Chapter 1. Designing for People

This book is almost entirely about the look and behavior of applications, web apps, and interactive devices. But this first chapter is the exception to the rule. No screenshots here; no layouts, no navigation, no diagrams, no visuals at all.

Why not? After all, that's probably why you picked up the book in the first place.

It's because good interface design doesn't start with pictures. It starts with an understanding of people: what they're like, why they use a given piece of software, and how they might interact with it. The more you know about them, and the more you empathize with them, the more effectively you can design for them. Software, after all, is merely a means to an end for the people who use it. The better you satisfy those ends, the happier those users will be.

A framework for achieving this is described here. It covers four areas. These are not strict rules or requirements for creating great designs. But having a plan for how you will inform yourself and your team in each area will give you confidence that your work is based on real insights into valuable problems to solve for your target customers. Decide for yourself what level of time and effort is right for your project or company. Revisiting these areas regularly will keep key insights

top of mind and help focus everyone's effort, especially UI design, on creating great outcomes for people.

The four part structure for understanding design for people is this:

1. Context: Who is your audience?
2. Goals: What are they trying to do?
3. Research: Ways to understand context and goals
4. The Patterns: Cognition and behavior related to interface design

Context

Know Your Audience

There's a maxim in the field of interface design: "You are not the user!"

So, this chapter will talk about people. It covers a few fundamental ideas briefly in this introduction, and then discusses some patterns that differ from those in the rest of the book. They describe human behaviors—as opposed to system behaviors—that the software you design may need to support. Software that supports these human behaviors better helps users achieve their goals.

Interactions are conversations

Each time someone uses an application, or any digital product, he carries on a conversation with the machine.

It may be literal, as with a command line or phone menu, or tacit, like the “conversation” an artist has with her paints and canvas—the give and take between the craftsperson and the thing being built. With social software, it may even be a conversation by proxy. Whatever the case, the user interface mediates that conversation, helping users achieve whatever ends they had in mind.

The key points are these:

- There are two participants in the conversation, the person and the software.
- There is a constant, back and forth exchange of information.
- The exchange is a series of requests, commands, reception, processing and responses.

The human in the conversation needs continuous feedback from the interface that confirms that things are working normally, inputs are being processed, and that she is proceeding satisfactorily towards the goal of the moment

For this feedback loop to work, the software—which can’t be as spontaneous and responsive as a real human (at least not yet)—should be designed to mimic a conversation partner. It should be understandable to its partner, it should indicate it’s active, when it’s “listening,” and it should be obvious when it’s responding. Another layer on this is having some

anticipated next steps or recommendations, in the same way a considerate person might be helpful to another.

As the user interface designer, then, you get to script that conversation, or at least define its terms. And if you're going to script a conversation, you should understand the human's side as well as possible. What are the user's motives and intentions? What "vocabulary" of words, icons, and gestures does the user expect to employ? How can the application set expectations appropriately for the user? How do the user and the machine finally end up communicating meaning to each other?

Match your content and functionality to your audience

Before you start the design process, consider your overall approach. Think about how you might design the interface's overall interaction style—its personality, if you will.

When you carry on a conversation with someone about a given subject, you adjust what you say according to your understanding of the other person. You might consider how much he cares about the subject, how much he already knows about it, how receptive he is to learning from you, and whether he's even interested in the conversation in the first place. If you get any of that wrong, bad things happen—he might feel patronized, uninterested, impatient, or utterly baffled.

This analogy leads to some obvious design advice. The subject-specific vocabulary you use in your interface, for instance, should match your users' level of knowledge; if some users won't know that vocabulary, give them a way to learn the unfamiliar terms. If they don't know computers very well, don't make them use sophisticated widgetry or uncommon interface-design conventions. If their level of interest might be low, respect that, and don't ask for too much effort for too little reward.

Some of these concerns permeate the whole interface design in subtle ways. For example, do your users expect a short, tightly focused exchange about something very specific, or do they prefer a conversation that's more of a free-ranging exploration? In other words, how much openness is there in the interface? Too little, and your users feel trapped and unsatisfied; too much, and they stand there paralyzed, not knowing what to do next, unprepared for that level of interaction.

Therefore, you need to choose how much freedom your users have to act arbitrarily. At one end of the scale might be a software installation wizard: the user is carried through it with no opportunity to use anything other than Next, Previous, or Cancel. It's tightly focused and specific, but quite efficient—and satisfying, to the extent that it works and is quick. At the other end might be an application such as Excel, an "open floorplan" interface that exposes a huge number of features in one place. At any given time, the user has hundreds of things

that he could do next, but that's considered good, because self-directed, skilled users can do a lot with that interface. Again, it's satisfying, but for entirely different reasons.

Skill Level

How well can your audience(s) use your interface now and how much effort are your users willing to spend to learn it?

Some of your customers may use it every day on the job—clearly they'd become an expert user over time. But they will become increasingly unhappy with even small dissatisfactions. Maybe they'll use it sometimes, and learn it only well enough to get by ("Satisficing"). Difficulties in usage may be tolerated more. Maybe they'll only use it once. Be honest: can you expect most users to become intermediates or experts, or will most users remain perpetual beginners?

Software designed for intermediate-to-expert users includes:

- Photoshop
- Excel
- Code development environments
- System-administration tools for web servers

In contrast, here are some things designed for occasional users:

- Kiosks in tourist centers or museums
- Windows or Mac OS controls for setting desktop backgrounds
- Purchase pages for online stores
- Installation wizards
- Automated teller machines

The differences between the two groups are dramatic. Assumptions about users' tool knowledge permeate these interfaces, showing up in their screen-space usage, labeling, and widget sophistication, and in the places where help is (or isn't) offered.

The applications in the first group have lots of complex functionality, but they don't generally walk the user through tasks step by step. They assume users already know what to do, and they optimize for efficient operation, not learnability; they tend to be document-centered or list-driven (with a few being command-line applications). They often have entire books and courses written about them. Their learning curves are steep.

The applications in the second group are the opposite: restrained in functionality but helpful about explaining it along the way. They present simplified interfaces, assuming no prior knowledge of document- or list-centered application styles (e.g., menu bars, multiple selection, etc.). "Wizard" frequently show up, removing attention-focusing responsibility from the user. The key

is that users aren't motivated to work hard at learning these applications—it's usually just not worth it!

Now that you've seen the extremes, look at the applications in the middle of the continuum:

- Microsoft PowerPoint
- Email clients
- Facebook
- Blog-writing tools

The truth is that most applications fall into this middle ground. They need to serve people on both ends adequately—to help new users learn the tool (and satisfy their need for instant gratification), while enabling frequent-user intermediates to get things done smoothly. Their designers probably knew that people wouldn't take a three-day course to learn an email client. Yet the interfaces hold up under repeated usage. People quickly learn the basics, reach a proficiency level that satisfies them, and don't bother learning more until they are motivated to do so for specific purposes.

You may someday find yourself in tension between the two ends of this spectrum. Naturally you want people to be able to use your design "out of the box," but you might also want to support frequent or expert users as much as possible. Find a balance that works for your situation. Organizational patterns in Chapter 2, such as "Multi-Level Help", can help you serve both constituencies.

Goals: Your interface is just a means to their ends

Everyone who uses a tool—software or otherwise—has a reason for using it. These are their goals. Goals could be outcomes such as:

- Finding some fact or object
- Learning something
- Performing a transaction
- Controlling or monitoring something
- Creating something
- Conversing with other people
- Being entertained

Well-known idioms, user behaviors, and design patterns can support each of these abstract goals. User experience designers have learned, for example, how to help people search through vast amounts of online information for specific facts. They've learned how to present tasks so that it's easy to walk through them. They're learning ways to support the building of documents, illustrations, and code.

Ask Why

The first step in designing an interface is to figure out what its users are really trying to accomplish. Filling out a form, for example, is almost never a goal in and of

itself—people only do it because they're trying to buy something online, renew their driver's license, or install software. They're performing some kind of transaction.

Asking the right questions can help you connect user goals to the design process. Users and clients typically speak to you in terms of desired features and solutions, not of needs and problems. When a user or client tells you he wants a certain feature, ask why he wants it—determine his immediate goal. Then to the answer of this question, ask "why" again. And again. Keep asking until you move well beyond the boundaries of the immediate design problem.[1]

Design's value: Solve the right problem, then solve it right

Why should you ask these questions if you have clear requirements? Because if you love designing things, it's easy to get caught up in an interesting interface design problem. Maybe you're good at building forms that ask for just the right information, with the right controls, all laid out nicely. But the real art of interface design lies in solving the right problem, defined as helping the user achieve their goal.

So, don't get too fond of designing that form. If there's any way to finish the transaction without making the user go through that form at all, get rid of it altogether. That gets the user closer to his goal, with less time and effort spent on his part (and maybe yours, too).

Let’s use the “why” approach to dig a little deeper into some typical design scenarios.

Why does a mid-level manager use an email client? Yes, of course—”to read email.” Why does she read and send email in the first place? To converse with other people. Of course, other means might achieve the same ends: the phone, a hallway conversation, a formal document. But apparently, email fills some needs that the other methods don’t. What are they, and why are they important to her? The convenience of choosing when to send or respond? Privacy? The ability to archive a conversation? Social convention? What else?

A father goes to an online travel agent, types in the city where his family will be taking a summer vacation, and tries to find plane ticket prices on various dates. He’s learning from what he finds, but his goal isn’t just to browse and explore different options. Ask why. His goal is actually a transaction: to buy plane tickets. Again, he could have done that at many different websites, or over the phone with a live travel agent. How is this site better than those other options? Is it faster? Friendlier? More likely to find a better deal?

Sometimes goal analysis really isn’t enough. A snowboarding site might provide information (for learning), an online store (for transactions), and a set of Flash movies (for entertainment). Let’s say someone visits the site for a purchase, but she gets sidetracked into the information on snowboarding tricks—she has switched goals from accomplishing a transaction to

browsing and learning. Maybe she'll go back to purchasing something, maybe not. And does the lifestyle and entertainment part of the site successfully entertain both the 12-year-old and the 35-year-old? Will the 35-year-old go elsewhere to buy his new board if he doesn't feel at home there, or does he not care? It's useful to expand your goal framework to include an understanding of the specific business purchase cycle. Your snowboarding customer will have different goals at different stages of this cycle. Alternately, you may want to design how you could foster a long term loyalty between the brand and the customer. This could be done via content and functionality that fosters an identity, builds a community, and celebrates a lifestyle.

It's deceptively easy to model users as a single faceless entity—"The User"—walking through a set of simple use cases, with one task-oriented goal in mind. But that won't necessarily reflect your users' reality.

To do design well, you need to take many "softer" factors into account: expectations, gut reactions, preferences, social context, beliefs, and values. All of these factors could affect the design of an application or site. Among these softer factors, you may find the critical feature or design factor that makes your application more appealing and successful.

So, be curious. Specialize in finding out what your users are really like, and what they really think and feel.

Understanding People with Research

Empirical discovery is the only really good way to obtain this information. Qualitative research, such as one on one interviews, gives you the basis for understanding your audience's expectations, vocabulary, and how they think about their goals or structure their work. You can often detect patterns in what you're hearing. These are your signals for guiding the design. Quantitative research, such as a survey, can give numerical validation or disqualification to your quant findings.

To get a design started, you'll need to characterize the kinds of people who will be using your design (including the softer factors just mentioned), and the best way to do that is to go out and meet them.

Each user group is unique, of course. The target audience for, say, a new mobile phone app will differ dramatically from the target audience for a piece of scientific software. Even if the same person uses both, his expectations for each are different—a researcher using scientific software might tolerate a less-polished interface in exchange for high functionality, whereas that same person may stop using the mobile app if he finds its UI to be too hard to use after a few days.

Each user is unique, too. What one person finds difficult, the next one won't. The trick is to figure out what's generally true about your users, which means

learning about enough individual users to separate the quirks from the common behavior patterns.

Specifically, you'll want to learn:

- Their goals in using the software or site
- The specific tasks they undertake in pursuit of those goals
- The language and words they use to describe what they're doing
- Their skill at using software similar to what you're designing
- Their attitudes toward the kind of thing you're designing, and how different designs might affect those attitudes

I can't tell you what your particular target audience is like. You need to find out what they might do with the software or site, and how it fits into the broader context of their lives. Difficult though it may be, try to describe your potential audience in terms of how and why they might use your software. You might get several distinct answers, representing distinct user groups; that's OK. You might be tempted to throw up your hands and say, "I don't know who the users are" or "Everyone is a potential user." But that doesn't help you focus your design at all—without a concrete and honest description of those people, your design will proceed with no grounding in reality.

This user-discovery phase will consume time and resources early in the design cycle, especially if you don't really have a handle on who your audience is and why they might use your designs. It's an investment. It's worth it, because the understanding you and the team gain gives long term payback in better designs: Solving the right problems, and fit for purpose.

Fortunately, lots of books, courses, and methodologies now exist to help you. Although this book does not address user research, here are some methods and topics to consider:

Direct observation

Interviews and onsite user visits put you directly into the user's world. You can ask users about what their goals are and what tasks they typically do. Usually done "on location," where users would actually use the software (e.g., in a workplace or at home), interviews can be structured—with a predefined set of questions—or unstructured, where you probe whatever subject comes up. Interviews give you a lot of flexibility; you can do many or a few, long or short, formal or informal, on the phone or in person. These are great opportunities to learn what you don't know. Ask why. Ask it again.

Case studies

Case studies give you deep, detailed views into a few representative users or groups of users. You can sometimes use them to explore "extreme" users that

push the boundaries of what the software can do, especially when the goal is a redesign of existing software. You can also use them as longitudinal studies—exploring the context of use over months or even years. Finally, if you're designing custom software for a single user or site, you'll want to learn as much as possible about the actual context of use.

Surveys

Written surveys can collect information from many users. You can actually get statistically significant numbers of respondents with these. Since there's no direct human contact, you will miss a lot of extra information—whatever you don't ask about, you won't learn about—but you can get a very clear picture of certain aspects of your target audience. Careful survey design is essential. If you want reliable numbers instead of a qualitative "feel" for the target audience, you absolutely must write the questions correctly, pick the survey recipients correctly, and analyze the answers correctly—and that's a science.

Personas

Personas aren't a data-gathering method, but they do help you figure out what to do with your data once you've got it. This is a design technique that "models" the target audiences. For each major user group, you create a fictional person that captures the most important aspects of the users in that group: what tasks they're trying to accomplish, their ultimate goals, and their

experience levels in the subject domain and with computers in general. Personas can help you stay focused. As your design proceeds, you can ask yourself questions such as “Would this fictional person really do X? What would she do instead?”

Design research is not marketing research

You might notice that some of these methods and topics, such as interviews and surveys, sound suspiciously like marketing activities. They are closely related. Focus groups, for example, can be useful, but be careful. In group settings, not everyone will speak up, and just one or two people may dominate the discussion and skew your understanding. There is also the very robust marketing practice of market segmentation. It resembles the definition of target audiences used here, but market segments are defined by demographics, psychographics, and other characteristics. Target audiences from a UI design perspective are defined by their task goals and behaviors.

In both cases, the whole point is to understand the audience as best you can.The difference is that as a designer, you’re trying to understand the people who use the software. A marketing professional tries to understand those who buy it.

It’s not easy to understand the real issues that underlie users’ interactions with a system. Users don’t always have the language or introspective skill to explain what they really need to accomplish their goals, and it takes a

lot of work on your part to ferret out useful design concepts from what they can tell you—self-reported observations are usually biased in subtle ways.

Some of these techniques are very formal, and some aren't. Formal and quantitative methods are valuable because they're good science. When applied correctly, they help you see the world as it actually is, not how you think it is. If you do user research haphazardly, without accounting for biases such as the self-selection of users, you may end up with data that doesn't reflect your actual target audience—and that can only hurt your design in the long run.

But even if you don't have time for formal methods, it's better to just meet a few users informally than to not do any discovery at all. Talking with users is good for the soul. If you're able to empathize with users and imagine those individuals actually using your design, you'll produce something much better.

The Patterns

The following patterns describe some of the most common ways people think and behave as it relates to software interfaces. Even though individuals are unique, people in general behave predictably. Designers have been doing site visits and user observations for years; cognitive scientists and other researchers have spent many hundreds of hours watching how people do things and how they think about what they do.

So, when you observe people using your software, or doing whatever activity you want to support with new software, you can expect them to do certain things. The behavioral patterns that follow are often seen in user observations. Odds are good that you'll see them too, especially if you look for them.

(A note for pattern enthusiasts: these patterns aren't like the others in this book. They describe human behaviors—not interface design elements—and they're not prescriptive, like the patterns in other chapters. Instead of being structured like the other patterns, these are presented as small essays.)

Again, an interface that supports these patterns well will help users achieve their goals far more effectively than interfaces that don't support them. And the patterns are not just about the interface, either. Sometimes the entire package—interface, underlying architecture, feature choice, documentation, everything—needs to be considered in light of these behaviors. But as the interface designer or interaction designer, you should think about these as much as anyone on your team. You might be in a better place than anyone to advocate for the users.

- "Safe Exploration"
- "Instant Gratification"
- "Satisficing"
- "Changes in Midstream"

- “Deferred Choices”
- “Incremental Construction”
- “Habituation”
- “Microbreaks”
- “Spatial Memory”
- “Prospective Memory”
- “Streamlined Repetition”
- “Keyboard Only”
- “Social Proof”

Safe Exploration

“Let me explore without getting lost or getting into trouble.”

When someone feels like she can explore an interface and not suffer dire consequences, she’s likely to learn more—and feel more positive about it—than someone who doesn’t explore. Good software allows people to try something unfamiliar, back out, and try something else, all without stress.

Those “dire consequences” don’t even have to be very bad. Mere annoyance can be enough to deter someone from trying things out voluntarily. Clicking away pop-up windows, reentering data that was mistakenly erased, suddenly muting the volume on one’s laptop when a website unexpectedly plays loud music—all can be

discouraging. When you design almost any kind of software interface, make many avenues of exploration available for users to experiment with, without costing the user anything.

This pattern encompasses several of the most effective usability guidelines, based on research, as identified by usability expert Jakob Nielsen. These guidelines are[2]:

- Visibility of system status
- Match between the system and the real world
- User control and freedom

Here are some examples of what “Safe Exploration” is like:

A photographer tries out a few image filters in an image-processing application. He then decides he doesn’t like the results, and clicks Undo a few times to get back to where he was. Then he tries another filter, and another, each time being able to back out of what he did. (The pattern named “Multi-Level Undo”, in Chapter 6, describes how this works.)

A new visitor to a company’s home page clicks various links just to see what’s there, trusting that the Back button will always get her back to the main page. No extra windows or pop ups open, and the Back button keeps working predictably. You can imagine that if a web app does something different in response to the Back button—or if an application offers a button that seems like a Back button, but doesn’t behave quite like

it—confusion might ensue. The user can get disoriented while navigating, and may abandon the app altogether.

Instant Gratification

“I want to accomplish something now, not later.”

People like to see immediate results from the actions they take—it’s human nature. If someone starts using an application and gets a “success experience” within the first few seconds, that’s gratifying! He’ll be more likely to keep using it, even if it gets harder later. He will feel more confident in the application, and more confident in himself, than if it had taken a while to figure things out.

The need to support instant gratification has many design ramifications. For instance, if you can predict the first thing a new user is likely to do, you should design the UI to make that first thing stunningly easy. If the user’s goal is to create something, for instance, then create a new canvas, put a call to action on it, and place a palette next to it. If the user’s goal is to accomplish some task, point the way toward a typical starting point.

This also means you shouldn’t hide introductory functionality behind anything that needs to be read or waited for, such as registrations, long sets of instructions, slow-to-load screens, advertisements, and so on. These are discouraging because they block users from finishing that first task quickly.

To summarize, anticipate their need, provide an obvious entry point, provide value to the customer first before

asking for something valuable (email address, a sale) in return.

Satisficing

“This is good enough. I don’t want to spend more time learning to do it better.”

When people look at a new interface, they don’t read every piece of it methodically and then decide, “Hmmm, I think this button has the best chance of getting me what I want.” Instead, a user will rapidly scan the interface, pick whatever he sees first that might get him what he wants, and try it—even if it might be wrong.

The term satisficing is a combination of satisfying and sufficing. It was coined in 1957 by the social scientist Herbert Simon, who used it to describe the behavior of people in all kinds of economic and social situations. People are willing to accept “good enough” instead of “best” if learning all the alternatives might cost time or effort.

Satisficing is actually a very rational behavior, once you appreciate the mental work necessary to “parse” a complicated interface. As Steve Krug points out in his book Don’t Make Me Think (Krug, Steve. *Don’t Make Me Think, Revisited: A Common Sense Approach to Web Usability*. New Riders, 2014.), people don’t like to think any more than they have to—it’s work! But if the interface presents an obvious option or two that the user sees immediately, he’ll try it. Chances are good that it

will be the right choice, and if not, there's little cost in backing out and trying something else (assuming that the interface supports "Safe Exploration").

This means several things for designers:

- Use "calls to action" in the interface. Give directions on what to do first: type here, drag an image here, tap here to begin, and so forth.
- Make labels short, plainly worded, and quick to read. (This includes menu items, buttons, links, and anything else identified by text.) They'll be scanned and guessed about; write them so that a user's first guess about meaning is correct. If he guesses wrong several times, he'll be frustrated, and you'll both be off to a bad start.
- Use the layout of the interface to communicate meaning. Chapter 4 explains how to do so in detail. Users "parse" color and form on sight, and they follow these cues more efficiently than labels that must be read.
- Make it easy to move around the interface, especially for going back to where a wrong choice might have been made hastily. Provide "escape hatches" (see Chapter 3). On typical websites, using the Back button is easy, so designing easy forward/backward navigation is especially important for web apps, installed applications, and mobile devices.

Keep in mind that a complicated interface imposes a large cognitive cost on new users. Visual complexity will often tempt nonexperts to satisfice: they look for the first thing that may work.

Satisficing is why many users end up with odd habits after they've been using a system for a while. Long ago, a user may have learned Path A to do something, and even though a later version of the system offers Path B as a better alternative (or maybe it was there all along), he sees no benefit in learning it—that takes effort, after all—and keeps using the less-efficient Path A. It's not necessarily an irrational choice. Breaking old habits and learning something new takes energy, and a small improvement may not be worth the cost to the user.

Changes in Midstream

"I changed my mind about what I was doing."

Occasionally, people change what they're doing while in the middle of doing it. Someone may walk into a room with the intent of finding a key she had left there, but while she's there, she finds a newspaper and starts reading it. Or she may visit Amazon.com to read product reviews, but ends up buying a book instead. Maybe she's just sidetracked; maybe the change is deliberate. Either way, the user's goal changes while she's using the interface you designed.

This means designers should provide opportunities for people to do that. Make choices available. Don't lock

users into a choice-poor environment with no connections to other pages or functionality unless there's a good reason to do so. Those reasons do exist. See the patterns called "Wizard" (Chapter 2) and "Modal Panel" (Chapter 3) for examples.

You can also make it easy for someone to start a process, stop in the middle, and come back to it later to pick up where he left off—a property often called reentrance. For instance, a lawyer may start entering information into a form on an iPad. Then, when a client comes into the room, the lawyer turns off the device, with the intent of coming back to finish the form later. The entered information shouldn't be lost.

To support reentrance, you can make dialogs and web forms remember values typed previously, and they don't usually need to be modal; if they're not modal, they can be dragged aside on the screen for later use. Builder-style applications—text editors, code development environments, and paint programs—can let a user work on multiple projects at one time, thus letting her put any number of projects aside while she works on another one. See the "Many Workspaces" pattern in Chapter 2 for more information.

Deferred Choices

"I don't want to answer that now; just let me finish!"

This follows from people's desire for instant gratification. If you ask a task-focused user unnecessary

questions in the process, he may prefer to skip the questions and come back to them later.

For example, some web-based bulletin boards have long and complicated procedures for registering users. Screen names, email addresses, privacy preferences, avatars, self-descriptions…the list goes on and on. “But I just wanted to post one little thing,” says the user plaintively. Why not allow him to skip most of the questions, answer the bare minimum, and come back later (if ever) to fill in the rest? Otherwise, he might be there for half an hour answering essay questions and finding the perfect avatar image.

Another example is creating a new project in a video editor. There are some things you do have to decide up front, such as the name of the project, but other choices—where on the server are you going to put this when you’re done? I don’t know yet!—can easily be deferred.

Sometimes it’s just a matter of not wanting to answer the questions. At other times, the user may not have enough information to answer yet. What if a music-writing software package asked you up front for the title, key, and tempo of a new song, before you’ve even started writing it? (See Apple’s GarageBand for this bit of “good” design.)

The implications for interface design are simple to understand, though not always easy to implement:

- Don’t accost the user with too many upfront choices in the first place.
- On the forms that he does have to use, clearly mark the required fields, and don’t make too many of them required. Let him move on without answering the optional ones.
- Sometimes you can separate the few important questions or options from others that are less important. Present the short list; hide the long list.
- Use “Good Defaults” (Chapter 8) wherever possible, to give users some reasonable default answers to start with. But keep in mind that prefilled answers still require the user to look at them, just in case they need to be changed. They have a small cost, too.
- Make it possible for users to return to the deferred fields later, and make them accessible in obvious places. Some dialog boxes show the user a short statement, such as “You can always change this later by clicking the Edit Project button.” Some websites store a user’s half-finished form entries or other persistent data, such as shopping carts with unpurchased items.

If registration is required at a website that provides useful services, users may be far more likely to register if they’re first allowed to experience the website—drawn in and engaged—and then asked later about who they are. Some sites let you complete an entire purchase

without registering, then ask you at the end if you want to create a no-hassle login with the personal information provided in the purchase step.

Incremental Construction

“Let me change this. That doesn’t look right; let me change it again. That’s better.”

When people create things, they don’t usually do it all in a precise order. Even an expert doesn’t start at the beginning, work through the creation process methodically, and come out with something perfect and finished at the end.

Quite the opposite. Instead, she starts with some small piece of it, works on it, steps back and looks at it, tests it (if it’s code or some other “runnable” thing), fixes what’s wrong, and starts to build other parts of it. Or maybe she starts over, if she really doesn’t like it. The creative process goes in fits and starts. It moves backward as much as forward sometimes, and it’s often incremental, done in a series of small changes instead of a few big ones. Sometimes it’s top-down; sometimes it’s bottom-up.

Builder-style interfaces need to support that style of work. Make it easy for users to build small pieces. Keep the interface responsive to quick changes and saves. Feedback is critical: constantly show the user what the whole thing looks and behaves like, while the user works. If the user builds code, simulations, or other

executable things, make the “compile” part of the cycle as short as possible, so the operational feedback feels immediate—leave little or no delay between the user making changes and seeing the results.

When creative activities are well supported by good tools, they can induce a state of flow in the user. This is a state of full absorption in the activity, during which time distorts, other distractions fall away, and the person can remain engaged for hours—the enjoyment of the activity is its own reward. Artists, athletes, and programmers all know this state.

But bad tools will keep users distracted, guaranteed. If the user has to wait even half a minute to see the results of the incremental change she just made, her concentration is broken; flow is disrupted.

If you want to read more about flow, there are multiple books by researcher Mihaly Csikszentmihalyi. One title is Flow (Csikszentmihalyi, Mihaly. *Flow: The Psychology of Optimal Experience*. Harper Row, 2009.)

Habituation

“That gesture works everywhere else; why doesn’t it work here, too?”

When one uses an interface repeatedly, some frequent physical actions become reflexive: pressing Ctrl-S to save a document, clicking the Back button to leave a web page, pressing Return to close a modal dialog box, using gestures to show and hide windows—even

pressing a car's brake pedal. The user no longer needs to think consciously about these actions. They've become habitual.

This tendency helps people become expert users of a tool (and helps create a sense of flow, too). Habituation also measurably improves efficiency, as you can imagine. But it can also lay traps for the user. If a gesture becomes a habit, and the user tries to use it in a situation when it doesn't work—or, worse, does something destructive—the user is caught short. He suddenly has to think about the tool again (What did I just do? How do I do what I intended?), and he might have to undo any damage done by the gesture.

Millions of people have learned the following keyboard shortcuts based on using Microsoft Word and other word processors. They are true universals now. Consistency across applications can be an advantage to use in your software design.

- Ctrl-X: Cut the selection
- Ctrl-V: Paste the selection
- Ctrl-S: Save the document

Just as important, though, is consistency within an application. Some applications are evil because they establish an expectation that some gesture will do Action X, except in one special mode where it suddenly does Action Y. Don't do that. It's a sure bet that users will make mistakes, and the more experienced they

are—that is, the more habituated they are—the more likely they are to make that mistake.

Consider this carefully if you're developing gesture-based interfaces for mobile devices. Once someone learns how to use his device and gets used to it, he will depend on the standard gestures working consistently on all applications. Check that gestures in your design all do the expected things.

This is also why confirmation dialog boxes often don't work to protect a user against accidental changes. When modal dialog boxes pop up, the user can easily get rid of them just by clicking OK or pressing Return (if the OK button is the default button). If the dialogs pop up all the time when the user makes intended changes, such as deleting files, clicking OK becomes a habituated response. Then, when it actually matters, the dialog box doesn't have any effect, because it slips right under the user's consciousness.

(I've seen at least one application that sets up the confirmation dialog box's buttons randomly from one invocation to another. One actually has to read the buttons to figure out what to click! This isn't necessarily the best way to do a confirmation dialog box—in fact, it's better to not have them at all under most circumstances—but at least this design sidesteps habituation creatively.)