Once you have your design principles, you can use them as a measuring stick against the concepts you've generated to see which ones best fit. Hopefully several ideas will work within the guidelines, or could be tinkered with to fit.

But design principles can also be used from this point in the process forward to help make design decisions. When there are multiple options to choose from ("Should we ask users first, or just do it for them?"), the design principles can sometimes help make the correct decision clear.

Design principles can sometimes outlast the specific product itself, or even be extended across lines of products to give them all a similar grounding.

Summary

Brainstorming can be mysterious. Frequently an idea will come to you when you are not in a brainstorming session. Ideas seem to have a life of their own, but they can sometimes be coaxed into existence, and that's what you hope ideation will do.

The design principles you create are a way—granted, a subjective way—of measuring your ideas for value and feasibility. Of course, the only way to really tell if an idea is a good one is to play with it, test it out, and refine it. That is the topic of the next chapter.

For Further Reading

Six Thinking Hats, Edward de Bono

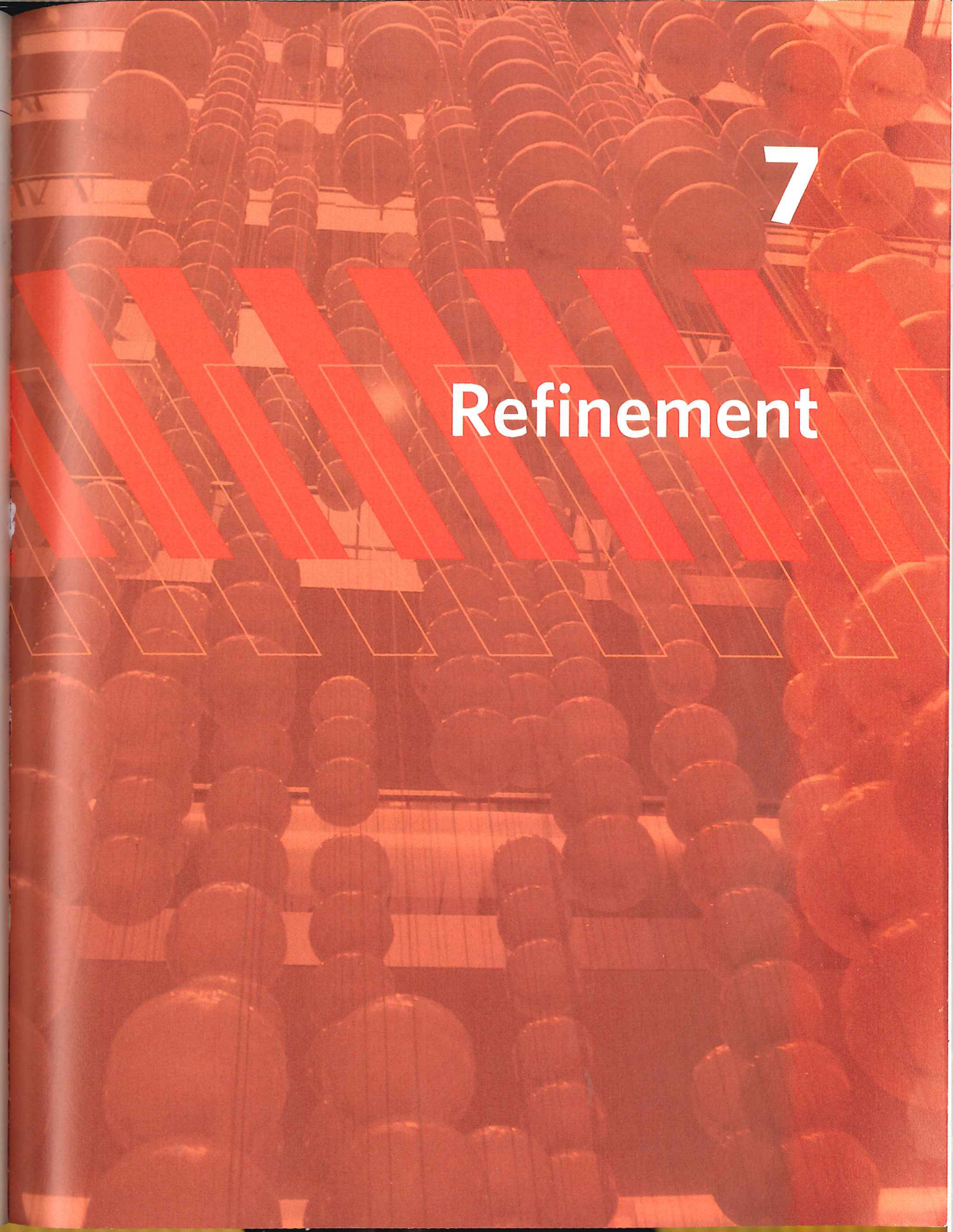
A Technique for Producing Ideas, James Webb Young

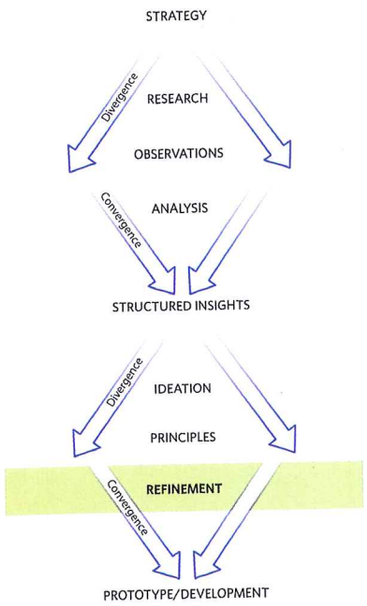
Thinkertoys: A Handbook of Creative-Thinking Techniques, Michael Michalko

The Seeds of Innovation: Cultivating the Synergy That Fosters New Ideas, Elaine Dundon

7

Refinement





Having a concept, no matter how brilliant, is not enough for a product. Concepts are relatively easy to come by; it is the execution of those concepts that matters. And execution is all about defining the details of the concept, fleshing it out until it works in a functionally- and aesthetically-pleasing way. Execution means refining the concept in order to work on the **details**.

Details are the small parts of the design where designers earn their paychecks. They provide moments of efficiency and delight for users, and are also where designers earn the respect of the developers, businesspeople, and manufacturers. Details often get overlooked in just concept projects. Constraints also are somehow less solid in the world of concepting than they are once you start to figure out how the product actually works.

Constraints

It's at this point in the process where constraints really rear their ugly head. Hopefully, via stakeholder interviews and design research (and experience), you're aware of a number of constraints by now:

- **Time.** How much time do you have to finish the project? When does the product need to launch/ship?
- **Money.** What's the budget for finishing the project? What's the price point or the business model of the product?
- **Technology.** What platform is the solution going to be made on? What systems are or need to be in place for this to work? Can you have the technology you need in the given time and budget?
- **Business needs.** How will this meet the business success metrics? What organizational support is there for this product?
- **User needs.** What does the user need to accomplish? How will this be better than any other solution? Does the solution have to be accessible to those with disabilities?
- **Context.** Are there physical limitations of size or weight? Where will the product be used, and how does that affect it?

- **Tools.** What kind of tools (software, manufacturing) will be used to build and maintain the product?
- **Teams.** What kind of team do you have to build this? What are their skills? Realistically, what can you collectively accomplish in the time given?
- **You.** What skills do you have? What are your weaknesses, and how can they be overcome?

Constraints end up defining the product more than one cares to admit. The best designers are those who can juggle the most constraints. "Design depends largely on constraints," noted Charles Eames. The trick is to figure out which constraints are impassible barriers and which can be moved or changed, given enough effort.

All projects, no matter what their constraints, should follow certain general principles and fundamentals of interaction design.

The Laws and Principles of Interaction Design

Interaction design, being a new field, doesn't have very many hard and fast rules, or "laws," to speak of. In a sense, interaction designers are still figuring out many of the basic principles of the work they do. However, there is a handful of laws that interaction designers may use from time to time, as well as basic principles that underlie all interaction design work. These laws and principles should always guide the work, not dictate it.

Direct and Indirect Manipulation

Objects can be manipulated in two ways: directly and indirectly. Although technically digital objects can be manipulated only indirectly (you can't touch something that's made of bits and bytes, after all), direct and indirect manipulation represent two ways of thinking about how to work with objects.

Direct manipulation is a term coined by University of Maryland professor Ben Shneiderman in the early 1980s. It refers to the process in which, by selecting a digital object with a finger or with a mouse or with some other extension of the hand, we can then do something to the object: move it, turn it, drag it to the trash, change its color, and so on. We can mimic an

action that we might perform on a similar object in the physical world. For example, we can scale an object by dragging a corner of it as though we were stretching it. Direct manipulation, because it more closely maps to our physical experiences, is supposedly more easily learned and used, especially for manipulating 3-D objects in digital space.

Of course, we also directly manipulate physical objects all the time, by pushing buttons, turning dials, flipping switches, and so on, which can cause either mechanical or digital effects. Through sensors, the behavior of objects can be affected by movement, such as an MP3 player with an accelerometer going into shuffle mode by being shaken.

In **indirect manipulation**, we use a command or menu or gesture in space or voice command that isn't directly a part of the digital object to alter that object. Choosing the Select All command in a menu and pressing the Delete key on the keyboard are examples of indirect manipulation. In the past, especially during the years before the Macintosh popularized the GUI, nearly all computer commands were indirect.

Interaction designers need to decide how digital objects in their products can be manipulated: directly, indirectly, or (more and more frequently) in both ways.

ure 7.1
e design of this
te latch provides
ual affordances
dicating how it
ould be used.

Affordances

How something manifests gives us cues as to how it behaves and how we should interact with it (**Figure 7.1**). The size, shape, and even weight of mobile devices let us know that they should be carried with us. The sleek black or silver look of digital video recorders like TiVo devices tell us that they are pieces of electronic equipment and belong alongside stereos and televisions.

Appearance is the major source of what cognitive psychologist James Gibson, in 1966, called *affordances*. Gibson explored the concept more fully in his 1979 book *The Ecological Approach to Visual Perception*, but it wasn't until Don Norman's seminal book *The Psychology of Everyday Things*, in 1988, that the term spread into design. An affordance is a property, or set of properties, that provides some

indication of how to interact with an object or feature. A chair has an affordance of sitting because of its shape. A button has an affordance of pushing because of its shape and the way it moves (or seemingly moves). The empty space in a cup is an affordance that tells us we could fill the cup with liquid. An affordance (or, technically, a *perceived* affordance) is contextual and cultural. You know you can push a button because you've pushed one before. On the other hand, a person who has never seen chopsticks would be puzzled about what to do with them.

Interaction design, especially in the refinement phase, can be thought of in part as providing affordances so that the features and functionality of a product can be discovered and correctly used.

Feedback and Feedforward

Feedback, as the term is commonly used in interaction design, is some indication that something has happened. Feedback should occur like crooked voting: early and often. Every action by a person who engages with the product or service, no matter how slightly, should be accompanied by some acknowledgment of the action: Moving the mouse should move the cursor. Pressing a key on your mobile phone should display a number.

To proceed otherwise is to court errors, some of them potentially serious. Frequently, if there is no immediate or obvious feedback, users will repeat the action they just did—for instance, pushing a button twice. Needless to say, this can cause problems, such as accidentally buying an item twice or transferring money multiple times. If the button is connected to dangerous machinery, it could result in injury or death. People need feedback.

Designing the *appropriate* feedback is the designer's task. The designer has to determine how quickly the product or service will respond and in what manner. Should the response be something simple such as the appearance of a letter on a screen (the feedback in word processing for pressing a key), or should it be a complex indicator such as a pattern of blinking LED lights on a device that monitors your stock portfolio?

There is little more annoying than talking to someone who doesn't respond. The same holds true for "conversations" with products and services. We need to know that the product "heard" what we told it to do and that it is

working on that task. We also want to know what the product or service is doing. A spinning circle or a tiny hourglass icon doesn't give users much transparency into what is happening.

COURTESY ORBITZ

Figure 7.2

The Orbitz searching screen won't make the wait shorter for search results, but it will make it seem shorter because of its responsiveness.

If a response to an action is going to take significant time (more than 1 second, which, believe it or not, can seem like a long wait), a good design provides some mechanism that lets the user know the system has heard the request and is doing something (**Figure 7.2**). This doesn't shorten the waiting time, but it makes it seem shorter. Excellent examples of such a mechanism are the indicators that tell you how long software installation will take. These indicators also assure the user that the process hasn't stalled.

The responsiveness of digital products, determined by the time between an action and the product's response (also called **latency**), can be characterized by these four basic levels:

- **Immediate.** When a product or service responds in 0.1 second or less, the user considers the response immediate and continues the task with no perceived interruption. When you push a key on your keyboard and a letter instantly appears, that is an immediate response.
- **Stammer.** If a product or service takes 0.1 second to 1 second to respond, users will notice a delay. If such a delay is not frequently repeated, users will overlook it. Repeated, it will make the product or service feel sluggish. For instance, if you press a key on your keyboard and it takes a second for the letter to appear on your screen, you'll notice the delay. If this happens with every key press, you will quickly become frustrated with your word processor.
- **Interruption.** After a second of no response, users will feel that the task they were doing was interrupted and their focus will shift from the task at hand to the product or service itself. If you click a Submit button to execute a stock trade and nothing happens for several seconds, you will worry about the trade and wonder if the Web site is broken. Multiple interruptions can lead to a disruption.

- **Disruption.** If a delay of more than 10 seconds occurs, users will consider the task at hand completely disrupted. Feedback such as a progress bar or a timer that indicates how long a process will take will allay users' concerns and also allow the user to decide whether to continue the process. Marquees in the London Underground indicating when the next trains will arrive are excellent examples of responsiveness that addresses this level of delay.

Related to feedback and also to affordance is what designer Tom Djajadiningrat calls **feedforward**: knowing what will happen *before* you perform an action. Feedforward can be a straightforward message ("Pushing this button will submit your order") or simple cues such as hypertext links with descriptive names instead of "Here."

Feedforward allows users to perform an action with confidence because it gives them an idea of what will happen next. It is harder to design into products and services than feedback, but designers should keep an eye out for opportunities to use it.

Mental Model

Mental model is the term for a user's internal understanding of how a system or object works, which may or may not reflect how the thing actually does work. The best mental models allow for a deep understanding of the thing, minus the complexities involved in making the thing work. For instance, most people have a mental model of how a car behaves, even though they don't know how a combustion engine works.

Mental models are usually constructed by users from the cues provided by the designer in the form of affordances, feedback, and feedforward. Indeed, using those very things, designers can manipulate the user's mental model significantly, hiding or exposing the product's inner workings. For example, a label on a car's steering wheel that reads, "Blow horn before starting car" would certainly change how you think about the functionality of the car, and especially the horn. Or imagine if when you turned the car on, a voice instructed the driver about fuel going into the engine. Sure, you would know more about the car, but it wouldn't make your driving any better.

Standards

There is a perennial debate among interaction designers about how closely to follow interface standards and when to break them. Do all applications have to work in a similar way? Should Ctrl-C or Command-C always copy whatever is selected? Does every menu bar have to have the same headings (File, Edit, View, and so on)? Both Microsoft and Apple have standards guidelines online that are religiously followed by many. Usability gurus such as Jakob Nielsen promote and swear by them.

There are certainly good reasons for having and using standards. Over many years, designers have trained users to expect certain items to be located in certain places (for example, the company logo goes at the top left of a Web site) and certain features to work in a particular way (for example, pressing Ctrl-Z undoes the last command). A design that ignores these conventions means that your users will have to learn something different, something that doesn't work like all their other applications work. A variation from the standard can cause frustration and annoyance.

So why ever violate or alter standards? Alan Cooper solved this dilemma with his axiom: *Obey standards unless there is a truly superior alternative.* That is, ignore standards only when the new way of doing a task is markedly, significantly better than what the users have previously used. Feel free to propose a new method of cutting and pasting, but it had better be unequivocally better than what users are accustomed to now. New standards don't have to be radical departures from the old standards, but even a slight change should be made with care because it subverts the user's expectations of how a product should work.

Fitts's Law

Published in 1954 by psychologist Paul Fitts, Fitts's (pronounced "fitzez") Law simply states that the time it takes to move from a starting position to a final target is determined by two things: the distance to the target and the size of the target. Fitts's Law models the act of pointing, both with a finger and with a device like a mouse. The larger the target, the faster it can be pointed to. Likewise, the closer the target, the faster it can be pointed to.

Fitts's Law has three main implications for interaction designers. Since the size of the target matters, clickable objects like buttons need to be a reasonable

size. This is especially true for touchscreens or on screens at a distance such as a television. As anyone who has tried to click a tiny icon will attest, the smaller the object, the harder it is to select. Second, the edges and corners of screens are excellent places to position things like menu bars and buttons. Edges and corners are huge targets because they basically have infinite height or width. You can't overshoot them with the mouse; your mouse will stop on the edge of the screen no matter how far you move it, and thus will land on top of the button or menu. The third major implication of Fitts's Law is that controls that appear next to what the user is working on (such as a menu that appears next to an object when the user right-clicks the mouse) can usually be opened more quickly than pull-down menus or toolbars, which require travel to other parts of the screen.

Hick's Law

Hick's Law, or the Hick-Hyman Law, says that the time it takes for users to make decisions is determined by the number of possible choices they have. People don't consider a group of possible choices one by one. Instead, they subdivide the choices into categories, eliminating about half of the remaining choices with each step in the decision. Thus, Hick's Law claims that a user will more quickly make choices from one menu of 10 items than from two menus of 5 items each.

A controversial implication of this law is that it is better for products to give users many choices simultaneously instead of organizing the choices into hierarchical groups, as in drop-down menus. If followed to an extreme, this approach could create some truly frightening designs. Imagine if a content-rich site like Yahoo or Amazon presented all of its links on the home page, or if your mobile phone displayed all of its features on its main screen.

Hick's Law also states that the time it takes to make a decision is affected by two factors: familiarity with the choices, such as from repeated use, and the format of the choices—are they sounds or words, videos, or buttons?

The Magical Number Seven

Hick's Law seems to run counter to George Miller's Magical Number Seven rule. In 1956, Miller, a Princeton University psychology professor, determined that the human mind is best able to remember information in

chunks of seven items, "plus or minus two."[1] After five to nine pieces of information (for instance, navigation labels or a list of features or a set of numbers), the human mind starts making errors. It seems that we have difficulty keeping more than that amount of information in our short-term memory at any given time.

Some designers have taken the Magical Number Seven rule to an extreme, making sure that there are never any more than seven items on a screen at any given time. This is a bit excessive, because Miller was specifically talking about bits of information that humans have to remember or visualize in short-term memory. When those bits of information are displayed on a screen, users don't have to keep them in their short-term memory; they can always refer to them.

But designers should take care not to design a product that causes "cognitive overload" by ignoring the Magical Number Seven rule. For example, designers should never create a device that forces users to remember unfamiliar items across screens or pages. Imagine if you had to type a new phone number on three separate screens of your mobile phone. You'd scramble to do so before the number faded from your short-term memory.

Tesler's Law of the Conservation of Complexity

Larry Tesler, one of the pioneers of interaction design (see the interview with him in Chapter 6), coined Tesler's Law of the Conservation of Complexity, which states that some complexity is inherent in every process. There is a point beyond which you can't simplify the process any further; you can only move the inherent complexity from one place to another.

For example, for an e-mail message, two elements are required: your e-mail address and the address of the person to whom you are sending the mail. If either of these items is missing, the e-mail can't be sent, and your e-mail client will tell you so. It's a necessary complexity. But some of that burden has likely been shifted to your e-mail client. You don't typically have to enter your e-mail address every time you send e-mail. The e-mail program handles that task for you. Likewise, the e-mail client probably also helps you by remembering e-mail addresses to which you've sent mail in the past, so that you don't

1 See "The magical number seven, plus or minus two: Some limits on our capacity for processing information" Psychological Review, 63, 81-97

have to remember them and type them in fully each time. The complexity isn't gone, though—instead, some of it has been shifted to the software.

Interaction designers need to be aware of Tesler's Law for two reasons. First, designers need to acknowledge that all processes have elements that cannot be made simpler, no matter how much they tinker with them. Second, designers need to look for reasonable places to move this complexity in the products they make. It doesn't make sense for users to type their e-mail addresses in every e-mail they send when the software can handle this task. The burden of complexity needs to be shared as much as possible by the products interaction designers make.

The Poka-Yoke Principle

Legendary Japanese industrial engineer and quality guru Shigeo Shingo created the Poka-Yoke Principle in 1961 while working for Toyota. Poka-Yoke roughly translates in English to mistake proofing: avoiding (**yokeru**) inadvertent errors (**poka**). Designers use Poka-Yoke when they put constraints on products to prevent errors, forcing users to adjust their behavior and correctly execute an operation.

Figure 7.3
An illustration of the Poka-Yoke Principle. The USB cord will fit into only a particular slot on this laptop computer.

Simple examples of the application of Poka-Yoke are the cords (USB, FireWire, power, and others) that fit into electronic devices only in a particular way and in a particular place, and thus prevent someone from, say, plugging the power cord into the hole where the headphones go (**Figure 7.3**). In this way, Poka-Yoke ensures that proper conditions exist *before* a process begins, preventing problems from occurring in the first place. Poka-Yoke can be implemented in lots of forms: by signs (Do not touch the third rail!), procedures (Step 1: Unplug toaster), humans (police directing traffic around an accident), or any other entity that prevents incorrect execution of a process step. Where prevention is not possible, Poka-Yoke mandates that problems be stopped as early as possible in the process.

Poka-Yoke often manifests itself in interaction design via the disabling of functionality (or the navigation, for example, the menu item or the icon)

when conditions for its use have not yet been met. When doing this, it's also a good practice when possible to show via tool tip or other means what conditions will enable the functionality.

Errors

In an ideal situation, no system should ever present an error message to a user unless the user has done everything *right* but the system itself cannot respond correctly. "Errors" are often a sign of poor design or engineering, or of not applying the Poka-Yoke Principle.

When an error occurs, you should always provide a way to fix the error, or otherwise provide information about why the error occurred.

Frameworks

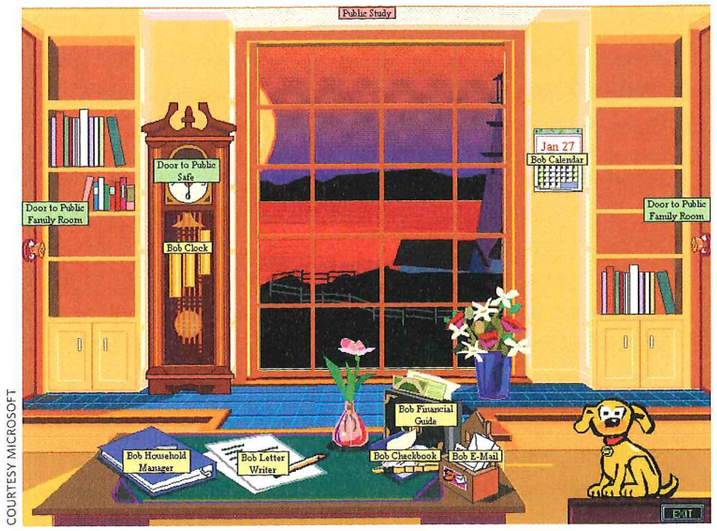
Every product needs a framework: an actual or metaphysical structure that defines the product and integrates the content and functionality into a unified whole. Without it, your product will seem disjointed, a collection of random stuff that users will struggle to make sense of.

There are three main kinds of frameworks that can be applied to a product: metaphor, postures, and structure.

Metaphor

Metaphor can be a way for users to understand abstract (digital) concepts. The most famous example, of course, is the desktop metaphor, which helped unify the graphical user interface (GUI) we've used for the last 30 years. Metaphor can suggest everything from how a product should function to how its visual or physical form is shaped.

Of course, metaphor can be abused. Witness Microsoft's infamous Bob (**Figure 7.4**), which imagined the operating system as a physical house. And metaphor can also be inaccurate—you can't do everything on your virtual desktop that you can on your physical one, after all. But it can be a powerful frameworking tool.

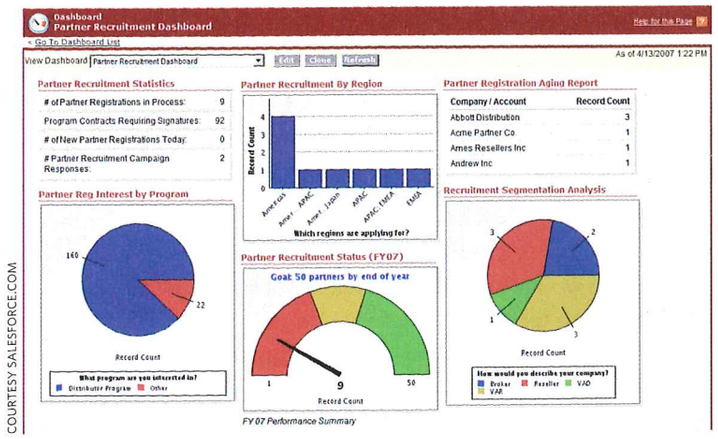


COURTESY MICROSOFT

Figure 7.4
Microsoft's Bob is an example of metaphor run amok.

Dashboards and Control Panels

A very common metaphor is that of a dashboard or control panel (**Figure 7.5**). Many of the applications, Web sites, consumer electronics, and appliances use this metaphor, lifted directly from physical objects, as a way of clustering and grouping both system information and controls.



COURTESY SALESFORCE.COM

Figure 7.5
An example of a dashboard-style layout.

Postures

Over the last 30 years, several common types of structures have emerged for the design of software. Alan Cooper calls these **postures**[2] and there are four principal ones:

- **Sovereign.** For applications users will need to use often, intensively, and for long periods of time, a sovereign posture might make sense. Sovereign applications such as Microsoft Word are complex, large, and take up a large portion of the screen when in use. Sovereign applications have many features and lots of work or viewing space, and typically the application window is broken up into several panes (for example, one pane for an overview, one for working, one for a detail view).
- **Transient.** For applications that are temporary and users need only briefly, such as installers and widgets like calculators, transient posture is appropriate. Transient applications use only a small amount of screen real estate and have few, simple controls that are clearly labeled.
- **Daemonic.** Applications that mostly run in the background, such as Growl or virus detectors, often utilize the daemonic posture. Unless absolutely necessary, these applications shouldn't intrude on the user's attention. The controls for them are mostly limited to setup and configuration through a simple control panel.
- **Parasitic.** An application, such as the Windows Start bar or TweetDeck, that supplements another application or service can have a parasitic posture. Parasitic posture applications are present for long periods, but are generally smaller than sovereign applications and have limited functionality.

Structure

Even if you choose a posture such as sovereign, you will likely have to determine, at least roughly, what overall form and layout it will take, such as the layout of panes in the application. And if there is particular hardware involved, such as on a mobile device or piece of consumer electronics, the interplay between hardware and software has to be considered as well.

2 For more details on postures, see *About Face 3.0*

Functional Cartography

If your project combines hardware and software, you need to determine where the functionality "lives." This is the **functional cartography**. Once you have a list of functionality and an understanding of their context of use, you can go about determining whether the controls for that functionality should be analog (physical buttons, sliders, dials, and so on), digital (onscreen controls), or some hybrid of the two (for example, soft keys). Soft keys are physical buttons alongside screen labels, and the context changes the functionality (and accompanying label) of the button.

How the functional cartography is decided depends on a number of factors:

- **Context.** When and where will the functionality be used? Does it need to be accessed rapidly? In the dark or unseen (in a pocket or behind the device)? With the screen idle or off?
- **Priority.** How important is this piece of functionality? Does it always need to be available? Is it used very often?
- **Cost.** How much does it cost (in terms of money, resources, weight, and power consumption) to have a screen at all? Or an additional physical control?
- **Ergonomics.** For the target users, what is the easiest physically to use?
- **Aesthetics.** Is another physical control going to ruin the form? Is the screen needed to have these controls going to be too large?
- **Tangibility.** How tactile does the feature need to be? Does it need to have the presence (and resulting affordance) of a physical control, or does a touchscreen (perhaps with haptic feedback) work as well?

The resulting functional cartography can be documented in a variety of ways. Often a simple table will suffice, with columns for Physical and Digital, and the functionality in rows below, showing in which category they will be placed. Another way to document (if you are farther along in the design process) is an illustration or sketch of the physical form, noting the functionality that resides on- and off-screen.

Once a functional cartography is done, it becomes easier to sketch, model, and prototype the device, as the location of functional pieces is now better known. A functional cartography need not be set in stone, however. During prototyping or modeling, it might become clear that a physical control is necessary, or vice versa. But a functional cartography will at least give a

starting place to distribute features and discuss the interplay between form and function.

Site/Screen/State Maps

One way of determining structure organically is to figure out how the pieces of functionality flow into one another and how the user navigates between them. This is often done after a task flow (see later in this chapter) has been made.

On the Web, this is sometimes accomplished via a site map (**Figure 7.6**), as different pieces of functionality live in different physical areas of the site, accessed by hyperlinks. This is also true of other products like mobile phones where, for example, Web browsing is often in a separate space from dialing the phone.

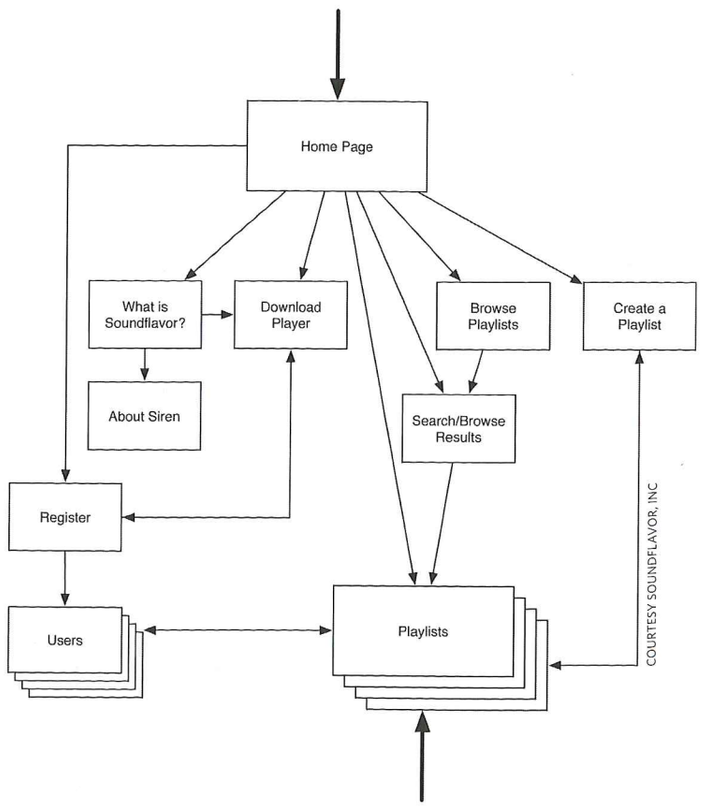


Figure 7.6
A simple site map.

The organization and labeling of features, information, and content is the discipline of **information architecture**. Information architecture draws upon library science techniques to structure information spaces in ways that make finding, navigating between, and understanding content easy.

But increasingly, even on the Web, functionality involves not going to another area, but simply shifting state. Practically every product involving interaction design changes over time. A **state** can be thought of as a paused moment of a time-based system. A state captures a particular moment in an interaction. States to pay particular attention to are initiation, activation, and updates.[3] Initiation is the default state immediately before an action begins. What does the screen (if any) look like, and what does the user do to change that (for example, rolling over a button, clicking a link)? Activation is what happens during an action. For example, what happens while the user is dragging an item across the screen or when a button is pushed? Updates are the state after the user has finished an action, how the product has changed.

Modes are a general condition created by the user or the system that allows for different functionality (and/or different states) to be accessed. For example, in some applications, you can only affect content when you go into editing mode. Modes are controversial in that they add complexity to any system and make mental models more challenging. They also create a lot more *conditional situations* that need to be documented and accounted for. In addition to pages or screens, you can create a flow between modes and states as well.

Mapping out pages, screens, states, and modes can create an overview that helps unify the product for *you*. You still need to pay attention to the overall impression that you're giving your users via affordances and feedback. Users need to understand the product as a whole before becoming familiar with the details.

Documentation and Methods of Refinement

Product concepts are refined mostly by thinking through them, which means either by simply starting to build them (see Chapter 8) or by putting ideas on paper, whiteboard, or screen and seeing where they lead. The

3 See *Plans and Situated Actions: The Problem of Human–Machine Communication* by Lucy Suchman

documents generated by this process are typically called documentation, but that implies detailing something that is completed. These are working documents that should evolve over time; documentation is an unfortunate name for them.

Designers should create exactly as much documentation as it takes to execute the project well, and no more. If the designer's team responds well to use cases, then by all means the designer should produce them. If a client couldn't care less about them, the designer shouldn't do one unless the designer or the team finds it helpful.

If a document doesn't communicate anything useful, it is worthless—less than worthless, in fact, because it squanders the designer's time. Each document produced should take the project one step closer to completion.

Scenarios

Scenarios provide a fast and effective way to imagine the design concepts in use. In a sense, scenarios are prototypes built of words.

Scenarios are, at their heart, simply stories—stories about what it will be like to use the product or service once it has been made. The protagonists of these stories are the personas (see chapter 5). Using a scenario, designers can place their personas into context and further bring them to life. Indeed, scenarios are one of the factors that make personas worth having. Running through the same scenario using different personas is an excellent technique for uncovering what needs to be included in the final product.

Consider an e-commerce Web site, for example. One persona is Juan, a very focused shopper who always knows exactly what he wants. Another persona is Angela, who likes to look around and compare items. If the designer imagines them in a scenario in which they are shopping for an item, the scenario starring Juan will have him using search tools, and the scenario starring Angela will have her using browsing tools.

One common scenario that works well for almost every product or service is one that imagines first-time use. What happens when the personas encounter the product or service for the first time? How do they know what to do and how to use the product or service? What does it feel like to them? Running each persona through a first-time use scenario can reveal how to tailor the final design to appeal to and work for each persona.

A picture can be worth a thousand words, but a few words can also be worth quite a few pictures. Consider this example from a scenario for an online grocery delivery service:

Sarah logs onto her BigGrocery account. She sees her order from last week and decides to use it again for this week's order. She removes a few items by dragging them off her BigGroceryList. Her total cost adjusts appropriately. She has all the groceries she wants now, so she clicks the Deliver button. Her saved credit card account is charged, and her confirmation page tells her to expect the groceries in about an hour.

This scenario took only a few minutes to write, but it would have taken hours to storyboard, days to wireframe, and weeks to prototype. Using scenarios, designers can sketch with words.

Sketches and Models

Of course, designers can sketch with images as well as words (**Figure 7.7**). As stated earlier, the designer's best tool has been and continues to be the physical drawing surface (paper, whiteboard) and the physical drawing instrument (pencil, pen, crayon, marker). Nothing digital thus far has been able to match the flexibility, speed, and ease of sketching on a piece of paper or whiteboard. Space is just one reason—even the largest monitor cannot compete with wall-sized whiteboards or sheets of paper fastened together.

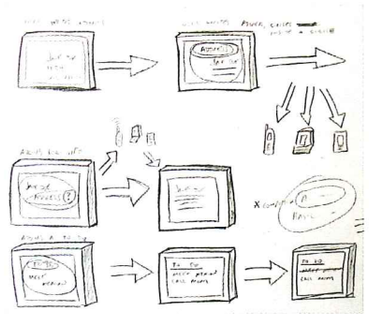


Figure 7.7
Before opening up any software, spend some quality time sketching with pens, pencils, and paper. Sketches have the added bonus of looking unfinished, so no one is inhibited from discussing their flaws.

Another form of sketching is modeling, which is useful for exploring physical forms. Models can be made of a variety of materials, from clay to cardboard to Styrofoam. Large blocks of Styrofoam can even be used to model physical spaces. Even crude blocks of wood, like those carried around by Jeff Hawkins to test the size, shape, and weight of the original PalmPilot,[4]

4 See, for instance, "Jeff Hawkins: The Man Who Almost Single-Handedly Revived The Handheld Computer Industry" by Shawn Barnett in *Pen Computing* magazine. Online at www.pencomputing.com/palm/Pen33/hawkins1.html

can be models. Models, like sketches, can be rapidly put together, to give rough approximations of physical objects and environments.

Sketching and modeling should be done throughout the design process, of course, but they are most helpful as visualizations of concepts and ideas that are still being formed to help to clarify and communicate those ideas and concepts.

Sketches and models are, by their nature, informal, and they can be easily changed. Viewers feel free to comment on them for just this very reason. This is a good thing, and no designer should feel overly attached to them.

Storyboards

Once a scenario and sketches have been created to show what a product or service could be like, designers can create a storyboard (**Figure 7.8**) to help illustrate the product or service in use.

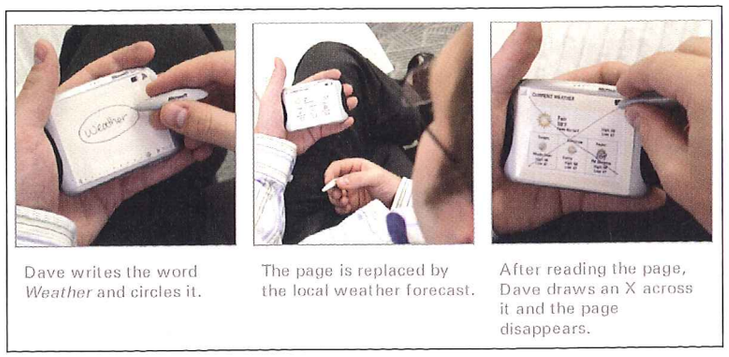


Figure 7.8
This storyboard was done with photography, but many storyboards are drawn.

Storyboarding is a technique drawn from filmmaking and advertising. Combining a narrative with accompanying images, designers can powerfully tell a story about a product or service, displaying its features in a context.

The images on a storyboard can be illustrations or staged photos created expressly for the storyboard. Generic or stock images are not recommended, as they will come off as stilted and likely won't be specific enough to match the scenario). Storyboards consist of these image panels, with accompanying text that can be drawn directly from the scenarios.

Storyboards can also be used in conjunction with a wireframe (discussed later in this chapter) to illustrate the details of a complicated process or function. Using a storyboard, a designer can show key moments of an action. For example, a complicated drag-and-drop procedure could be shown with panels illustrating the first moment that the user picks up an object, what happens during dragging, and what happens when the object is dropped.

Task Flows

Once you know what tasks have to be designed for (possibly after doing a task analysis as detailed in Chapter 5), putting those tasks into a sensible order, or flow, is important. Task flows (**Figure 7.9**) show the logical connections that will be built into wireframes (discussed later in this chapter). You can't, for instance, use the Back button on your Web browser to go back to a previous page until you've been to more than one page. You can't connect a phone call until you've entered a number. You can't change your preferences until you've registered. And so on.

Figure 7.9
A task flow for a voice interface.

Putting tasks into flows helps the designer begin to see the product take shape. Task flows can suggest page order on a Web site or in a wizard. Since task flows show where users will have to perform certain actions, they help clarify the implementation of controls (see later in the chapter). And where decisions have to be made, flows show where menus and information (or affordances) will have to be included.

Use Cases

Programmers have used use cases in the design of software for years. Indeed, it is this tradition that gives use cases some of their power: developers are very accustomed to seeing them and will likely understand them immediately, as will the business people who have over the years had to use them to communicate with programmers. Other design documents, while gaining recognition and acceptance, are simply not as well established.

Use cases are a means of roughing out the functionality of a product or service. A use case attempts to explain in plain language what a certain function does and why.

Uses cases also describe whom the function involves. Cases begin by identifying a set of potential actors. These actors can be based on the personas, but they can even be simpler than that. For example, "the user" can be one of these actors. Another of these actors is typically "the system." The system is the generic actor for any automatic or computing process. It is typically these processes that programmers have been interested in defining, but use cases don't need to be limited to describing events involving the system.

Use cases have the following form:

- **A title.** This should be descriptive, since it will be referenced often, both in documents and conversation. For example, a use case from an e-mail project might be called "Send an E-mail."
- **The actors.** Who is performing the function? In the e-mail example, the actors are the user and the system.
- **The purpose.** What is this use case meant to accomplish and why? For the function sending an e-mail, the purpose would be something like this: "An actor wants to send a message to someone electronically."
- **The initial condition.** What is happening when the use case starts? In our example, it is simply that the e-mail client is open.

- **The terminal condition.** What will happen once the use case ends? In the e-mail example, the condition is again simple: an e-mail has been sent.
- **The primary steps.** Discrete moments in this piece of functionality. In the e-mail example, these would be the steps:
  1. Actor opens up a new mail window.
  2. Actor enters the e-mail address of the recipient or selects it from the address book.
  3. Actor enters a subject.
  4. Actor enters message.
  5. Actor sends mail via some method (for example, a button click).
  6. The system checks to make sure the mail has a recipient address.
  7. The system closes the mail window.
  8. The system sends the mail.
  9. The system puts a copy of the mail into the sent mail folder.
- **Alternatives.** Alternatives are other use cases that may consider the same or similar functionality. In the e-mail example, Reply to Sender and Forward Mail might be use case alternatives.
- **Other use cases used.** Frequently, one piece of functionality is built upon another. List those for reference. The e-mail example includes a few functions that could have their own use cases: Open an E-mail Window, Select an Address from the Address Book, and Confirm Recipient Address might all be separate use cases.

Use cases can be broad (Send an E-mail) or very detailed (Confirm Recipient Address). Use cases can also be very time consuming to create, and a complicated system could potentially have dozens, if not hundreds, of use cases. Use cases are, however, an excellent tool for breaking down tasks and showing what the system will have to support.

Mood Boards

Mood boards (**Figure 7.10**) are a means for the designer to explore the emotional landscape of a product. Using images, words, colors, typography, and any other means available, the designer crafts a collage that attempts to

convey what the final design will feel like. Images and words can be found in magazines and newspapers or online image galleries, or can be created by the designer. Some designers take and use their own photographs for mood boards.

Figure 7.10
Mood boards are one way for designers to consider the emotional content of products.

Traditionally, mood boards were made on large sheets of poster board (thus, the name). The advantage of this approach was that the result could be posted on a wall to be glanced at frequently for inspiration. But this doesn't need to be so. Mood boards can be created digitally: as animations, movies, screen savers, or projections on a wall. The advantage of digital mood boards is that they can include movement and sounds—something traditional paper mood boards obviously cannot do.

The important point is that whatever form the mood board takes, it should reflect *on an emotional level* the feeling the designer is striving for in the product or service. The mood board shouldn't be challenging intellectually; like a good poem or piece of art, it should affect viewers viscerally.

Wireframes

Wireframes (**Figure 7.11**) are a set of documents that show structure, information hierarchy, controls, and content. They have their roots in architectural drawings and network schematics (in fact, they are sometimes called schematics). Next to prototypes, wireframes are usually the most important document that interaction designers produce when working on products. (Services don't typically have wireframes. Instead they have service blueprints; see later in this chapter.) Wireframes are a means of documenting the features of a product, as well as the technical and business logic that went into those features, with only a veneer of visual design (mostly just the functionality's controls). They are the blueprints of a product. Developers, industrial and visual designers, copywriters, and business people use wireframes to understand and build the product in a thoughtful way without being distracted by the visual or physical form.

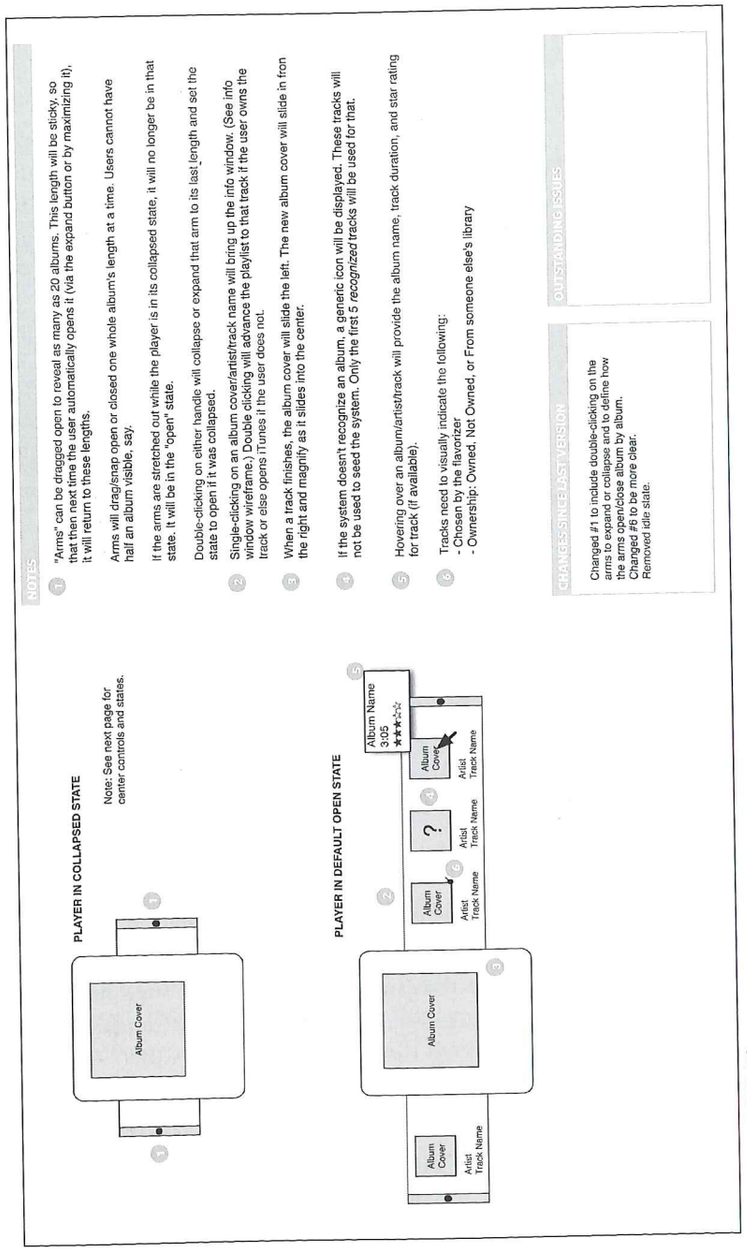
Wireframes are tricky documents to create because of the multiple audiences that read and use them. Clients want to see how the design meets their business goals. Developers want to see how the product works (and doesn't work—for instance, what happens when an error occurs) so they can know what they need to code. Visual or industrial designers want to see what visual or physical elements will need to be designed, such as the number and type of buttons. Copywriters want to see what they need to write: help text, manuals, headlines, and so on. And designers want to be able to refer to them in the future to remember details such as why there are two buttons instead of one for a certain feature. Accommodating the needs of these various audiences in one document is the designer's task.

In short, the wireframe is an inventory of all the elements that must be accounted for on a particular screen, Web page, or state.

Wireframes typically have three main areas: the wireframe itself, the accompanying annotations, and information about the wireframe (wireframe metadata).

Figure 7.11

A wireframe for a desktop music player.



The Wireframe Itself

The wireframe itself is a detailed view of a particular part of a product. Wireframes can show anything from an overview of a product—the form of a PDA, for instance—to detailed documentation of a particular functionality, such as the volume control on a music editing application.

Wireframes should rough out the form of a product. Form is shaped by three factors: the content, the controls necessary to discover and engage with the functionality, and the means of accessing or navigating to those two things. Thus, the wireframe needs to include indicators of content and functions as well as the elements for navigating them (buttons, switches, menus, keystrokes, and so on).

Content is a deliberately vague term that includes text, movies, images, icons, animations, and more. Content strategy is the planning for the creation, publication, and governance of the content that goes into a product.[5] Ideally, any content you are working with will be known before you begin the wireframing process, or at least the specific types and components as determined by the content strategy.

If you don't know the content or have pieces of representative content, you will have to represent it on wireframes by leaving placeholders (usually boxes with an X through them) for images/video and greeked or dummy text. This dummy text is often the one used by typesetters since the 1500s: *Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua*. It's become somewhat of a tradition to use it in wireframes.

Functionality consists of the controls—the buttons, knobs, sliders, dials, input boxes, and so on—of a feature, as well as the accompanying labels and feedback to those controls. A simple Web site form, for example, usually consists of labels ("Enter your name"), text boxes (where you enter your name, for instance), radio buttons ("Male? Female?"), check boxes ("Check here to join our mailing list!"), a Submit button, a Cancel button, and error messages ("You forgot to enter your name!"). All of these need to be documented on the wireframe.

5 For more about content strategy, see "The Discipline of Content Strategy" by Kristina Halvorson at www.alistapart.com/articles/thedisciplineofcontentstrategy/ or her book *Content Strategy for the Web*

There also needs to be a way to find and use the content and functionality: navigation. Navigation can consist of any number of methods, such as hyperlinks, simple drop-down menus, toolbars with widgets, and complex manipulations in physical space. On some mobile phones, for instance, pushing the cursor key down while pressing the star key locks the phone. On a digital camera, to view the content (the pictures that were taken), the user may have to change the mode of the camera and then use buttons to flip through the photos.

All these components should appear on the wireframe in a way that shows their general placement and importance. Note that the same wireframe can be used to design many different forms; wireframes can be interpreted in different ways by the visual or industrial designer. What is important is that all the items (content placeholders, functionality, and navigation) needed to create the final product be on the wireframes.

For many products, such as those with small screens or touchscreens, it can make sense to do wireframes drawn to the exact scale of the screen, so that problems do not arise when moving into visual design, prototyping, and production.

Anything on a wireframe that is not obvious or labeled should have a corresponding annotation.

Annotations

Annotations are brief notes that describe nonobvious items on the wireframe. They explain the wireframe when the designer isn't there to do so. When developers or clients want to know the reason for a button, they should be able to read the annotation and understand not just what the button does, but also *why* the button is there. Documenting "why" is a challenge, since annotations should be brief. But there is a vast difference between an annotation that says, "This button stops the process" and one that says, "This button stops the process so users don't have to wait for long periods." In the second version, the reader immediately knows the reason for the button. If a change occurs in the process ("The process now takes only a second"), it's easier to see how to adjust the design appropriately.

Here is a partial list of wireframe objects that should be annotated:

- **Controls.** (See later in this chapter for a list of controls.) What happens when a button is pushed or a dial is turned or a hyperlink is clicked.

- **Conditional items.** Objects that change based on context. For example, in many application menus, certain items are dimmed depending on what the user is doing at the time.
- **Constraints.** Anything with a business, legal, logical, or technical constraint (for example, the longest possible length of a password or the legal reason that minors cannot view certain content).
- Anything that, due to space, could not be shown in the wireframe itself (for example, every item on a long drop-down menu).

Wireframe Metadata

Each wireframe should have information about that wireframe—that is, wireframe metadata. Every wireframe should include the following:

- **The designer's name.**
- **The date the wireframe was made or changed.**
- **The version number.**
- **What has changed since the last version.** Clients like this; it shows that the designer is actively addressing issues that have arisen during the project.
- **Related documentation.** Any related documentation (ideally with a specific page number) that is relevant to this wireframe: business requirements, technical specifications, use cases, and so on. If there are questions about the wireframe ("Did we really say that the robot wouldn't swim?"), appropriate documents can be referenced.
- **Unresolved issues.** Are there problems with the wireframe that still need to be decided?
- **A place for general notes.** This is the place for the designer to express any final reservations about the product—especially the constraints that affected it. I have occasionally noted where business or technical constraints have had a negative impact on a product and should be addressed. In this way, designers can either argue for changes upon presenting the wireframes, or, if the clients or developers are reluctant to change the constraints, bring them up in the future when complaints arise or another version is planned.

Service Blueprint

Much as wireframes are key documents for digital products, **service blueprints** (**Figure 7.12**) are critical documents for services (which most products are part of anyway). Service blueprints present two major elements: service moments and the service string.

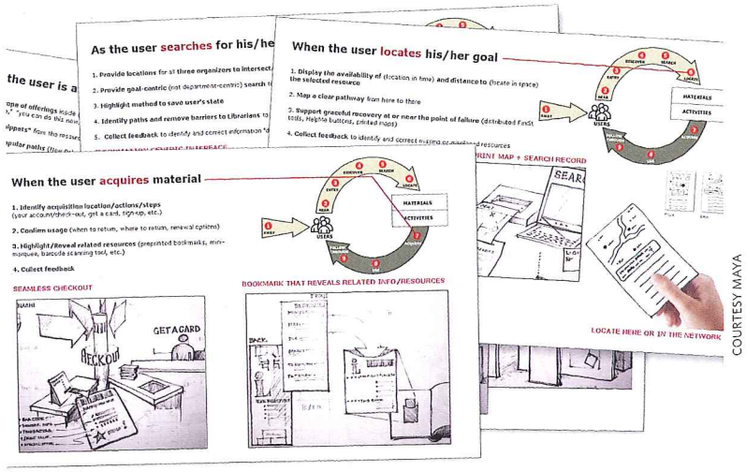


Figure 7.12
A piece of a service blueprint, part of the MAYA Carnegie Library of Pittsburgh project. Service blueprints show not only discrete moments in the service, but also how those moments flow together in a service string.

Service Moments

Every service is composed of a set of discrete moments that can be designed. For example, a car wash service has (at least) the following service moments:

- Customer finds the car wash.
- Customer enters the car wash.
- Customer chooses what to have done (washing, waxing, and so on).
- Customer pays.
- Car moves into the car wash.
- Car is washed.

- Car is dried.
- Interior of the car is cleaned.
- Customer leaves the car wash.

Each of these moments can be designed, right down to how the nozzles spray water onto the car. The service blueprint should include all of these moments and present designs for each one. And since there can be multiple paths through a service, there can be multiple designs for each moment. In the car wash scenario, perhaps there are multiple ways of finding the car wash: signs, advertisements, a barker on the street, fliers, and so on.

Here, the list of touchpoints (see chapter 5) can come into play. Which touchpoint is or could be used during each service moment? For each service moment, the touchpoints should be designed. In our car wash example, for instance, the customer paying probably has at least two touchpoints: a sign listing the washing services available and their costs, and some sort of machine or human attendant who takes the customer's money. All of these elements—what the sign says and how it says it, how the machine operates (Does it accept credit cards? How does it make change?), what the attendant says and does—can be designed. A major part of the service blueprints should be the brainstormed ideas for each touchpoint at each service moment. Each service moment should have a concept attached to it, such as the sketches in Figure 7.12 showing a possible check-out kiosk and a bookmark for related library information.

Ideally, each moment should have a sketch or photograph or other rendering of the design, similar to single storyboard frames.

For each service moment, the service blueprint should show what service elements are affected: the environment, objects, process, and people involved. Designers should especially look for service moments that can deliver high value for a low cost. Sometimes small, low-cost changes or additions to a service can quickly provide high value to users. For instance, some airlines found that passengers want a drink as soon as they board. But because other passengers are still boarding and in the aisle, flight attendants cannot offer drink service at that time. The solution was to put a cooler with water bottles at the front of the plane, so that passengers, if they want, can get a drink as they board—a low-cost, high-value solution.

Service String

The second component of a service blueprint is the **service string**. The service string shows the big idea for the service in written and visual form, usually in the form of storyboards. Designers create service strings by putting concepts for various service moments together to form a scenario, or string, of events that provide a pathway through the service.

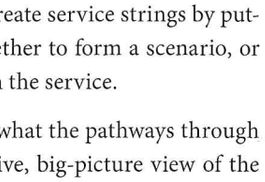
The service string demonstrates in a vivid way what the pathways through the service will be and provides a comprehensive, big-picture view of the new service. Viewers can see how customers order, pay for, and receive the service, and how employees provide the service. For example, a service string for the earlier car wash example would show in a single scenario customers seeing the new signs, customers using the new machine to pay for the car wash, the special washing service, the attendants who hand-dry the cars, and the new vacuum for cleaning out the cars after they are washed.

Controls

Most applications and devices that interaction designers currently design have some sort of visible controls to manipulate the features of the product—a dial to control volume on a stereo, for example, or a slider to select a date range. (The major exceptions are voice and gestural interactions, discussed later in this chapter.) **Controls** provide both the affordances needed to understand what the product is capable of, and the power to realize that capability.

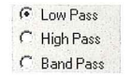
This section describes many of the basic controls that interaction designers can use as a palette. Almost all of these controls have their own standard feedback mechanisms (a switch moves and stays in its new position, for instance) that interaction designers should consider:

- **Switch.** A toggle switch is a very simple control. It moves from one setting ("on") to another ("off") and stays there until changed.



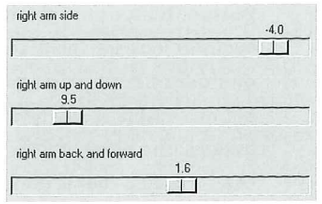
- **Button.** Buttons are the interaction designer's best friend. Once you begin to look for them, it's apparent that buttons are everywhere, all over our interfaces. In a word processing program, there are about 30 buttons visible at any given time. A mobile phone may have about 40 buttons: the number keys for dialing and a keyboard. A button is, at base, a switch that is pressed or clicked to activate it. The button can stay pressed (a **toggle button**), requiring another press to reset it (like most on/off buttons), or it can reset itself automatically (like keys on a keyboard). Buttons can be used for a wide variety of actions: from changing modes (from writing text to drawing, say) to moving an item or a cursor via arrow keys. Buttons can take many forms, from tiny icons to physical squares on a floor that can be stepped on. Buttons, however, are good only for simple actions.

- **Radio button.** Radio buttons enable users to choose from (often preset) items from a set. Typically, these are used to constrain selections, when only one answer is allowed ("What color hair do you have?" Black, Blonde, Red, Brown).



- **Dial.** Dials provide more control than buttons, allowing the user to select a setting along a continuum (such as the amount of heat on a stove's burner) or to choose between different settings or modes (such as the mode for taking pictures and the mode for viewing them on a digital camera. Dials can move freely, or simply turn from an established point to other established points on a wheel. These points are called detents. Some dials, like those often found on clothes driers, can be pushed in and pulled out, performing an action (such as turning on or off) that can vary based on the dial's rotation.

- **Latch.** A latch opens an otherwise tightly closed area. Latches are useful for keeping some areas or items hidden or safe until needed. They are good to use when a button or drop-down menu might be too easy to click or open. For example, latches are frequently used on handheld devices to keep the battery compartment safe.



- **Slider.** Sliders, like dials (although linear instead of round), are used for subtle control of a feature, often to control output (such as speaker volume) or the amount of data displayed (such as the number of houses on an interactive map). Sliders with more than one handle can be used to set a range within a range.

- **Handle.** A handle is simply a protruding part of an object that allows it to be moved or, in some cases, resized. Handles on the frames of most digital windows allow the windows to be moved around the screen or resized.

Physical-Only Controls

Some common controls are found only in the physical world and not on screens (although they can certainly manipulate objects on a screen).

- **Jog dial.** A jog dial is a type of dial that can be manipulated with a single finger, usually a thumb. It can be dial-like, or it can be a pad of buttons, typically used on small devices for moving a cursor or moving through menus. Jog dials are somewhat difficult to control, especially for young children and the elderly.

- **Joystick.** A joystick is a physical device typically used in digital gaming or in other applications that require rapid movement and intensive manipulation of remote physical or digital objects. Joysticks can move in any direction or can be constrained to move only left to right or only up and down.

- **Trackball.** A trackball is a physical device for manipulating a cursor or other digital or physical objects. Trackballs are typically in a stationary base, but the ball itself moves in any direction. A computer mouse is often a trackball in a case.

- **5-way.** A 5-way is a combination button and cursor. It generally moves a cursor on a screen in four directions (up/down, left/right) and has a button in the center in order to select what has been navigated to.

COURTESY PALM

Bill DeRouchey on Frameworks and Controls

Bill DeRouchey is a Senior Interaction Designer at Ziba Design. Bill has over 15 years of experience as a writer, information architect, product manager, coder, and interaction designer. He has designed a wide variety of products, from handheld satellite radios and medical devices to community Web sites, interactive spaces, and product architectures.

How do you go about choosing a structure or framework for your designs?

Most often, the directions that I explore are largely bounded by the physical constraints at the beginning of the client engagement. Many clients already have specific components selected for manufacturing before engaging with them, so I have to treat those as givens, specific display dimensions and resolutions being the most common example. When you're given a 160x128 pixel space to work within, that tends to inform your structure quite a bit.

Beyond that, my designs tend to follow a pattern of reminiscence. A new medical monitor needs to behave like clinicians are used to them behaving. New satellite radios need to convey reminiscent qualities of "radio" so that people have a basis from which they approach the device. It's all about giving people a head start for understanding how to interact with the new product in front of them. This allows it to better fit into their lives as seamlessly as possible.

You've written a lot about the history of the button. Why is that important?

Interaction design existed as an activity decades before it was explicitly named as one. Industrial designers applied knobs, switches, and buttons to their products, and determined how they would be used by consumers. So these products created a rich history of people interacting with technology long before computers entered our daily lives. These first decades of products paved the way and formed our expectations of interacting with products.

This is where the button gets interesting. Consider that our main concern as interaction designers today is how we interact with products. In the early days, the question was why we should interact with products at all? Convenience, luxury, efficiency and visions of the leisurely future were all used as aspirational triggers to buy blenders, washers, radios, and more. And all of this aspiration was communicated via imagery of fingers pushing buttons.

Bill DeRouchey on Frameworks and Controls *(continued)*

The phrase "push-button" itself meant easy, simple, even-you-can-use-this product. That's a lot of social burden placed on a single UI widget, which is why I love this story.

What should interaction designers know about controls?

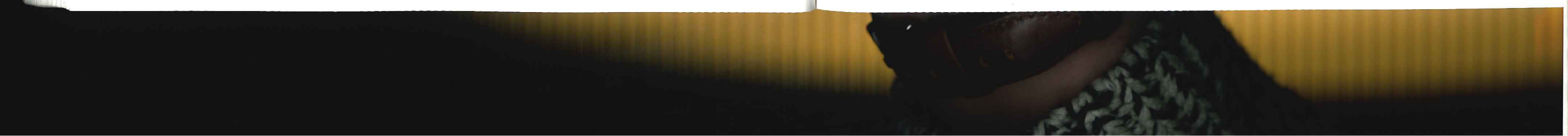
Physical controls have strong metaphors and history attached to them. Knobs and sliders typically indicate that you're looking for something vague along a spectrum: the right volume or temperature setting. Buttons and switches typically indicate a choice is being made. Turn the lights on. Start the microwave. Controls usually do only one thing.

Accordingly, one of the biggest challenges of controls is that space, size, and cost limit you for how many features are important enough to warrant their own physical controls. Do you really need to adjust bass levels that often, or do you bury that feature in another control somehow? Like all design, it's a delicate dance to determine this hierarchy, and the best way to solve it is to put prototypes in front of other people.

What are the most important things to remember when laying out controls?

Laying controls requires a strong sense of hierarchy, zoning, and priority. Control panels typically focus on a tight set of tasks, with one hero task in that set. In air conditioners, changing temperature is more important than adjusting schedules. In radios, adjusting volume is the hero task. These controls should be larger, offset, or otherwise designed to have the highest priority. It should be clear to people what is the single most important thing to do. Determine your hero task.

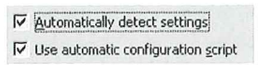
A common mistake is designing all the controls with an overly uniform look and feel. It may look clean to have 12 different buttons with uniform shape and color lined up into a grid, but that approach offers no quick visual appraisal to determine what control does what. In these situations, the labels become more important, creating a secondary problem.



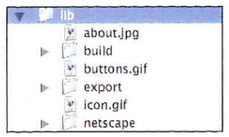
Digital-Only Controls

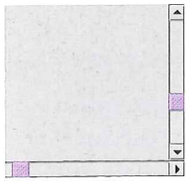
While many controls are found in both the physical, analog world and the digital one, some controls are only found on screens. These digital controls have grown from the original graphical user interface (GUI) vocabulary that was invented at Xerox PARC in the 1970s, reinvented in the 1980s in the Macintosh and PC operating systems, and added to and expanded by Web conventions in the 1990s:

- **Check box**. A check box enables users to select items from a short list.

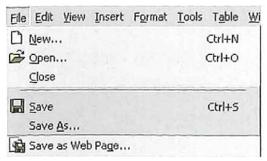


- **Twist.** Twists turn up or down, either revealing or hiding content or a menu in a panel.

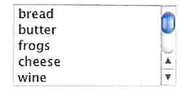




- **Scroll bar.** Scroll bars enable users to move content within a particular window or panel. Scroll bars can be vertical or horizontal. Scroll bars themselves can be manipulated via the cursor or buttons (for instance, by using arrow keys).
- **Drop-down menu.** Drop-down menus allow designers to cluster navigation, functionality, or content together without having to display it all at once. Drop-down menus can be displayed by rolling over them, or they can be opened with a click. They can retract after a selection has been made or the cursor rolls off them, though not necessarily.

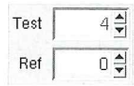


- **Multiple-selection list (or list box).** Multiple-selection lists enable users to select multiple items in a list.



- **Text box**. Text boxes enable users to enter numbers, letters, or symbols. They can be as small as (and constrained to) a single character or as large as the whole screen.

- **Spin box.** Spin boxes are text boxes with additional controls that enable users to manipulate what is inside the text box without having to type a value. They are good for suggesting values in what otherwise might be an ambiguous text box.

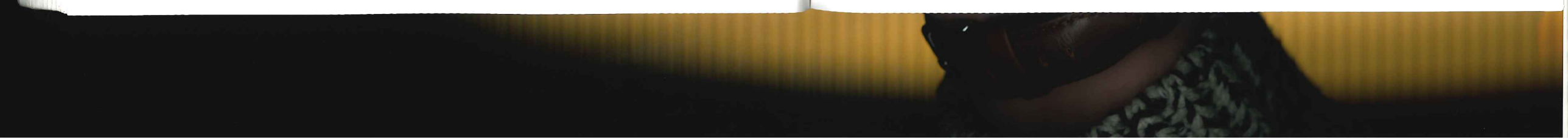


The combination of one (and usually more) controls plus the system response is called a **widget**. Widgets are the building blocks of any application or device. An MP3 player, for instance, is made of widgets: one for controlling volume, one for controlling the playing of music files, one for organizing files, one for exporting files, and so on. In each case, the user uses controls to perform an action, and the system responds. All applications and devices are made up of widgets.

Non-traditional Inputs

We are arriving at a time when keyboards, mice, and styluses aren't the only—and possibly not even the primary—way we interact with the digital world. With the dawn of ubiquitous computing, interactive environments, and sensor-enabled devices (see Chapter 9), people will engage with many different sorts of objects that have microprocessors and sensors built into them, from rooms to appliances to bicycles.

The controls for these faceless interfaces are the human body: our voices, our movements, and simply our presence.



COURTESY KERRY BODINE

Figure 7.13

The author screams at Kelly Dobson's Blendie, a voice-controlled blender, to get it to frappé.

Voice

Widespread implementation of **voice**-controlled systems has been on the horizon for at least a decade now. For now, voice-controlled interfaces are most prevalent (naturally) on phone systems and mobile phones. For example, people call their banks and perform transactions or dial their mobile phones with just their voices. Voice commands typically control limited functionality, and the device typically has to be ready to receive voice commands, either because it only functions via voice commands (as with automated phone systems and some voice-controlled devices—see **Figure 7.13**), or because it has been prepared to receive voice commands, as with mobile phones that allow voice-dialing.

Gestures

To most computers and devices, people consist of two things: hands and eyes. The rest of the human body is ignored. But as our devices gain more awareness of the movement of the human body through sensors such as cameras, the better able they will be to respond to the complete human body, including **gestures**. Devices like the Wii and the iPhone with their built-in accelerometers allow for all manner of new ways of controlling our devices via movements in space. See **Figure 7.14**.

COURTESY CANESTA AND KICKER STUDIO

Figure 7.14

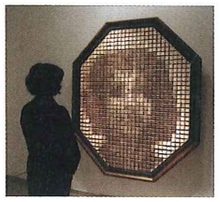
This gestural entertainment center uses a camera from Canesta to detect gestures in space that control the television.

Designers need to be especially aware of several issues when designing gestural interfaces:

- **Physiology and kinesiology.** Designers have to know how humans move and what the limitations are for that movement. For example, holding an arm out and making gestures can be quickly tiring—a condition known as "gorilla arm."
- **Presence and instruction.** Since there might be no visible interface—for example, consider the hands-free paper towel dispenser in many public restrooms—letting users know a gestural device is there and how to use it needs to be addressed.
- **Avoiding "false positives."** Since human beings make gestures all the time in the course of just moving around, designing and then detecting deliberate gestures can be challenging.
- **Matching gesture to task.** Without standard controls, figuring out the best motion to trigger an action is important. Simple gestures should be matched to simple tasks.

Presence

Some systems respond simply to a person's **presence**. Many interactive games and installations such as Daniel Rozin's "Wooden Mirror" (**Figure 7.15**) respond to a body's being near their sensors.



COURTESY DANIEL ROZIN

Figure 7.15

The "Wooden Mirror" creates the image of what is in front of it (seen by a camera) by flipping wooden blocks within its frame.

There are many design decisions to be made with presence-activated systems. Consider a room with sensors and environmental controls, for example. Does the system respond immediately when someone enters the room, turning on lights and climate-control systems, or does it pause for a few moments, in case someone was just passing through?

In addition, sometimes users may not want to be known to be present. Users may not want their activities and location known for any number of reasons, including personal safety and simple privacy. Designers will have to determine how and when a user can become "invisible" to presence-activated systems.

Summary

Refinement of design concepts is about making smart, deliberate choices about how the concept would work and could be built given the known constraints. It's about using the known laws of interaction design to guide design choices, and about putting in the right affordances and feedback so that users can create the right mental model of the product in order to properly use it.

Of course, right now, these are just documents; they don't live and breathe and you cannot really "interact" with them. For that, prototyping is necessary, and that is what the next chapter covers.

For Further Reading

About Face 3: The Essentials of Interaction Design, Alan Cooper, Robert Reimann, and David Cronin

The Design of Everyday Things, Donald A. Norman

Designing for the Digital Age: How to Create Human-Centered Products and Services, Kim Goodwin

Designing Interfaces: Patterns for Effective Interaction Design, Jenifer Tidwell

Designing Gestural Interfaces, Dan Saffer

Designing Web Interfaces: Principles and Patterns for Rich Interactions, Bill Scott and Theresa Neil

Mobile Interaction Design, Matt Jones and Gary Marsden

Communicating Design: Developing Web Site Documentation for Design and Planning, Dan Brown

Designing the Obvious: A Common Sense Approach to Web Application Design, Robert Hoekman Jr.

Information Architecture for the World Wide Web: Designing Large-Scale Web Sites, Louis Rosenfeld and Peter Morville

Ambient Findability: What We Find Changes Who We Become, Peter Morville

Content Strategy for the Web, Kristina Halvorson

8

Prototyping, Testing, and Development