Chapter 3

Thinking about Prototyping

This chapter explores two main topics—how a prototyping method supports (or interferes with) the process of creating an interface and the effects of paper prototyping on the people involved in producing, testing, and refining a design. You'll see the pros and cons of paper prototyping and learn how to compare it to whatever other method of prototyping you may be considering for your product. Other than paper, I don't cover specific prototyping methods here (in Chapter 12 I mention a few) because I think it's important to first think in terms of the concepts and benefits pertaining to prototyping. If a new prototyping tool comes along 2 years after this book, you should still be able to use the material in this chapter to evaluate it.

Creating an Interface—Look and Feel

Broadly speaking, you can think of an interface in terms of look and feel. The look comprises the screen layout, graphics, wording, and so on, and the feel is the behavior and processing. Look is pixels; feel is code. Both look and feel require work to produce, and the amount of effort varies depending on the method of prototyping. Let's examine look and feel in the context of designing a screen; for the sake of simplicity I'll choose an online order form.

The effort needed to create the form's look can be further divided into design time and rendering time. First you have to understand the purpose of the form, determine what fields it must contain, think about what order they should appear in, decide how they should be labeled, and so on. That's design time, and it's a process that happens inside your head, often with the aid of specs or other input. An order form is a relatively simple example; other screens are much more difficult to design. To get the order form out of your head and into a place where others can see it, you have to render it, either by hand or on a computer. In reality, design and rendering aren't discrete steps as I've described them here—they happen repeatedly and often simultaneously. But thinking of them separately will help clarify how a prototyping method supports the work.

The feel of an interface is, in essence, its programming—all the code that must be written to display the form, accept the user's inputs, process them, and produce some type of output. (Once you start thinking about how that output will appear to the user, then you're back to the look.) Depending on the development environment, some things are more arduous to code than others. Strictly speaking, programming also has design time and rendering time, although that distinction isn't as helpful for feel as it is for look, so I'm going to ignore it.

So we have three main activities to consider in prototyping—the time you spend

1. Designing—in other words thinking about what the user will experience
2. Rendering what the user will see
3. Coding the behavior

Now let's examine how paper and computer-based prototyping tools support each of these activities. You'll notice that each section has some questions to ask yourself while considering which prototyping method is best for you.

Designing

Design is probably the hardest activity of the three, and correspondingly it's hard for a prototyping method—any prototyping method—to support it very well. As sophisticated as computers are, they're a poor replacement for our creative problem-solving abilities, which is why interfaces are still designed and developed by humans. Prototyping methods focus mainly on the rendering and coding aspects of the process; they don't necessarily help us become better designers.

On the other hand, it's possible for a prototyping method to *constrain* the design process, as Alan Cooper eloquently explains (see Of Interest box that follows). One benefit of paper prototyping is that it imposes relatively few constraints on a design. The next chapter provides many examples of how to construct various interface widgets. For now, proceed under the premise that paper prototyping will let you make most anything you can envision.

Of Interest . . . Excerpt from "The Perils of Prototyping"

By Alan Cooper, Chairman & Founder
August/September 1994
Originally Published in Visual Basic Programmer's Journal;
available at ***www.cooper.com/articles/art_perils_of_prototyping.htm***

Software has a life of its own. It is full of little quirks and idiosyncrasies that influence how it is built. This is true whether the tools are high-level (VB or HyperCard) or low-level (asm, C). Some things are easy to build, and other things are hard to build. That this strongly influences the design is almost universally unrecognized. If you take the great risk of building something divergent from the tool's strength, you will quickly find yourself in a minefield of tactical choices: "Hmmm, let's see, if I code it this way the user presentation is better but then I have to build my own [giant-piece-of-reinvented-wheel], or I could code it the other way and have this operational conflict with this store-bought gizmo that I need to make this other part of the program easy to do, or I could roll my own and then it would be correct but I won't be able to test it for 6 months, or I could . . ." None of this has anything to do with software design!

Of course, any design process has constraints. Some things are hard to implement, and others are impractical or even impossible. A good designer doesn't ignore constraints, but on the other hand you don't want to choose a tool that introduces additional and unnecessary ones, especially early in the process when you're trying to be creative. When considering a prototyping method, ask yourself: *"What does this tool make easy (or hard) to prototype? Will this tool affect my creativity? How much will it distract me from thinking about the design?"*

Rendering

There are many ways to render an interface. You can draw it by hand or do it on a computer using Dreamweaver, Visual Basic, or any one of a myriad of tools. There are even tools like DENIM that let you *sketch* an interface on a computer by means of a drawing tablet, which may initially seem odd but as discussed in Chapter 12, it does have some benefits. There are plenty of choices here. In thinking about rendering, consider how much effort is needed with a particular method and how good the prototype needs to look. The following sections address these issues in more detail.

Effort to Render

The effort needed to render a prototype varies not only with the method but also with the skills of the individuals involved—drawing comes naturally to some people, whereas others may be proficient at creating interfaces with software. There are three factors to consider regarding the effort required to create a prototype and test it with users: creation, duplication, and modification.

Creation

How quickly can a screen be made the first time, starting from a blank slate? Some prototyping tools give you a leg up because they provide a library of widgets, or maybe you've even developed your own. Although it's possible for some people to mock up screens quite quickly using software, it's often possible to draw that same screen by hand in even less time. From what I've seen, I believe that drawing by hand is probably faster for many people, most of the time, but naturally there are exceptions.

Duplication

Does your interface have many similar screens (for example, several online product pages that all have the same layout)? How easy is it to make the variations? In software, this is pretty straightforward—you load a template, put in the unique parts, and do a Save As. With a hand-drawn paper prototype you can do the analogous process using a copy machine, so there may not be a significant difference here.

Modification

When you discover a reason to change a screen, how quickly can you tweak it? If this happens in a usability test, can you do it while the user is still present and thus get immediate feedback? An interface doesn't just get designed, it gets redesigned as well, often many times before it's done. With a paper prototype, you can make changes very quickly, even during a usability test. If a term is confusing, you can cross it out and try a different one. You can write an example next to a confusing edit field. You can write a sentence of explanation about the next step in a process. Simple changes like these can solve many usability problems.

I once saw a developer (let's call him Ed) make changes on the fly to a working Web application. During the usability tests, Ed sat in the room along with the other observers. I thought he was just using his laptop to take notes—little did I know that he had a connection to the development server. Along came a task where the users became confused and clicked the wrong link. During the post-task discussion, Ed asked the users to click Back and then Refresh. Voila! Up came a new version of the page that solved the problem. Although Ed's real-time interface changes were entertaining as well as successful, it's rare that this is feasible for developers to do with a software prototype.

With some software prototyping tools it may also be possible to make quick changes, but think carefully about the logistics of how this would work in a usability test. When the user finds a problem and you want to make a change, you might have to grab the keyboard and mouse from the user, get out of the running prototype and into the prototyping tool, make the change, and then get back into the same spot in the interface (which may have been reached after a sequence of actions) to see whether your fix works. One advantage of a paper prototype is that you can make the change without losing the user's context.

How Good Should It Look?

This is a complex question, so I'm providing only a partial answer here and will tackle other parts of the answer elsewhere in this book. The short answer is that a prototype has to look (and act) realistic enough to elicit feedback for the issues you're most worried about, but it doesn't need to be any better than that. Chapter 12 goes into detail about the kinds of problems that paper prototypes are good and not so good at finding. Chapter 13 presents evidence that paper prototypes seem to reveal about as many (and as severe) problems as you'd get by testing the real thing; as you saw in the previous chapter, there are plenty of real-life examples to back this up.

One drawback of a prototyping tool that gives you several options for specifying the appearance (fonts, colors, shading, and so on) is that it's easy to get caught up in tweaking the design to make it look nice. If you end up changing it later (and has there ever been a design that didn't get changed?) then your tweaking time will have been wasted. A sketched prototype helps you avoid the temptation to perfect the appearance because it isn't *supposed* to look finished. Time is money, so you want to weigh whatever effort you're putting into a prototype against the benefit you'll get out of it.

Coding

I once saw a demo of a tool that was touted as a fast and easy way to create interfaces. The guy conducting the demo first put several controls on the screen in just a minute or two. Then he made an offhand comment about "linking in some prewritten code" and jumped to a completed version of the interface. Being a software engineer at the time, I immediately thought, "But wait—he's just skipped over the hard part!"

Coding the behavior of an interface (and the underlying databases and processing required to produce its outputs) is where the bulk of the work lies—90%, according to Joel Spolsky, a programmer who writes a column on software development at *www.joelonsoftware.com.* I don't want to get hung up on the exact percentage because it varies depending on the type of interface and also on the tool—a software application may have very complex underlying code, whereas many Web pages have little or none. But it's a reasonable generalization that most interfaces have substantial code beneath them.

In a paper prototype, the coding effort is always zero.

Joel explained to me why creating a working software prototype can actually take longer than making the real thing. The problem is that you have to either reconstruct code or leave it out. If you leave it out, you're very restricted in the kinds of tasks you can ask users to do, and thus you may not get good feedback about the new functionality. If you reconstruct it, however, your software prototype can take longer than if you'd just coded it into the real product. Joel offers this example:

> Several years ago at Microsoft, the developers wanted to explore the usefulness and feasibility of using drag & drop to move cells in Excel. They had a couple of interns work all summer to make a prototype of Excel to support this functionality. Trouble was, to actually test it they needed to implement a healthy chunk of the spreadsheet functionality so people could perform realistic tasks—if they'd just done the drag & drop part, they wouldn't have gotten to see which

> method people naturally chose in different situations. So they ended up re-creating a good chunk of Excel in order to make their prototype functional. When they usability tested it, they found that drag & drop was indeed a useful thing for users, so they put it in the product. It took the staff programmer about 1 week to implement in the product, as compared to the interns who worked all summer on their prototype. The irony of this is that the whole purpose of a prototype is to save time.

Joel also notes that you could always put new functionality into your code and take it out later if it isn't what users need. But ripping out code can introduce bugs, not to mention that destroying one's own work is a pretty unsatisfying thing to do.

As prototyping tools become more sophisticated, they'll grow in their ability to automate the behavior of certain interface widgets—put a Submit button on a form, and it'll automatically do the right thing. But a prototyping tool that could automate everything would in essence replace programmers, and I don't think that will ever happen. In a paper prototype, a human simulates what the code would do. Thus, *the coding effort is always zero.* No matter how sophisticated the prototyping tool or how proficient one is with it, you can't get any faster than that. This is an important benefit of a paper prototype, particularly for complex software—no code to write or (sometimes even more important) rewrite until the design has stabilized. The flip side of that coin is that at some point you do have to start writing code, and paper prototypes don't help with that.

To assess the importance of the coding factor for your product, ask yourself: *"How much code is required to support everything the user sees? How much of this work does the tool do for me? How much of a pain will it be to change that code while the design is still changing? Do we plan to evolve our prototype into the finished product, or is it okay to throw the prototype away when the real coding begins?"*

Don't forget the data. As Chapter 7 explains, if you plan to conduct usability tests of your design, users will need to see realistic data to perform meaningful tasks. For example, if you want users to look up a stock quote using your mobile phone interface, you'll need some way to make that stock quote appear on the display so that users will know when they've found the answer. Depending on where the interface's data comes from, this may be easy or hard. Sometimes in an electronic prototype it's necessary to hard-code the data or create a test database; it may be easier to use a paper prototype instead because you can simply write in whatever data you need. So consider the following: *"Where does the data come from? How easy will it be to have the prototype display the realistic data needed for usability testing?"*

Finding Technical Issues

Obviously, paper prototypes don't show you whether a design can be built (of course, other prototyping methods may not either, unless they use the same development environment in which you're planning to implement). Paper prototypes aren't constrained by existing architecture, databases, bandwidth, legal considerations, business policies, the operating system, accessibility concerns, and so on. In real life, developers must consider all these constraints and more. You don't want to create a prototype that's impossible to implement.

Interestingly, even though there is no code involved in a paper prototype, it can sometimes help the development team anticipate technical problems. Although software engineers naturally tend to think in terms of database schema, software architecture, communication protocols, and the like, sometimes they get technological tunnel vision and may overlook the implications that a particular user requirement has for the design.

Here's an example from my own experience:

> In the mid-1980s I was a project manager for the development of some internal tools for my company's field technicians. We had spent several weeks hashing out database schema, processing, and all sorts of technical innards. There were also to be some reports available to the user, but we'd left them until last under the assumption that they'd be easy—reports simply take information from database records and format it nicely, so what could go wrong with that?
>
> But then the engineer assigned to program the report modules walked into my office scratching his head. He couldn't figure out where Report X was supposed to get its data. We quickly realized that, in fact, Report X was impossible to produce with the databases we'd designed. D'oh! It took our five-person team about 2 weeks to revise the databases (and the code that had already been built on them, as well as the specs) to recover from this snafu.

Would paper prototyping have found this problem? Although I can't rewrite history, I believe the answer is yes. In paper prototyping, you first create a realistic set of tasks and then build the prototype to support them, which forces you to visualize the entire process of what users will be doing. If we had done usability testing on this project, we would have included the reports because users needed them to see the results of their work. The problem with Report X was blindingly obvious the first time someone looked at it—mocking up Report X would have put it on our radar screen before we'd spent all that time developing a database schema that was incompatible with it.

Strictly speaking, paper prototyping doesn't deserve all the credit for unearthing this type of problem—it's really walking through the tasks that allows us to see an interface from a user's perspective. The paper prototype simply makes it possible for this to happen earlier in the development process, when it's less painful to make changes. In working with many product teams over the years, I've seen that it's common for technical questions about what can and cannot be coded to arise from the process of constructing a paper prototype and using it to walk through tasks.

Now let's switch our focus from prototyping tools and interface development and start talking about the people—first the users, then the product team.

Psychological Benefits for Users

Suppose you had a software prototyping tool that could read your mind—you simply plunk down all the interface elements and everything would auto-magically behave the way you wanted, without any coding, so you could test it on a computer. Even if you could do this, you still wouldn't have the advantages of paper prototyping described in this section. It turns out that there are some interesting psychological effects in terms of the way users respond to an unfinished design that's shown to them on paper.

Less Intimidating Than a Computer

Many people, especially younger ones, take computers for granted—my niece was proficient with a mouse at age 3. But there is still—and will continue to be for decades—a substantial percentage of adults who are not yet comfortable with computers. When faced with technology they find intimidating, there's an increased risk that users will feel foolish when they can't figure something out during a usability test. In contrast, with a paper prototype there is no computer to contend with, just human beings. This can help some technophobes relax, so keep this in mind if your target audience includes people who lack confidence in working with computers. (One could make the opposite argument that some people may be more comfortable interfacing with computers than with humans, so again it depends on who your audience is.)

One might wonder about the opposite situation—if your users are technically savvy, does paper prototyping seem silly to them? I've tested several paper prototypes of sophisticated scientific and engineering applications, and in my experience the answer is no. Techie types such as network administrators and software developers readily understand that the paper prototypes are being used to get their feedback before the design is cast in concrete. I've yet to see one scoff at a paper prototype.

More Creative Feedback

An unfinished design seems to encourage a more creative response from reviewers. In a paper by Schumann, Strothotte, Laser, and Raab (1996), the researchers compared the methods used by architects to present building concepts to their clients. The researchers found that many architects preferred to show their clients sketches during the early stages of design, believing that this encouraged more discussion about the building and its intended use. Most architects reserved more accurate representations such as CAD drawings for the later stages of design. (Strangely enough, the researchers were working on a tool that would take a CAD drawing and produce from it something that *looked* like a sketch—a seeming step backward that was done deliberately for the purpose of encouraging creative discussion.)

Unfinished designs can have a dramatic effect on how stakeholders become involved with the project, as illustrated by the story in the From the Field box on p. 59. Instead of being a passive viewer, a person presented with an unfinished design becomes more creatively engaged in thinking about the concepts and functionality. I've seen this happen many times. If you show users a slick-looking interface and ask for feedback, they may subconsciously think, "These guys have obviously put a lot of thought into this, so I'll keep my hare-brained suggestion to myself." But with a paper prototype, it's obvious that the design is literally still "on the drawing board" and that changes can be made very easily, sometimes even by users themselves.

No Nitpicky Feedback

A polished-looking prototype can encourage low-level feedback about the visual aspects of the design. When something appears to be finished, minor flaws stand out and will catch the user's attention. To put it another way, people nitpick. Unless you're in the later stages of development or are specifically asking for feedback on the visual design, it isn't especially useful to hear comments such as "Those fields don't line up" or "I don't like that shade of green." I've known many

From the Field: Rough Prototypes Encourage Feedback

"My company researches and develops intelligent autonomous agents and agent-based applications, generally for the United States government. I was about to present a set of preliminary designs to a government research technical review committee. Previous review meetings of this nature had left the committee with a negative view of our progress and our willingness to involve their ideas.

"The project leader and I had initially discussed taking my paper prototype drawings and either redrafting them or implementing mock-ups (in VB or Director) of them. Our goals were to (1) reinforce that these designs were preliminary, (2) encourage a dialogue with the committee about the direction of the project, and (3) get the reviewers excited about being part of the design process and to invest some of their considerable energy and expertise into the project. After thinking about what we hoped to get out of the review presentation, we decided to stick with the rough drawings instead of something more polished.

"This approach was a huge success. The reviewers got very excited during the presentation and constantly interrupted me with suggestions and meaningful critiques. Afterward we got additional positive comments and ideas—something that had rarely happened with past projects. Committee members specifically mentioned that the sketchy nature of drawings made them feel included in the process.

"Three months later we presented an implemented demonstration of the application that included a number of changes suggested by the review committee and a presentation that showed the now-familiar design sketches next to the implemented screens. The demonstration was even a larger success than the prior meeting. The committee members felt like part of the team and were excited by seeing their ideas come to life. In addition, seeing the rough design sketches presented next to the polished screen shots left them impressed by the large degree of progress made. They left excited about the future of the project. Again, this high level of collaboration between the research team and the review committee had been rare in the past and bodes very well for both the quality of the final deliverable and for future funding for the research project."

Jack Zaientz, Soar Technology

developers who ran afoul of this problem when using software prototypes in an effort to get feedback about functionality—the Of Interest box on p. 61 is a great example. Hand-drawn paper prototypes avoid the nitpicky feedback because it's obvious that you haven't specified the look yet. This encourages users to focus on the concepts and functionality instead.

My colleague Ann Marie McKinnon, a graphic designer, has experienced something similar to Joel's Corollary Two—if it looks too pretty, people can become too attached to it. After she showed a nicely designed set of screens to some key stakeholders, Ann Marie reported that they "fell in love" with that particular version of the interface because it looked so nice. They subsequently resisted making any changes to the functionality, even when the rest of the team found a clear need to do so.

Effects on the Product Team

Perhaps even more significant than the users' reactions are the effects that paper prototyping has on the product team's mindset and the ways in which they work together.

Minimizes the Invested Effort

The more effort you've put into creating something, the harder it is to accept that it needs to be changed. Say that you're a developer who has spent 2 weeks creating a prototype based on the requirements spec you got from Marketing. Upon seeing it in action, Marketing decides they want something different. What's your reaction?

a. Oh my yes, your way is obviously better. No problem, I'll change it.
b. But I did it this way because . . .
c. Sure, I'd be happy to waste another 2 weeks of my life coding something else you don't want.
d. You want changes? Step outside and I'll make some to your face.

Unless you're a saint or a psychopath, you probably answered b, maybe c on a really bad day. It's a natural reaction to defend one's hard work to avoid the need to redo it. I've also heard, and I'll confess to having done this myself, that a developer

Of Interest . . . Excerpt from "The Iceberg Secret"

by Joel Spolsky
February 13, 2002
*Available at **www.joelonsoftware.com**.*

You know how an iceberg is 90% underwater? Well, most software is like that too—there's a pretty user interface that takes about 10% of the work, and then 90% of the programming work is under the covers. And if you take into account the fact that about half of your time is spent fixing bugs, the UI only takes 5% of the work. And if you limit yourself to the *visual* part of the UI, the pixels, what you would see in PowerPoint, now we're talking less than 1%.

That's not the secret. The secret is that *People Who Aren't Programmers Do Not Understand This.* There are some very, very important corollaries to the Iceberg Secret.

Important Corollary One

If you show a nonprogrammer a screen which has a user interface that is 90% worse, they will think that the program is 90% worse.

I learned this lesson as a consultant, when I did a demo of a major web-based project for a client's executive team. The project was almost 100% code complete. We were still waiting for the graphic designer to choose fonts and colors and draw the cool 3-D tabs. In the meantime, we just used plain fonts and black and white, there was a bunch of ugly wasted space on the screen, basically it didn't look very good at all. But 100% of the functionality was there and was doing some pretty amazing stuff.

What happened during the demo? The clients spent the *entire meeting* griping about the graphical appearance of the screen. They weren't even talking about the UI. Just the graphical appearance. *"It just doesn't look slick,"* complained their project manager. That's all they could think about. We couldn't get them to think about the actual functionality. Obviously fixing the graphic design took about one day. It was almost as if they thought they had hired *painters*.

Important Corollary Two

If you show a nonprogrammer a screen which has a user interface which is 100% beautiful, they will think the program is almost done.

People who aren't programmers are just looking at the screen and seeing some pixels. And if the pixels look like they make up a program which does something, they think "oh, gosh, how much harder could it be to make it *actually work?*"

The big risk here is that if you mock up the UI first, presumably so you can get some conversations going with the customer, then everybody's going to think you're almost done. And then when you spend the next year working "under the covers," so to speak, nobody will really see what you're doing and they'll think it's nothing.

Yeah, I guess I can change it.
Two more weeks of my life, wasted.
Ritti

It's hard to change something if you've invested a lot of effort in it. (Illustration by Rene Rittiner.)

who agrees to revise the interface may quietly postpone this effort under the (sometimes correct) assumption that in a couple of weeks Marketing will again ask for something completely different.

But with a paper prototype, the effort that went into its creation is measured in minutes or hours, rather than days or weeks. The feeling of ownership is proportionally less, as is the knee-jerk reaction to defend the design and resist changing it. If you've spent 15 minutes drafting a screen and the first user finds a huge problem with it, it's relatively easy to throw it away and try something different.

More Creativity

Veteran usability specialist Mary Beth Rettger describes an interesting phenomenon regarding how paper prototypes can result in design revolution, not just evolution. The MathWorks makes extensive use of paper prototypes during development. Mary Beth has noticed that sometimes a developer will test a paper prototype and find some minor problems—nothing too serious. They'll start to tweak the interface, then scrap it in favor of an approach that's very different and much better. It's hard to articulate, but amazing to watch. She says, "It's as if they turn the design inside out and upside down. Or keep one piece and change everything around it. It's almost organic in nature." Because paper prototypes are a flex-

Mary Beth conjectures that they facilitate the kind of quantum design leaps that lead to greatly improved usability.

Multidisciplinary Teams Can Participate

Interfaces aren't created by interface designers or programmers alone; there's a whole cast of characters whose insights need to be funneled into the design. Customer support reps often have a good grasp of what customers find confusing, and so do trainers. Technical writers can provide a "distant early warning" about usability problems when they run into something that's hard to document. (When I was a software project manager, I learned that when the tech writer wandered into my office with a puzzled look asking, "Can you explain to me again how this six-level alarm prioritization thing works?" it was usually a clue that we'd over-designed something.) And if the people in Sales and Marketing have done their homework, they'll have a lot of data about what the target market wants, and why.

Expertise in interface design or software development is not necessary to participate in the creation of a paper prototype. Thus, it's a technique where members from different disciplines can collaborate, including writers, marketers, trainers, customer service reps, and so on. Anyone who has knowledge of the subject area or the users can contribute directly to the design. In particular, technical writers are often valuable contributors to a paper prototype—they can get right in there and wordsmith the *interface,* rather than having to explain it in the manual or help.

I'm not implying that good design can be done by nondesigners or that the designers responsible for an interface should abdicate this responsibility in favor of design by committee. (Chapter 7 talks more about the dynamics of paper prototype creation.) All I'm saying is that paper prototyping facilitates the process of incorporating many people's ideas into a design, and usually that's a good thing.

Earlier Communication across Disciplines

Software engineers write code. Interface designers create screens. Technical writers draft help and manuals. Marketers research market segments, find opportunities, and create brands. Graphic designers create effective layouts and visuals that support the brand. Last but not least, the users themselves are going about their daily lives, unaware that you're working on something they're going to love (or hate). The trouble is, in the early stages of product development these activities are very separate. There's little means of communicating in a useful and collaborative way about the product. Paper prototyping alleviates the problem because

Paper prototyping is also a great way for new team members to get up to speed on the product and how it works. For example, instead of the technical writer going to the designer for an explanation of the interface, the writer can participate in the creation and testing of the prototype. The same end result—but less pain and suffering to get there.

From the Field: Fostering a Common Vision

"Our development team at RioPort is divided between California and Colorado. Typically when teams are remote they tend to see things differently—sometimes there's a communication breakdown when one person can't grasp what someone else is verbalizing. There are general issues everyone can see, but it's easy to get a myopic view when you don't have everyone in the same room, looking at the same white board.

"In the summer of 2001, the Colorado developers went to the west coast and both groups worked on the paper prototype for what eventually became our PulseOne Media Service. Walking through the flow of screens as a team before bringing in users was helpful in giving us a common understanding of the interface. When we did usability tests, it was valuable having customers echo the same issues that some of us had been concerned about, and to see their discovery and exploration process. Actually observing participants is a much richer form of communication compared to reading a report, plus we were able to ask them questions directly. This was important because our interface supported a model of online music distribution that was new to people, and we wanted to hear their impressions of it.

"As a result of the usability tests, we gained a common understanding of problems and areas that needed additional attention. The paper prototyping exercise allowed us to do this earlier in the project cycle; this accelerated communication and eliminated misunderstandings and rework that otherwise would have occurred later."

Tony Schaller, Sr. Vice President
Technology and CTO at RioPort, Inc.

Avoiding Miscommunication (The "Amelia Bedelia" Problem)

Miscommunication between any two people is inevitable at some point in their relationship. For people who come from different professions with different jargon, it can be difficult to establish a common vision of the product and how it serves its target user population.

For a developer, it's frustrating to implement something that wasn't quite what Marketing had in mind. I call this the "Amelia Bedelia" problem after a funny series of children's books about a well-meaning maid who follows her employers' instructions literally, with predictably disastrous results. It's not so funny when you've spent days or months developing the wrong thing. Interface specs—even with screen shots—don't solve the Amelia Bedelia problem because they don't

"My dress!" exclaimed Mrs. Rogers.

"It's full of holes."

"Yes, ma'am, I removed

every single spot,"

said Amelia Bedelia.

Like the children's book character Amelia Bedelia, we sometimes misinterpret what our customers are asking for. (From *Thank You, Amelia Bedelia* by Peggy Parish. New York: HarperTrophy, 1995. Reprinted by permission.)

show the behavior of the system and they also don't show you what people expect the interface to do (especially nontechnical people because they have a harder time inferring functionality). But when you put a paper prototype in front of someone and watch them use it, you'll quickly discover whether it meets their expectations. It's a good idea to get Marketing (or whoever collected the requirements and wrote the early project documents) involved in prototyping sessions and make sure they attend the usability tests.

Opinion Wars

Every development team has its opinion wars—those endless meetings where arguments go round and round about the best way for users to do something. Many opinions are based on assumptions, and particularly nasty disagreements are often based on *unspoken* assumptions. Or sometimes there's simply a lack of data—in the words of philosopher Bertrand Russell, "The most savage controversies are those about matters as to which there is no good evidence either way."

I once stopped two developers who were on the verge of a fistfight over whether there should be an Update button on a form. The interface in question was a client-server software application used by customer service reps (CSRs) while on the phone with customers. This particular form showed the customer record, including data such as address and telephone number.

	Developer A: "There must be an Update button!"	*Developer B: "No Update button, you idiot!"*
Unspoken assumption	Data integrity is paramount. If the CSR accidentally types into a field, customer data could be overwritten and lost.	Speed of customer service is paramount. If the CSR has to click an Update button and the system is slow to respond, the customer will get impatient.
Conclusion	Make CSRs click an Update button first if they need to make changes.	Let CSRs make changes directly to the customer record.

The fascinating thing about this disagreement was that neither developer was saying a word about their unspoken assumptions, they were simply battering each other with their conclusions. Once I started asking them *why* they held their position, then the assumptions came out. Now we had some tangible questions to research: How often did records get overwritten in the current system? How likely were delays in bringing up the form for editing? Did the slowness of the system cause trouble for the CSRs? (Unfortunately, none of these questions got answered because the company was bought and the entire development team was laid off 2 days later.)

Strictly speaking, any kind of usability testing gives you a way to discover whether your assumptions about the design are true or false. Paper prototypes let you get this information early, before you've invested effort in coding. When a question arises, the team can say, "We're testing the prototype next week, so we can ask some users." Then it magically becomes an issue for research, not argument.

Summary

Although paper prototyping isn't a perfect technique, it's useful in a wide variety of situations. Its main benefits are that it:

- Is fast and inexpensive
- Identifies problems before they're coded
- Elicits more and better (i.e., less nitpicky) feedback from users
- Helps developers think creatively
- Gets users and other stakeholders involved early in the process
- Fosters teamwork and communication
- Avoids opinion wars

Chapters 12 and 13 go into more detail about the kinds of things that paper prototyping is and isn't good for, as well as how people react to it.