no trouble with these operations. The complexity has been hidden behind *abstractions*.

Programmers write to “files,” read from “sockets,” draw with “fonts” and “images,” and think in “programming languages.” Behind each abstraction are experts who devote their entire careers to their particular niche, following cutting-edge research and participating in the community. In front of the abstractions are armies of application programmers, blessedly able to take all this for granted. *Without these abstractions, our modern software landscape simply wouldn't exist.*

As I see it, the primary challenge for the machine learning community is not the generation and tuning of yet more algorithms, but the design of simple abstractions. Learning magic must be *packaged*. Like a “file,” the abstraction must be usable by *any* engineer working in *any* domain. It must be so simple that it can be taken for granted.

Today, a Perl programmer needs just four letters to invoke decades of research into filesystems and physical media: “open.” A finely-tuned mergesort is available with the word “sort,” and even more finely-tuned hashing algorithms require just a pair of brackets. **Until machine**

learning is as accessible and effortless as typing the word “learn,” it will never become widespread.

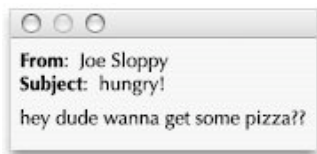
One small step for trip planners gets mankind nowhere.

## Engineering inference from the environment

The section “Inferring context from the environment” presented a number of environmental sources from which information software could infer context. The hardware-related sources, such as clocks and location sensors, might have seemed obvious. The software-related sources, such as other information software and documents created with manipulation software, might have seemed so far-fetched as to be implausible. This section will present the **information ecosystem**, a software architecture which might allow for such behavior.

Consider this scenario:

- I receive an email from a friend.



- After reading the email, I open my map software to find that nearby pizza restaurants are prominently marked.

How might such behavior be implemented?

One approach is to build a *system* that directly performs the desired behavior. In this case, perhaps one would design an email program with a built-in map. If the current email contains the word “pizza,” the program would perform an internet search for pizza places and display them on its map.

There are several reasons why the system-based approach is unappealing:

- Monolithic systems don’t scale. The system described is a trivial solution to a general problem. What about information from a website showing up on my calendar? What about seeing encyclopedia entries related to the paper I’m writing? The possibilities grow combinatorially—it is impossible to deliberately handle them all.
- Monolithic systems are bad for users. Email and maps are distinct concepts. There is no reason why a user should turn to the same software package for two unrelated purposes.\* Also, the components of integrated systems tend to be of lower quality than their dedicated counterparts. You could chop your vegetables and assemble your furniture with a Swiss Army knife, but you probably don’t.
- Monolithic systems are bad for software providers. In a healthy marketplace, whether of groceries or auto parts, individual providers offer components which combine with

**Other information software**, such as open websites. By reading some information, the user is indicating a topic of interest. All other information software should take heed. Consider a person reading the website of an upcoming stage play. When she opens her calendar, the available showings should be marked. When she opens a map, she should see directions to the playhouse. When she opens a restaurant guide, she should see listings nearby, and unless the play offers matinees, they shouldn’t be lunch joints.

**Documents created with manipulation software.** *Creating* some information indicates an even stronger topic of interest. Consider a person who requests information about “cats” while writing a paper. If the paper’s title is “Types and Treatment of Animal Cancer,” the information should skew toward feline medical data. The title “History of Egypt” indicates interest in ancient feline worship instead. And if the paper contains terms related to building construction, “cats” probably refers to the decidedly non-feline Caterpillar heavy machinery.

\* For that matter, email and calendars are distinct concepts as well.

others for a complete solution. A small software provider could provide an excellent email program, or an excellent map. But only a large corporation has the resources to develop an integrated package. Once small companies can't compete, progress stagnates.

What we need, then, is not a *system* that implements this behavior, but a *platform* that enables such a system to grow organically, via small contributions from diverse providers.

In forsaking integration, however, we forsake designed coordination between components. The email program and the map will be designed by two different software providers, oblivious to one another. The programs must somehow exchange information *without knowing anything about each other*—without even knowing the other exists.

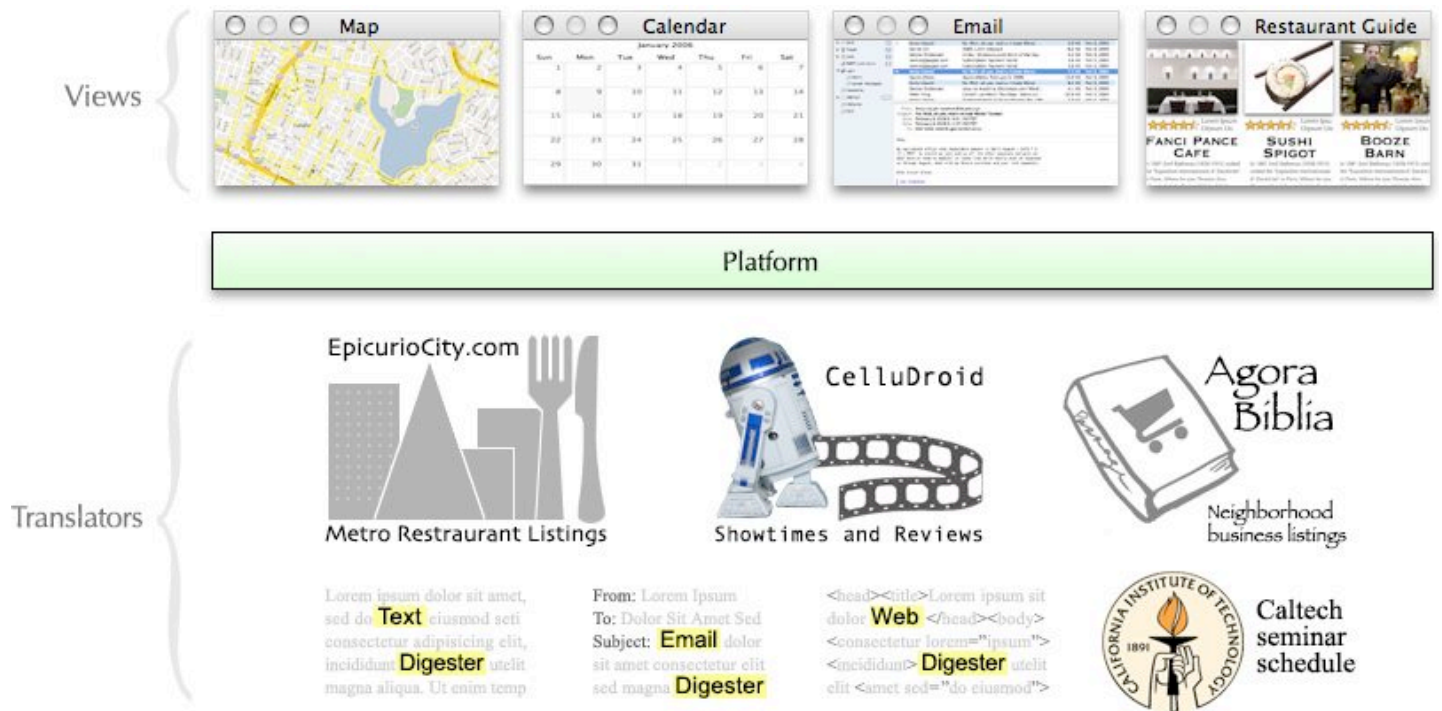
As it happens, such a mechanism has long existed for *manipulation* software—copy-and-paste. This mechanism uses the platform as an intermediary. When the user “copies” a picture in a drawing program, the program hands data off to the platform. When the user then “pastes” the picture into a word processing document, the program requests data from the platform, and handles it according to its type. The drawing and word processing programs know nothing of each other—they know only of the platform and standard data exchange formats.

Extending this concept to information software involves two additional concerns:

- **Autonomy.** As befitting manipulation software, copy-and-paste requires explicit manipulation by the user. Information software must be able to share information implicitly and autonomously, with no user interaction.
- **Translation.** An email is not a map location. Nor is a website a calendar event, nor a word processing document an encyclopedia entry. The information must be *translated* from one form to another.

Given that this platform exists to promote inference from the environment, let us take some inspiration from a *biological* environment. The very essence of a biological environment is autonomous translation. Plants translate sunlight into fruit, large animals translate fruit into dung, small animals translate dung into soil, plants translate soil into fruit. **An ecosystem is a network of individual components which consume nutrients and translate them to an enriched form consumed by others, autonomously and with no knowledge of the system as a whole.**

If we adopt this process in software, considering our “nutrients” to be information, we have an *information ecosystem*. Consider this system:



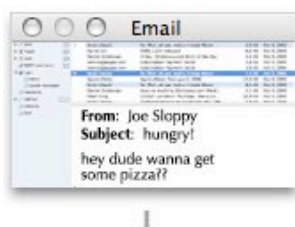
The components above the platform are **views**. This is the software that the user sees and interacts with. Views interact with the platform in two ways:

- Views *nominate* a **topic of interest**. For example, if the user is reading an email, she is probably interested in information related to the contents of the email. The email program would give the email to the platform as a topic. This is analogous to “copying,” but happens implicitly.
- Views *request* topics of interest, of some particular type. The map, for example, would request topics that have a geographical location associated with them. If a restaurant were a topic, the platform would give it to the map, and the map would display it. This is analogous to “pasting,” but again is implicit.

The components below the platform are **translators**. The platform gives them information objects, which they convert from one type to another and return to the platform. Sometimes this involves decomposing an object into constituent parts (“digesting” it); other times, it involves enriching the object with additional information.

The platform itself acts as an intermediary between components, attempting to fulfill requests by constructing a chain of translators to convert topics into the requested types.

This is how the behavior in the original example might come about:



I receive and read an email from my friend.

```
</email>
```

From: Lorem Ipsum  
To: Dolor Sit Amet Sed  
Subject: **Email** dolor  
sit amet consectetur elit  
sed magna **Digester**

The "email digester" breaks down the Email object into several other objects, one of which contains the body of the email as a Text object.

```
<text>  
<content> hey dude wanna get s  
...  
</text>
```

```
<person>  
<name> Joe Sloppy </name>  
<picture>  
 </picture>  
<favorite_smurf> Hefy </favorite  
...  
</person>
```

Lorem ipsum dolor sit amet,  
sed do **Text** eiusmod seti  
consectetur adipiscing elit,  
incididunt **Digester** utelit  
magna aliqua. Ut enim temp

The "text digester" translates Text objects into Keyword objects by looking for important words.

```
<keyword>  
<content> pizza </content>  
...  
</keyword>
```

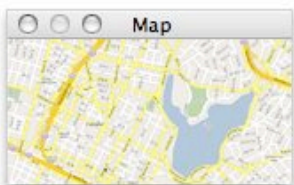
One of the words it finds is "pizza".



A component provided by EpicurioCity.com translates Keywords into Restaurants, if the keyword corresponds to a restaurant name or cuisine.

```
<restaurant>  
<name> Ark-Sign One </name>  
<tagline> "Have a pie!" </tagline>  
<cuisine> pizza </cuisine>  
<location> 2090 Kittredge St., Be  
...  
</restaurant>
```

It recognizes "pizza", and translates it into a Restaurant object.



The map view requests topics with a "location" property. The Restaurant has a location, so the platform gives it to the map to display.

Although it is clearer to visualize the process as described, an actual implementation would have to be *lazy*, driven by demand. That is, the process would start at the end with the map's request, and the platform would then construct the chain *back* toward the email program, according to the types and properties of the objects each component claims to consume and produce. This is necessary for efficiency reasons, but also explains how EpicurioCity knows the area to look for restaurants—the map actually requests objects with locations *around a particular area*, and EpicurioCity then attempts to produce objects that match this type. It also explains how EpicurioCity knows how many objects to produce from its almost infinite

in response to Joe's email. Now, consider what would happen if, instead of receiving an email, I were to type the word "pizza" into a document. Surely the last word typed would be nominated as a topic. It would then get picked up by EpicurioCity and translated into restaurants, and these would show up on the map. Thus, we have the remarkable emergent behavior of being able to look up pizza places simply by *typing the word "pizza" anywhere on the computer*.

This sort of emergent, non-designed behavior is the overall goal of the platform. Through topic nomination, the system models the user's immediate interests, and through translation, every view can attempt to serve these interests in any way possible.

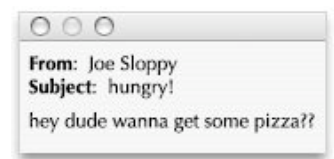
**Confidence.** At this point, the biggest problem concerns the question, "Just *what* should be a topic?" If every component nominated everything that could possibly be relevant, the map would become so cluttered as to be useless. The problem is addressed by recasting the question more fuzzily: "How *much* is something a topic?"

- Every topic is nominated with a **level of confidence**.<sup>\*</sup> An email that I'm reading *right now* would be nominated with high confidence. When I finish reading it and move on to something else, its confidence diminishes. The title of a paper I'm currently typing in would have high confidence; the title of a paper I haven't touched for a few minutes is lower. The paragraph that I'm working on has higher confidence than surrounding paragraphs.
- Translators produce **dilution of confidence**. As they translate, they multiply the object's confidence level by their own confidence in the translation. The text digester will have more confidence in unusual, prominent words, and words that seem related to other topics or the context in which topics were found. Partial or tenuous matches can be translated with low confidence instead of omitted. For example, a movie translator from CelluDroid.com might translate "pizza" into the film "Pizza Cato: The String Cheese Connection," but because the name is only partially matched and the context of the consumed object contains no references to movies, the confidence would be low.
- Views, such as the map, use confidence to determine the **graphical emphasis** of displayed objects. This is a critical part of the graphic design. Beyond simply adjusting size, emphasis can involve all sorts of standard graphical techniques—contrast, color saturation, shading, shadowing, grouping, or placement on a different graphic entirely. Objects with confidence below some threshold will probably be discarded.

As events cause confidence levels to change, the changes propagate through the chain of translators, adjusting the emphasis of displayed objects.

**Feedback.** The next problem with this system is inappropriate translations. Consider again my friend's email. The text digester might pick out the word "dude," which would go through the business listings at AgoraBiblia.com, resulting in the neighborhood dude ranch showing up on my map. This would be a nuisance if it occurred every time I received an email from my friend.

\* I will consider confidence levels to lie between 0 and 1, so that multiplication makes sense.





The problem is addressed through **backpropagation of feedback**. Feedback can be either explicit or implicit. Explicitly, I can indicate to the map that I am uninterested in dude ranching. This negative feedback is returned to the AgoraBiblia.com translator, resulting in low confidence in future dude ranch matches. The feedback may even propagate back to the text digester, slightly lowering the confidence that the word “dude” indicates a topic of interest. Implicitly, simply looking at the map without indicating interest in the dude ranch will cause a slight negative feedback, resulting in its de-emphasis over time. On the other hand, if I frequently click on pizza places, positive feedback will backpropagate through the chain of translators, increasing confidence in all things pizza-related and resulting in their emphasis on the map.

In effect, the entire environment becomes a learning system, tailoring itself to the individual user. While topics model the user’s immediate interests, the history acquired through feedback allows the system to model the user’s long-term characteristics.

**Protocol.** The last problem I will consider here is the political issue of protocol creation. Just what is a Restaurant object, and who decides that? Standards, especially premature ones, stifle invention and progress, but anarchy results in incompatibility. It may be possible to address this problem through namespacing and published proprietary protocols.

To answer the above question, there is no Restaurant object. Instead, EpicurioCity produces a `com.EpicurioCity.Restaurant` object,\* whose protocol is defined and managed by EpicurioCity.com. This proprietary object can be composed of other proprietary objects, as well as some standard objects defined by the platform, such as `Text`, `Keyword`, and `Location`. Note that this proprietary Restaurant is not hindered from showing up on the map, since the map will accept anything with a `Location` (and presumably some other standard properties such as a name and description).\* A restaurant guide view, on the other hand, would be written to take advantage of the extra information that `com.EpicurioCity.Restaurant` offers—ratings, reviews, and such.

When another provider, CuisineCousins.com, develops a competing restaurant translator, it can follow EpicurioCity’s published protocol and produce `com.EpicurioCity.Restaurant` objects. This makes their new translator immediately compatible with existing views. Meanwhile, the translator can *simultaneously* offer their own objects, such as a `com.CuisineCousins.Eatery`, with whatever advantages over EpicurioCity’s protocol. View providers can then update their software to also accept CuisineCousins’s protocol, if CuisineCousins offers a compelling enough advantage.

If a de facto standard emerges and stabilizes, it might eventually get canonized as the official Restaurant object. Even then, though, providers will be able to add proprietary namespaced extensions to it.

**Modularity.** An obvious benefit to this platform is that it enforces modularity between data and views. Unlike current systems, in which almost all data and functionality is locked up behind a user interface, every service on this system is available to every view. More subtly but

\* Or however namespacing is spelled in the implementation language.

\* In object-oriented terminology, `com.EpicurioCity.Restaurant` conforms to the Mappable interface, and the map requests Mappable objects. However, this “interface” can be very informal, and even unknown to the Restaurant. If the Restaurant happens to define enough standard properties, it can be mapped.

just as importantly, the fact that translators have no end-user interface means they can be created by engineers. Only the views must be designed for users. Meanwhile, a designer who is dissatisfied with a view can simply create and release a replacement, with no engineering worries about data acquisition. Because the system can be easily improved without cross-disciplinary concerns, creativity and invention should flourish.

## Information and the world of tomorrow

Today's ubiquitous GUI has its roots in Doug Engelbart's groundshattering research in the mid-'60s. The concepts he invented were further developed at Xerox PARC in the '70s, and successfully commercialized in the Apple Macintosh in the early '80s, whereupon they essentially *froze*. Twenty years later, despite thousand-fold improvements along every technological dimension, the concepts behind today's interfaces are almost *identical* to those in the initial Mac. Similar stories abound. For example, a telephone that could be "dialed" with a string of digits was the hot new thing ninety years ago. Today, the "phone number" is ubiquitous and entrenched, despite countless revolutions in underlying technology. Culture changes much more slowly than technological capability.\*

The lesson is that, even today, we are designing for tomorrow's technology. Cultural inertia will carry today's design choices to whatever technology comes next. In a world where science can outpace science fiction, predicting future technology can be a Nostradamean challenge, but the responsible designer has no choice. A successful design will outlive the world it was designed for.

With what artifact will the people of tomorrow learn information? I believe that in order for a personal information device to be viable in the long term, it must satisfy two conflicting criteria: portability and readability.

- **Portability.** Consider today's ubiquitous information device—the book. We have the technology to manufacture 5000-page desk-sized tomes, but despite the high information content, such books are rare. The reason is simply that they can't be carried around. As people increasingly expect information on demand, portability will become ever more critical. Today, people can talk to anyone on the planet by reaching into a pocket; tomorrow's information device must be just as accessible. Like a wallet and keys, the computer will be dropped into the pocket or purse before leaving the house.\* This implies light weight and small volume.
- **Readability.** Consider again the book. We have the technology to produce books smaller than a business card, but despite the improved portability, such books are also rare. The supremely-portable postage-stamp-sized book is non-existent. The catch: **Although technology miniaturizes, the human eyespan remains a fundamental constant.** In order to compete with the book, tomorrow's information device must provide a book-sized surface area. Anything less cannot be read and skimmed comfortably, and cannot support spatially-distributed information graphics.

\* Other obsolete but entrenched designs: the QWERTY key layout (intentionally sub-optimal to reduce typewriter jams), the von Neumann architecture (see John Backus, [Can Programming Be Liberated from the von Neumann Style?](#), 1978); C and UNIX (see Richard Gabriel, [The Rise of "Worse is Better"](#), 1991).

\* Ideally, it will even supplant both wallet and keys.

To resolve these contrasting size constraints, I predict a computer the size and thickness of a sheet of paper. Like paper, its entire surface is a graphical display. When in use, it is rigid; when not in use, it collapses and can be folded or rolled up (or crumpled!) and tucked into a pocket or purse.

Regardless of whether I've guessed its form accurately, we can predict the device's expected characteristics by extrapolating technological trends. Consider the capabilities relevant to context-sensitive information graphics: graphical output, history, environment, and user interaction.

- **Graphical output.** To serve as a book, the device must have a sufficiently large reading area and high pixel resolution. To serve as a computer, the device must produce dynamic color graphics. In matching each of today's devices, tomorrow's device will overcome the shortcomings of the other. Dynamic graphics with print resolution will open up a world of possibilities for detailed information graphics which are impossible today in either medium.
- **Environment.** Because the user will carry this device everywhere, the device's environment will literally be the user's own. Assuming a sufficient networking model, the device will be able to sense an enormous amount of information from the environment—geographical location, physical surroundings (streets, stores, transportation options, entertainment options), social surroundings (friends, strangers with interests in common, strangers who can serve a need), and more. The device will have a far better sense of the user's environment than the user herself.
- **History.** Since its inception, electronic storage has exponentially increased in density and decreased in cost. We can fully expect tomorrow's device to have onboard capacities that stagger modern sensibilities. But, perhaps more importantly, ubiquitous network access will make memory effectively *unlimited*. The device will have the means to remember everything the user has ever done and every environment in which she did it. With such a tremendous history and sense of the environment, software will have an unprecedented potential to predict the user's current context.
- **Interaction.** Touch or motion-based manipulation is somewhat more efficient than the mouse. Eye-tracking and speech may be better still, although even these are unlikely to match the order-of-magnitude improvements predicted for the capabilities above. But none of these mechanisms will ever approach the sheer amount of information that can be absorbed by the eye. No matter what new interactive technology comes along, the bandwidth between the device and the user will remain not merely asymmetric, but utterly lopsided.

Interaction is already a bottleneck. It will get much worse as graphics, environment, and history experience their expected breakthroughs. To me, the implication is clear—the principles of information software and context-sensitive information graphics will become *critical* as technology improves.

The future will be context-sensitive. The future will not be interactive.

Are we preparing for this future? I look around, and see a generation of bright, inventive designers wasting their lives shoehorning obsolete interaction models onto crippled, impotent platforms. I see a generation of engineers wasting their lives mastering the carelessly-designed nuances of these dead-end platforms, and carelessly adding more. I see a generation of users wasting their lives pointing, clicking, dragging, typing, as gigahertz processors spin idly and gigabyte memories remember nothing. I see machines, machines, *machines*.

I expect that designers who cling to these models will appear to the next generation like classical physicists as the world turned quantum, like epicycle-plotters as Kepler drew ellipses, like Aristotelians as Galileo stood atop the tower at Pisa. No matter how hard they work or how much they invent, these designers will not be revered as pioneers. They are blazing trails through a parking lot.

Our pioneers are those who *transcend* interaction—designers whose creations *anticipate*, not *obey*. The hero of tomorrow is not the next Steve Wozniak, but the next William Playfair. An artist who redefines how people learn. An artist who paints with magic ink.



## Summary

Software design consists of graphic design (drawing pictures) and industrial design (allowing for mechanical manipulation).

Information software is for learning an internal model. Manipulation software is for creating an external model. Communication software is for communicating a shared model.

Manipulation software design is hard, but most software is information software.

Information software design is the design of context-sensitive information graphics.

Information software is not a machine, but a medium for visual communication.

Context can be inferred from the environment, which can include physical sensors, other information software, documents created with manipulation software, and data such as email which acts as a user profile.

Context can be inferred from a history of past environments and interactions. Last-value predictors provide a rudimentary approach. Learning predictors can infer patterns and make dynamic predictions.

Context can be inferred from user interaction, but only as a last resort. The best way to reduce or eliminate interaction is through information-rich graphic design that uses the environment and history. Remaining interaction can be reduced with graphical manipulation, relative navigation, and tight feedback loops.

The information software revolution will require public recognition that information software is a medium of visual communication, designers with talent, skill, and tools, simple and general platforms, and an environment that encourages creativity and sharing.

A design tool for dynamic graphics that infers behavior from mockups may allow for natural-feeling creative design with no engineering-related distractions.

Learning predictors exist and are effective. For them to become widespread, simple abstractions must be invented.

An information ecosystem of views and translators may be able to offer relevant information of all forms with minimal interaction. Key aspects include topic nomination and translation, confidence levels, learning through feedback, and a fine-grained modular structure wherein small software providers can thrive.

As technology related to graphics, the environment, and history undergoes revolutionary improvements, interaction will become even more of a critical bottleneck. The best approach is to work towards eliminating it.

Two centuries ago, Playfair invented statistical graphics and changed the world. The time is ripe for another designer to invent the fundamental context-sensitive graphical forms, and change the world again.

## Recommended reading

Most of the works cited in this paper are recommended—just skim up the sidenotes. The following landmark books deserve special mention:

Edward Tufte. [The Visual Display of Quantitative Information](#) (2001), [Envisioning Information](#) (1990), [Visual Explanations](#) (1997). The three testaments of the information design bible. (A [fourth](#) is on the way.) If you've already read them, read them again.

Scott McCloud. [Understanding Comics](#) (1994). Like information graphics, comics convey information through arrangements of words and pictures—they are Tufte's "small multiples" applied to storytelling. McCloud's analysis of how people read and understand the visual language of comics is essential reading for all information graphic designers.

Thomas Kuhn. [The Structure of Scientific Revolutions](#) (1962). The only purpose of incremental improvement to a status quo is to reveal its flaws. Progress occurs when the status quo is *replaced*.

## Acknowledgments

I am grateful for the detailed and helpful feedback I received on earlier drafts from Andy Likuski, Jonathan Harel, Justin McCarthy, Jon Nakasone, and [Daniel Cook](#).

I also appreciate everyone who sent in feedback on the BART widget, and particularly [Joel Dreisbach-Penner](#), [Walter Jew](#), and [Omid Tavallai](#) for their contributions. This paper sprung from that little widget.

This paper was formatted with a modified version of [John Gruber's Markdown](#) utility. If you like, you can see the [modified Markdown](#) and the paper's [source code](#).

Half of this paper was written in the San Leandro and Dublin public libraries.\* The other half was written in Cordonices Park and Live Oak Park in Berkeley.

\* Have you been to your public library? It's like Starbucks, but free of charge, noise, and corporate branding.

## About the author

My most recent creation is [ClickShirt](#), a remarkable online T-shirt designer.

Prior to that, I designed, hardware-engineered, and software-engineered the award-winning [Alesis Micron](#) synth keyboard and groovebox. I also engineered the critically-acclaimed synthesis engines of the Alesis [Ion](#) and [Fusion](#) keyboards. Lots of older projects can be found at my [website](#).

My academic background is in electrical engineering (BSEE Caltech, MSEE UC Berkeley), but I've been creating software continuously from the age of seven. An explicit interest in design is, unfortunately, a recent revelation.

I believe that mankind's ability to avert the global catastrophes of the day will hinge on the design of excellent information software (so scientists can see the problems) and manipulation software (so engineers can create solutions). Accordingly, I'm interested in coming up with new models and tools for designing this software and presenting this information.

I am currently attempting to design a visual language for drawing dynamic graphics. (It is inspired by, but only loosely based on, the design tool discussed above.)

I actually have no interest whatsoever in the design of online retailers. It just makes for good examples.

## Epilogue

*In the early days, I was solving one problem after another after another; a fair number were successful and there were a few failures. I went home one*



Friday after finishing a problem, and curiously enough I wasn't happy; I was depressed. I could see life being a long sequence of one problem after another after another. After quite a while of thinking I decided, "No, I should be in the mass production of a variable product. I should be concerned with all of next year's problems, not just the one in front of my face." By changing the question I still got the same kind of results or better, but I changed things and did important work. I attacked the major problem—How do I conquer machines and do all of next year's problems when I don't know what they are going to be? How do I prepare for it? How do I do this one so I'll be on top of it? How do I obey Newton's rule? He said, "If I have seen further than others, it is because I've stood on the shoulders of giants." These days we stand on each other's feet!

You should do your job in such a fashion that others can build on top of it, so they will indeed say, "Yes, I've stood on so and so's shoulders and I saw further." The essence of science is cumulative. By changing a problem slightly you can often do great work rather than merely good work. Instead of attacking isolated problems, *I made the resolution that I would never again solve an isolated problem except as characteristic of a class.*

—Richard Hamming, *You and Your Research* (1986)

Before I release v1.0 of the BART widget, I'd like to write a little paper about its design...

—Bret Victor (2005)

## Epilogue 2

When I first prepared this particular talk ... I realized that my usual approach is usually critical. That is, a lot of the things that I do, that most people do, are because they hate something somebody else has done, or they hate that something hasn't been done. And I realized that informed criticism has completely been done in by the web. Because the web has produced so much uninformed criticism. It's kind of a Gresham's Law—bad money drives the good money out of circulation. Bad criticism drives good criticism out of circulation. You just can't criticize anything.

—Alan Kay, *How Simply and Understandably Could The "Personal Computing Experience" Be Programmed?* (2006)