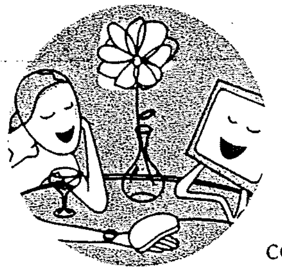
Chris Crawford,
The Art of
Interactive
Design

7 ARCHITECTURES

re information
ormation flowing



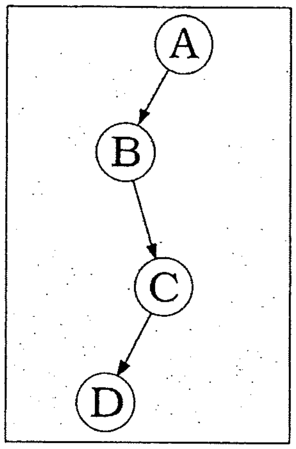
Architectural diagrams of interaction help us better understand the design problems we face. A variety of commonly used strategies are available, but most suffer from some serious defect. A bushy tree is required for good interactivity. One way to judge the interactive quality of a design is to examine the ratio of accessible states to conceivable states.

One of my physics professors used to say, "When in doubt, draw a picture." Any tough problem can be clarified by some sort of diagram laying out the elements of the problem and their relationships. For many people, myself included, drawing a picture serves the same purpose that vocalizing a feeling has for many people when they're upset: it may not by itself solve the problem, but it's a good start.

Interactivity design presents us with many tricky problems, and it can be especially difficult to articulate those problems because of the dynamic nature of interactivity. Accordingly, drawing a picture might have some value, not as an engineering diagram or a blueprint, but rather as a way to visualize the problem. I hijacked a standard diagramming scheme from computer science and modified it slightly to apply to problems of interactivity. It's ridiculously simple, but that simplicity makes it applicable to a wide variety of design problems. In this chapter, I'll present the diagram and show how it illustrates many of the problems and mistakes of interactivity design. Although I'll be using games and interactive storytelling as my primary examples, the principles apply to any kind of interactive application, as I shall later illustrate. Moreover, the design problems show up most clearly in games.

A Simple Interactivity Diagram

I shall begin with a simple diagram that presents the structure of a story:


$$\begin{tikzcd}
A \arrow[d] \\
B \arrow[d] \\
C \arrow[d] \\
D
\end{tikzcd}$$

In this diagram, each circle represents an event or action, while the arrows show the connections between the events. A story is thus a simple sequence of events:

A. Once upon a time there was a noble youth named Culhych.

B. His stepmother told him that he must win a beautiful girl named Olwen.

C. But to do this he had to gain the help of a great king.

D. So Culhych set out for Arthur's court. . . .

Note that the sequence of events is linear; that's why we refer to this structure as a *storyline*.

Now, how can we evolve this structure into something that is interactive? Some people don't bother to ask this question; they just take a story and cram it into a computer, add a few meaningless technological tricks, and pronounce it "interactive." I shall dismiss such travesties without further exercise.

For the purposes of this chapter, we need to focus on a single critical component of interaction: the element of choice on the part of the user. The user gets to make decisions, to effect choices. If the user doesn't get to make a choice, we don't have interaction; we have a plain old story. A choice would be expressed in our little diagrammatic scheme with something like this:

ısing games and
ıpplv to any kind
e ʒn prob-

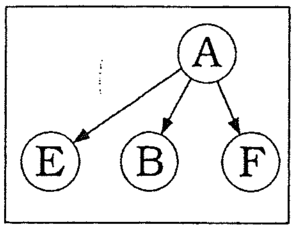
of a story:

. while the arrows
nple sequence of

:h.
l named Olwen.

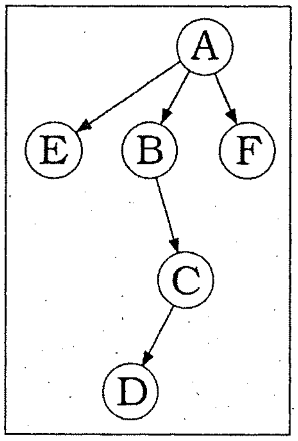
refer to this struc-

at is interactive?
a story and cram it
and pronounce it
xercise.
single critical com-
:he user. The user
ʒet to make a
A choice would be
; like this:

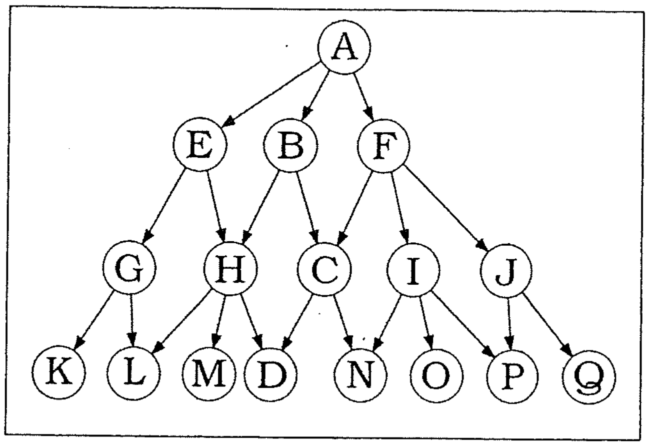


This is called a *branchpoint*, a term taken from programming. The user comes to a point in the story where she gets to make a decision. She has several options: that is, the story can proceed in any of several directions. The user decides which option to take. The important fact is that there is more than one choice–the user has a meaningful choice that will influence the future direction of the interactive story.

Now, how do we put this structure into our storyline? Well, the obvious thing to do is to simply cram it into the storyline structure like so:

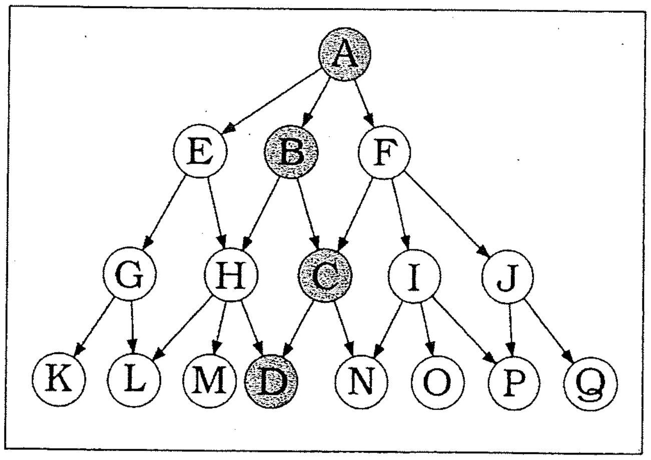


Of course, this creates a new problem: what falls below the empty nodes E and F? The obvious thing to do now is to continue the process of giving the user options by tacking additional branchpoints onto the structure. This yields a *storytree*:



Now I'd like to ask, how many layers should the structure have? That is, how many rows should there be in the pyramidal structure? I have drawn just four layers, but what would a real interactive storyteller have?

A quick way to answer this is to look again at the storytree with one portion highlighted:



The highlighted portion, you will note, is exactly the same as the storyline at the beginning of this example. In other words, a storytree is a storyline creator; a single path through a storytree yields a storyline. We can therefore apply what we know about stories to estimate the appropriate depth of a storytree. A movie, for example, will have many more than ten events or actions in it, and certainly less than 10,000. In other words, I think that a movie has something like a hundred or a thousand events or actions in it. Novels tend to be longer, perhaps.

Let's be conservative so we get the smallest reasonable number of nodes. Let's assume that our interactive story-thing needs only a hundred events or actions. In other words, there will be 100 layers in our interactive story. Let us

ıave? That is, how
lrawn just four

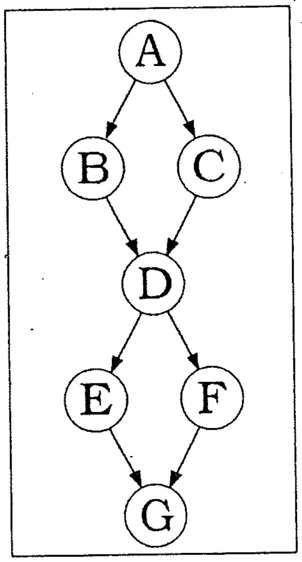
: with one portion

e as the storyline at
ı storyline creator;
erefore apply what
storytree. A movie,
in it, and certainly
ıething like a hun-
longer, perhaps.
ımber of nodes.
ıdred events or
ctive story. Let us

further assume that each branchpoint will have only two choices available to it—this is the absolute minimum required. This means (according to a standard calculation) that the storytree will have a total of 2^{100} nodes in it. How many is that? About 10^{30}. And how big is that? If you employed every human being on this planet to create nodes, each person making one node every second, working 24 hours per day, 365 days per year, then it would take 5 trillion years to make the nodes necessary to build that single storytree. You're going to have difficulty making your deadline.

Foldback

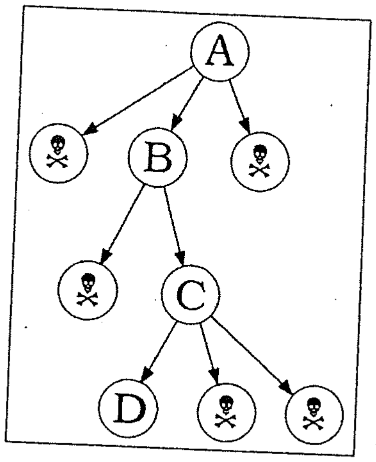
A variety of solutions to this problem of the geometric growth of nodes have been developed over the years. One of the earliest solutions, first presented 30 years ago, had a structure like this:



I call this trick *foldback*. The storyline folds back on itself. For example, consider the first four nodes, labeled A, B, C, and D. Suppose that node A represents Eloise saying to Bart, "I'm just not attracted to wimps like you, Bart." Suppose further that node B represents Bart wringing his hands and pleading with Eloise, while node C has him sneering at her, "Ha! I can find better sex in a broom closet!" Now, node D might represent Eloise declaring, "Oh, Bart, don't say that—I'll take you back!"—which response is appropriate to either of Bart's actions. It's clever, but the problem with this method is that it robs the interactivity of any meaning. Whatever Bart does, Eloise is going to take him back. Indeed, if you step back from this diagram a few feet, it doesn't look any different from a regular storyline. This is nothing more than a storyline masquerading as a storytree. This is fraudulent interactivity.

Kill 'Em If They Stray

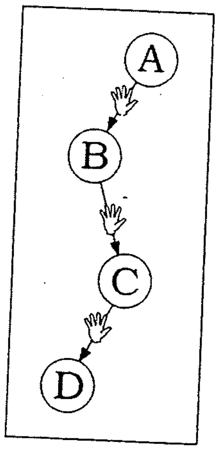
Here's another approach:



I call this the *kill 'em if they stray* approach to storytree design. The designer allows the user many options, but almost all of the options lead to the death of the user or the termination of the story. This scheme merely spruces up a linear storyline with many opportunities to fail along the way. This is not what I call user-friendly. Moreover, it's still essentially linear. Many adventure games and puzzle stories fit this model.

Obstructionist Stories

Another variation on this is the *obstructionist* scheme wherein the user is presented with a linear sequence of story nodes separated by obstacles:



n. The designer
to the death of
ɾuces up a linear
ıot what I call
ɾe games and

e user is pre-
:les:

The user runs into a brick wall at each stage in the story and must solve the puzzle to proceed to the next stage of the story. This is interactive after a fashion; the user is permitted to interact with the puzzle. But, of course, there's no interaction with the story itself. This is Skinner Box interactivity, reducing the user to the status of a rat in a maze, required to push the correct levers to get the reward—and punished for failure. Despite its serious flaws, the technique has been used in a number of commercially successful products. I consider it to have the staying power of any other fad.

Hand-Wired Storytree

Next comes the *hand-wired storytree*, too messy to diagram. The designer sits down and draws a big diagram showing how every node connects with every other node. He builds the entire structure by hand, plotting connections and figuring pathways. This sounds better; after all, it's hand tooled and surely must be finely tuned. The problem with such designs is that they are sorely limited by the amount of time that the designer can put into them. Even the largest such designs sport no more than a few hundred nodes. Moreover, these are often crippled in other ways. They support only the most primitive of Boolean connection schemes. In other words, the pathways in the system are opened or closed by the simplest of yes-or-no decision schemes involving such lame-brained factors as whether the user uses the correct item, whether he recites the magic incantation, or some other such simple-minded pap.

Combinations of the Above

Then there are the combinatorial schemes, which combine various elements of the other architectures. A bewildering array of such methods have been tried. One such scheme attempts to solve the problem of the gigantic storytree by linking together a group of smaller, more manageable storytrees. The links between the small storytrees are simple, direct connections.

This is a common approach in graphic adventures. Each small storytree is used to establish a particular subgoal. The first storytree might determine whether you wheedle the orange key from the one-eyed pirate. With the orange key, you can enter the secret room where the purple dragon awaits; now you enter a new storytree whose outcome, if successful, advances you to the next storytree.

This system fails because each subtree produces a simple success-or-failure result. This means that we can replace each of the small storytrees with a single node that asks the question, "Did the user succeed at the assigned task?" It operates in exactly the same way that the obstructionist story operates.

In general, combinatorial schemes fail because there are no exploitable synergies between the various strategies. Simple combination of two dissimilar components seldom accomplishes anything; there must exist some relationship between the two that permits a useful synergy to emerge from the combination. A metal alloy works because the atoms of the two metals are different sizes and will fit together in a tighter, stronger structure. Cramming random bits of metal together doesn't make better metal.

The Desirability of Bushiness

A good storytree is rich and bushy; a storytree that is narrow or scraggly is not particularly interactive. Why? There are two ways to answer this question.

One way is to think in terms of user choice. The user wants to make choices. Choice is the means by which we express our free will; choice is the manifestation of our personalities. Hence, a good design offers the user many choices—many branches emerging from each branchpoint.

Another way to look at this problem is to think of the interaction as an act of individuation. When our user enters our storytree, she desires to find her own personal resolution to the challenge presented in the interaction. This could be her own document or even her own spreadsheet. This is fundamental to human nature. In anything we attempt, we seek not to replicate the results of the masters, but to create our own unique solution. What cook fails to make some small individuating adjustments in the recipe in the cookbook? It is an act of self-expression, of asserting our individuality.

What do we do to the users of our interactive entertainment? We cram them into a slot, demanding that they follow *our* storyline. We narrow their options, declaring some options to be correct and others to be incorrect. We give them only a few choices, because we are too lazy to recognize just how varied our users are.

If we wish to offer our user a truly satisfying interactive experience, it is imperative that we allow each user to express her individuality during the experience. There should be billions of pathways through the storytree, so that each user can find her own path through, coming to her own conclusion—and it's no fair declaring that only a handful of such pathways are correct. The average user should be able to make her own choices and still find a pathway with a satisfying conclusion. The user of a word processor wins or loses according to her own writing standards, not those of the designer. Thus, we want to create a storytree that is thick and bushy, with billions of pathways leading through it. This is the only way to guarantee that each user will be able to find an individuating pathway through the storytree.

Broader Applications

These concepts can be applied to design areas other than games or interactive storytelling. To do so, I must engage you in a long-winded and roundabout reasoning process. It begins with another one of my simple diagrams:

the combination.
ifferent sizes and
dom bits of metal

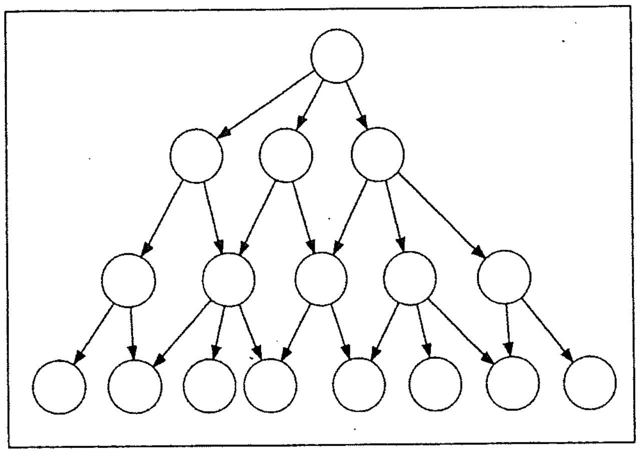
r scraggly is not
s question.
:s to make
choice is the
: the user many

raction as an act
es to find her
·action. This
: is fundamental
:ate the results of
fails to make
book? It is an act

nt? We cram
narrow their
incorrect. We
i' st how var-

)erience, it is
during the expe-
ree, so that each
.sion—and it's no
The average user
y with a satisfying
ıg to her own
:reate a storytree
;h it. This is the
ividuating path-

s or interactive
roundabout rea-
:ns:



By the way, we need a shorthand label for these diagrams. Computer scientists use the term *graph* for something similar to this, and some people prefer to use *state diagram*, but each of these is used in somewhat different ways than we need. To avoid confusion, we should use a distinct term, and I suppose that this burden falls upon my already fatigued shoulders; I therefore propose the wildly immodest term *Crawford diagram* to refer to them. At least it's better than *dactylodeiktous diagram.*

This tree structure could just as well be applied to a word-processing program, the only difference being that, instead of two or three branches at each node, we would see about a hundred branches: one for each key on the keyboard. The top node is a blank page. If this diagram were applied to a website, then the top node would be the home page.

To use the word processor, the user starts at the top of the tree with a blank page and selects a letter to type, which creates a new state, a document with that single letter. This act also moves him down one layer in the tree. He then selects another letter to type, which takes him to a new state and the next lower layer, and so on until he reaches the bottom of the tree and his document is complete. The bottom of the tree contains zillions of nodes, each representing a document containing characters in some sequence. The great majority of these imaginary documents would be nonsensical collections of characters, but one in a zillion might be a Shakespearean sonnet. Thus, in this way of thinking, the user does not write a document so much as he chooses a document through a long sequence of small choices (keystrokes).

Differences between Word Processors and Games: Two New Concepts

But there is a big difference between word processors and games: the former do their job better than games. My word processor does everything that I want to do, and then some. By contrast, I have yet to play a game that gave me such a sense of satisfaction. Every game I have ever played restricted my freedom of action, refused to permit me to do the things that I wanted to do. How can it be that games and word processors can yield such different results when they are structurally identical? What is it that word processors do that games don't do? The answer to this question will tell us how to make any interactive application—including word processors—better.

To answer that question, I shall introduce two new concepts. The first is the set of *accessible* states in a tree. These are all of the states that the user can get to as he moves down the tree. In a word processor, this is the humongous set of all documents that the user could theoretically create; in a game, it is the set of all realizable paths through the game; in a storytree, it is all the different stories that could be experienced.

The second concept is a little more difficult; it is the set of all *conceivable* states. These are the states that the user might expect to be able to access. In the case of the word processor, these would include all the documents that a user might want to create. In a game, it would include all the game endings that a user might visualize. In a storytree, it is all the ways that the user might imagine that the story could develop.

A Criterion for Excellence

I now direct your attention to the ratio of the number of accessible states to the number of conceivable states. You calculate this ratio by dividing the number of accessible states by the number of conceivable states. I suggest that this ratio provides us with a criterion for evaluating the overall merit of a product.

Consider now that the source of my satisfaction with my word processor lies in the fact that "anything I want to do, I can do." In other words, any state that I expect to access, I can access. Let's make up some plausible numbers for a word processor to see how the ratio works. Let's say that I can expect 100 gazillion states, but my word processor can't quite handle all of them; it can reach only 99.9 gazillion states. So we might calculate the ratio for a word processor as follows:

```
interactive excellence    = accessible states / conceivable states
                          = 99.9 gazillion states / 100 gazillion states
                          = 0.999
```

The ratio of accessible states to conceivable states is very nearly 1. On the other hand, I do not have the same experience with games. Many times in games I feel trapped by the design, unable to do the thing that I want to do. In other words, the state that I expect to access is not there. Thus, with games, the calculation might look like this:

```
interactive excellence    = accessible states / conceivable states
                          = 20 gazillion states / 100 gazillion states
                          = 0.20
```

We have here a way of gauging the interactive excellence of any interactive design. Now, I'm not suggesting that one should literally perform this calculation by counting up all the accessible states and all the conceivable states; I don't have a gazillion fingers. But we can use our imaginations to get a hunch as to whether the ratio is big or small.

More important, this suggests a way to improve our designs: the more closely the ratio approaches 1, the better our design will be. There are only two ways to increase a ratio: increase the numerator or decrease the denominator. I shall take up the latter case first.

Decreasing the Number of Conceivable States

Decreasing the number of conceivable states sounds silly, doesn't it? How can a designer lower the expectations of users, short of including in the package a coupon for a free frontal lobotomy at the nearest hospital? As it happens, we designers have a great deal of power, for we set the expectations of our users with the cues we give them and with the language of interaction that we provide to them. Most programs inflate user expectations and then confound those expectations. We do this by suggesting that the software universe inside our program is larger than it actually is.

What we need here is the closure discussed in Chapter 4. A good interactive design presents a closed and complete universe. Leave out the petty things so that you can implement the important features completely. The emphasis is not on parsimony but on closure, although parsimonious design is often necessary to achieve closure. Just remember that every chink in your system's closure will leak users into the void of unanticipated feature space. Your users, strangling in the vacuum, will perceive a deficiency and blame you for it—quite rightly. The expressive range of your design must be hermetically sealed against such leaks.

There are two means by which you can fail to achieve closure: your choice of feature set and your user interface. Too many designers succumb to overweening pride in creating products that attempt too much. In their eagerness to expand the universe of their design, they toss in features willy-nilly without recognizing the geometrically increasing expectations that such features instill in the users. Whenever you add a new feature to a design, that feature can interact with every other feature in your design. You may never consider what might happen if the gamma correction tool were used in conjunction with the Korean language translator, but some of your users certainly will. It is incumbent upon you to anticipate all those possible combinations and provide for them.

But the problem extends beyond the mere possibility of unanticipated combinations of existing features. A new feature added to a program often suggests related features to the user. If you add a hyperbolic sine function to your productivity application, you'd damn well better put in the hyperbolic cosine function and the hyperbolic tangent function, as well as the inverse of each of these functions; by creating one function, you have also created the expectation of five others. If you permit the user to export a special file format, then you must also permit the user to import that file format. If your game shows a door, the user must be able to open that door. If your word processor offers superscripts, it must also offer subscripts.

A good example of the user interface side of this problem is the *command-line interface*, such as used in DOS. You may not remember this once-ubiquitous operating system, so I'll describe it here. The screen was not graphical but textual; you saw a bunch of obscure text codes on the screen. Then you might type a command like this:

```
>C:\MYDIR\WORDPRO.EXE
```

This is one of the simpler commands. It directs the computer to run a program that can be found on the hard disk (C:) in a directory called MYDIR, and the program itself is called WORDPRO.EXE.

This looks simple enough, but personal computers kept growing, adding ever more capabilities and complexity. To cope with this growth, designers kept adding new commands to the basic set. The commands grew longer and more complicated; there were so many that only the experts could remember how to use them.

There is no question that command-line systems are inferior to the graphical user interfaces (GUIs); except for UNIX, a hoary and hairy command-line operating system for programming wizards, and Linux, its open-source derivative, the industry has abandoned such interfaces in favor of the GUIs. But the shift away from DOS was not arbitrary or accidental; there were fundamental reasons for it. The most important of these is that command-line interfaces increase the number of conceivable states far beyond the number of accessible states. You can type zillions of different expressions on a keyboard, and given the exotic spelling of DOS commands, almost anything you type looks reasonable, but only a few thousand of them will actually do anything useful. In other words, for DOS, the calculation looks like this:

```
interactive excellence     = accessible states / conceivable states
                           = a few thousand states / 100 gazillion states
                           = 0.000000...0001
```

That ratio is the best explanation for why DOS is dead.

The term *command-line interface* is usually applied to operating systems. When we talk about the same thing used inside an application program, we call it a *text parser*—and text parsers are an even worse disaster than command-line interfaces, because these latter use a parsimonious set of contractions. For what it's worth, you can be pretty sure that neither UNIX nor DOS will have a command as long-winded as, say, *dissimulate, DSMLT,* or *DSMT,* perhaps, but never anything so obvious as the actual word. Text parsers, on the other hand, purport to recognize a subset of normal English vocabulary. You can type expressions such as "Pick up the rock," and the parser will understand what you mean; isn't that wonderful? But there's a vicious devil lurking in the details of all parsers. A parser suggests to the user that any valid English expression will be accepted by the parser, but in fact most parsers have working vocabularies of a few thousand words. "Pick up the rock" might work, whereas "Pick up the stone," "Grab the pebble," "Take hold of the geode," or "Seize the stony slab" won't. Thus, text parsers perpetrate a cruel falsehood on a user. They create gazillions of conceivable states, but they provide only a few million accessible states (at best).

The simplest, easiest, and most honest way to decrease conceivable states is seldom used: tell the user candidly what you can't do. The hype-driven world of computers has closed its eyes to the clarifying power of the disclaimer. I'm sure that all the marketing suits out there will reach for the garlic and silver cross when I suggest that a little truth goes a long ways. I realize that marketing considerations require you to put a positive face on your software, but I warn you that a positive face does not compensate for a negative reality.

For example, it would have been much easier on us if all of Windows' capabilities *and* incapabilities were clearly presented somewhere. By including hundreds of obscure functional keypresses in the operating system, Microsoft ensured that reasonable people would suspect every key combination of doing something—and by failing to explicitly deny such expectations, it compounded the problem. Windows should have a little warning box that pops up whenever a user types some oddball combination of keys. The warning box should say something like, "Here is a list of all the oddball key combinations that do something, as well as their functions. And here is a list of all the oddball key combinations that don't do anything at all."

The same thing goes for mouse input. Every time a user clicks an inactive screen object, a pop-up window should explain why that object is inactive and what might make it active, if anything. A good system will go further and refer the user to controls that might accomplish what the user seems to be driving at. As I explain in Chapter 5, dimming menu items declares that they aren't functional, which is good, but it's even better to explain why they are dimmed and what can be done to activate them.

This rule of negativity is especially important with websites. The typical browsing user has a pretty clear need in mind and seeks only to determine if your website can satisfy that need. We all know that a website can be gigantic; that creates the expectation that it just might have what we want. So we browse and meander through the website, examining all manner of useless pages in search of our goal. The bigger and more thorough a website is, the more time we are likely to waste searching for our grail. It would be much better if every website devoted some page space to describing what it does *not* include; that might save the users lots of time. For example, I wasted a good deal of time at the Apple website trying to locate a bug report page, as I had found a couple of bugs in AppleWorks and I considered it my solemn duty as a user-citizen of the AppleWorks world to report bugs. Although my expectation of a bug report page was entirely reasonable, no such page was accessible, and the only way to learn this was by trying and failing repeatedly. Remember, negative information is every bit as useful as positive information.

If you build a big website all about George Washington, but you have no information about his financial affairs, you might as well say it up front. The user who pokes all through your website looking for that information is going to be just as disappointed either way; do you want her to be angry with you as well?

To conclude: we can decrease the number of conceivable states by carefully designing our listening structures to suggest nothing more than we intend. Where ambiguities remain, we should explicitly disclaim reasonably conceivable (but inaccessible) states. Remember that great black-and-white photographers pay just as much attention to shadow as to light.

Increasing the Number of Accessible States

How might we increase the number of accessible states? How can we make our trees thicker and bushier? The obvious answer is to provide more branch-points from each state—more verbs. Here are some techniques that will suggest more verbs.

Add More Variables

The first tactic we might use is to increase the number of variables used in our designs, most often by replacing a constant element of our application with a variable element. This is the most commonly used way to increase the power of applications, although in most cases I suspect it is used unconsciously, as an indirect result of other (less strategic) considerations. The earliest word processors, for example, did not permit multiple fonts; there was just one standard font. It didn't matter back then, as the printers also offered only a single font. The decision to add font capability to word processors could also be described as a decision to transform the printing front from a constant to a variable. Designers then gave control of that variable to the user. In general, adding more variables to your design always increases its flexibility and power. Thus, this one change led to other possibilities: new variables in font size and style.

Sometimes the constants in your program are not immediately recognizable as constants. For example, imagine a web page with a button that takes the user to another page. That link is a constant; would it be more useful as a variable? A search function, for example, could be thought of as a variable link whose value depends on the search text entered in the associated text field.

There is a dark side to this power: button-mania on the part of the designer often instills button-phobia on the part of the user. Every variable you add is one more chore for the user to handle, one more page in the user manual to be read. An old concept from game design is useful here: the *color-to-dirt ratio*. Every new feature that a game designer adds increases its colorfulness. If I were to add, say, stock speculation to Monopoly, that would certainly make the game more colorful to play. It would give the users more choice in their financial decisions, more flexibility, and more game-playing power. But it would also entail a whole raft of new rules, more cards for the Chance and Community Chest sets, some special spaces on the board, and so forth. All this bureaucratic detail constitutes *dirt* in the game design. The designer must then ask, "Is the color I'm adding with this feature worth all the dirt it brings in?" Similarly, the designer of any interactive application must always consider the color-to-dirt ratio of any contemplated feature. More variables add more color, but they also add more dirt.

For example, a great many programs now offer "configuration sets." These are different versions of the preferences options available in so many programs. The use of preferences allows the user to set up the program exactly the way she likes it. For example, preference files might contain information about the default font to use, the most commonly accessed server, the size and placement of the window for a new document, and so forth. Preferences are a great idea, and most programs now have some kind of preferences system. But some programs go even further: they make the preference file itself a variable, so that the user can choose among different groups of preferences by selecting one of several preference files. If you're in a mood for something exotic one morning, you can choose the Exotic preference file to get odd-shaped windows with that Asian-looking font. Later in the day, when you are to demonstrate something for your boss, you might change to the "Safe and Staid" preference file to give the program a more conventional look. I'm sorry, folks, but this is carrying the use of variables too far: you don't get much color for a lot of dirt. Sure, there are a few hundred geek users around the world who love the feature, but the great majority of users are too staid to appreciate such a feature.

Replace Boolean Variables with Arithmetic Variables

When you do add a variable, you should usually make it an arithmetic variable as opposed to a single-bit Boolean variable. A Boolean variable can take only two values: 1 or 0, yes or no. There are plenty of variables that require Boolean treatment. For example, in the Print dialog box, the check box for collation presents a Boolean variable. Let's face it; collation is something you either do or don't do; there's no partial form of collation. Thus, the user must answer the simple question, "Do you want it collated?" with either a yes or a no—a Boolean variable. We normally use check boxes to allow users to control Boolean variables.

An arithmetic variable, by contrast, is a number; it specifies how much or how many instead of just yes or no. In general, you should strive to find uses for arithmetic variables rather than the more limited Boolean ones. Don't simply keep track of *whether* the user has entered any keystrokes in the word processing document—record *how many* keystrokes he has entered. That way, instead of merely refusing to save an unchanged document, you can more intelligently advise your user: "You've typed 5,000 keystrokes without saving; why not take a break and let me save them for you?"

On a website, you can keep track of how many times a unique visitor has hit each page in the last 10 minutes; this knowledge permits you to treat that visitor in a more responsive manner. Perhaps he's lost, perhaps he's uncertain, perhaps he could use some guidance. An addendum to the page might just help him out.

For your educational software, if you are testing a student, don't just ask whether the answer was wrong; determine how many times the student answered the same type of question incorrectly. In some cases, you can even measure just how far off the mark the answer was. This information can be profitably put to use devising material to assist the student later.

Even some game genres are still stuck using Boolean variables when they should have graduated to arithmetic variables. Interactive fiction has a particularly egregious attachment to Boolean variables. The field could develop much more richness if it started using variables like "how big a rock does he carry" rather than "is he carrying a rock or is he not?"

Eschew Hard-Wired Branching for Computed Branching

Another wrong-headed practice is the use of explicitly defined branches in a tree. I have never seen this error in productivity applications, but it is depressingly common in games and educational software, and central to the organization of websites.

Whenever a designer creates an explicit link between one node and another, she implicitly rules out the many variations that might pop into a user's head. Explicit links are like Boolean variables: either you follow the link or you don't. A user whose needs are not as starkly defined as the choice you offer will expect a greater range of choices. Simple arrangements like this are acceptable for the basic navigational operation of a website, but for delivering greater value, you'll need to build your pages on the fly to meet the individual requirements of each user. Why must your pages be cast in stone before the user even arrives?

Every time you use a search engine, it builds a custom page presenting the results of your search. Why must customization be confined to something so simple as a list of items from a database? Instead of building 3,000 pages for your users to delve tediously through, why not organize your site more algorithmically, and build custom pages for each user? If you're designing a small website, explicit links are fine, but as sites grow larger, the need for more powerful listening powers increases dramatically—and text is far more expressive than mouse clicks. Let your users type what they want and then calculate how to give it to them.

Use Indirection

As much as possible, push your design thinking away from the concrete and towards abstraction and indirection. This is a complicated issue; I'll come back to it in Chapters 20 and 21.

Crawford diagrams provide another way of thinking about interactivity designs. They illustrate some of the common errors, and suggest some useful desiderata. They are not adequate as design blueprints. The ratio of accessible states to conceivable states is a good measure of the quality of the interaction. Work to decrease conceivable states and increase accessible states.