

Cadence & Slang

Draft 209

Incomplete without surface noise

Most people shrink in fear from the task of designing their surroundings. They are afraid that they will make foolish mistakes, afraid that people will laugh at them, afraid that they will do something “in bad taste.” And the fear is justified. Once people withdraw from the normal everyday experience of building ... they are literally no longer able to make good decisions about their surroundings, because they no longer know what really matters, and what doesn’t.

— Christopher Alexander, *The Timeless Way of Building*

One day Soshi was walking on the bank of a river with a friend. “How delightfully the fishes are enjoying themselves in the water!” exclaimed Soshi.

His friend spoke to him thus: “You are not a fish; how do you know that the fishes are enjoying themselves?” “You are not myself,” returned Soshi; “how do you know that I do not know that the fishes are enjoying themselves?”

— Kakuzo Okakura, *The Book of Tea*

Cadence & Slang


by Nick Disabato

First edition written in Chicago between 2008 and 2010.

Second edition revised in Chicago between 2011 and 2013.

All errata pertaining to this edition are listed at <http://cadence.cc/errata/>.

All URLs were sourced at the time of this book’s publication. However, the web’s dual blessing and curse is that it’s ephemeral: content dies, or sites point to different locations. If you find a dead link that’s not yet listed on the above errata page, please get in touch with me.

 *Cadence & Slang* is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. You have the freedom to quote, send, translate, and reuse as much of this as you’d like—but you can’t make money off of it, and you have to give credit where credit’s due. Compared to copyright, this is an extremely lax and charitable way to publish, and it restores sensibility and comfort to our natural impulses to share information with each other. The full license is available at <http://creativecommons.org/licenses/by-nc-nd/3.0/us/> for the morbidly curious. If you have any questions, feel free to email the author.

Cadence & Slang is published by **Distance Press**, a small company focused on making great long-form design writing. You can read more about Distance Press at distancepress.com.

Please say hello:

nickd@nickd.org

nickd.org

cadence.cc



Slang *is the vernacular change of what we once knew.*

Page **10** begins chapter 1, **Empathy & Kindness.**

Starting off right, customer service, the tone of copy, humility.

Page **24** begins chapter 2, **Consistency & Character.**

Elements, layout, pattern languages, familiarity, reuse, natural mappings.

Page **47** begins chapter 3, **Simplicity & Clarity.**

The simplest complete solution, chartjunk and excess, metaphor.

Page **59** begins chapter 4, **Managing Expectations.**

Incremental disclosure, usability testing, research, mistakes, triage.

Cadence *is the rhythm that we apply to what we understand.*

Page **78** begins chapter 5, **Perceived Accomplishment.**

Rhythm, affordance, muscle memory, the inverted pyramid, preferences.

Page **86** begins chapter 6, **Cognitive Cost.**

Flow, instinct, modes.

Page **100** begins chapter 7, **Cultural Norms.**

Fitting the product to the audience, research, language, and culture.

Page **105** begins the **conclusion.**

Page **106** begins the **reference.**

Page **110** begins the **colophon.**

We've come a long way since *Cadence & Slang* was first released in 2010. In technology, mobile devices have exploded in use, with touch-based interaction models becoming the new normal. In design, "flat" has reigned, with significant changes towards plainer iconography, bold colors, and an emphasis on content. In book-related matters, *Cadence & Slang*'s first print run sold out, I've sold over eight hundred PDFs since, and copies of the original have gone on eBay for way too much money.

When I first released this book, I tried to build a compendium of evergreen principles for interaction design. So, why revise it? Shouldn't an evergreen book last forever?

I'll be the first to admit that many of *Cadence & Slang*'s principles were due for revision. The first edition was written hastily, with a firm deadline and a dozen editors handling disparate parts. This new edition has only two editors, one of whom is me. I have become a much better writer and editor in the past three years, with a far better-honed BS detector than I ever had before. And three years gave me the critical distance to edit this manuscript as if I had not written it, which allowed me to poke holes in my thinking on a level that I was previously incapable of.

No part of this book went untouched; every single section differs from the original in some way. The whole book is laid out and typeset anew, with a different typeface family and a fresh approach that can only come from added experience.

This is the 209th draft of *Cadence & Slang*. It, like technology itself, is an evolving medium, slowly moving towards something that can form the beginning of any suitable project. May there be many more projects to come. Thanks for reading, and for supporting this small, humble attempt to summarize a large, complicated thing.

Nick Disabato
Chicago, IL
August 2013

You can help improve *Cadence & Slang*. Suggestions are always welcome through [email](#) and [Twitter](#).

Nick Disabato, "On electronic copies of *Cadence & Slang*." <http://thedata.cc/post/1447280333/cadence-ebook>.

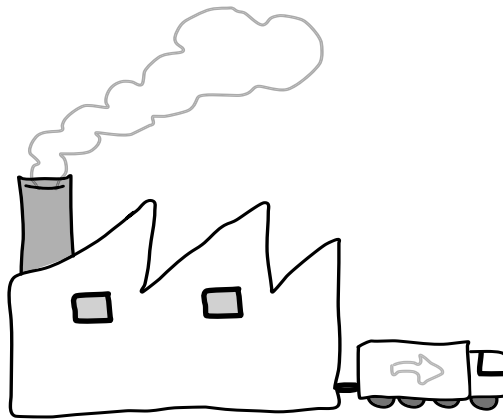
O:

Introduction

The first time I used a computer, I was two years old. Six years later, I built my first computer from scrap parts. (It didn't work.) In the ten years that followed, I learned a bunch of things that could only qualify as "power user": command line prompts, tweaking BIOS settings, applying the "Hot Dog Stand" theme to Windows 3.11, overclocking a computer with an air conditioner, and setting a motherboard on fire.

But around 2002, my mindset began to change. As I worked all-nighters in college, I cared more about how easily I could complete tasks than I did about tweaking my hardware and software. A natural progression followed: I became interested in usability, I paid more attention to the details of layout, and I tried to think about how I familiarized myself with something new.

At the same time, lots of products were being released that focused more on experience than features. They were tremendously successful, both in sales and in educating users about what to expect out of technology. And now we're trying to reverse engineer them. What is it about successful products that made them successful? What craft comes into play?



THE BEGINNING.

The history of manufactured products pivots around the Industrial Revolution, circa 1850. During this time, the rise of mass production of consumer goods shifted the focus of production away from personal and craftsmanlike methods of production. Prior to this, manufacturers were intimately connected with their consumers. Mass production presented a trade-off, as manufacturers opted for increased output at the expense of a solid manufacturer-consumer relationship, safe and humane working conditions, and high-quality construction. Quality and kindness competed with companies' financial success.

Of course, many technological products wouldn't exist without the inventions and attitudes that took root in this period. But "good enough" conflicts with consumers' desire for humane experiences. In the past ten years, new products have addressed the problem by providing better service, simpler functionality, and improved clarity—and they've done so with more

financial success than those who neglect to pay such attention. Consumers are willing to pay a premium for quality: they understand that in technology, as in much of life, you get what you pay for.

And yet, now there's an incentive to do only good *enough*. Notions of planned obsolescence have affected our field in tandem with the breakneck pace of technological progress. Services are abandoned, or “sunsetting” in business parlance. Applications are pulled from release. New operating systems don't work on the hardware of a few years ago. But if we're tool-builders, as I often hear, then it seems a bit disingenuous to obsolete ourselves so often. We can make fewer, better things that stand a greater chance of surviving the ravages of time. After all, nobody “sunset” a hammer.

AND SO.

This book is about interaction design, which is the art and craft of making things easier to use. Useful things make more sense to people. They add beauty and simplicity to our lives.

Technology is reductive by definition. It connects with our lives by metaphor, with an internal slang that has to be translated during its use.

The best technology is alive and humane. It treats you well. It listens. It responds in a way that makes sense. It apologizes when it screws up. It rejoices when you do well. In short, it's built to act like an empathetic and supportive person.

When a product fails, we believe that its bad behavior is *our* fault, that it's a quirk we must endure in order to complete the task. For better or worse, this impression could not be further from the truth. Human beings made this problem—and we have the power to fix it. We can connect the experiences of people in the real world to the native slang of a mechanical world. We can grudgingly accept machines' foibles, or we can try to change them so they act like supportive colleagues.

Well-executed designs are easier to learn. It's easier for us to finish tasks. And the cadence that our work attains will be dramatically more fulfilling than we had ever known.

HOW TO DO IT.

Good interactions involve more craft than art, which sets our field apart from other areas of design. Interaction designers cultivate a sensitivity to the issues, common and uncommon, that can make technology more frustrating. Moving that button to the right location—and making it look like a button. Cutting out that redundant paragraph of help text. Changing the layout of a web page to make it easier to read. These decisions are more functional than aesthetic, a fact that recalls a famous statement by Apple founder Steve Jobs: *design is how it works*. Once you understand what problems affect a product's usability and why, they're easy to see. All it takes is practice.

Rob Walker, “The Guts of a New Machine,” *The New York Times*, <http://www.nytimes.com/2003/11/30/magazine/the-guts-of-a-new-machine.html?pagewanted=all>.

So this book isn't limited to those who identify as interaction designers or usability analysts—in fact, seasoned interaction designers may find a lot of the content to be obvious, as very few new ideas are advanced.

But problems affect everyone in our field, because we *all* make design decisions, sometimes unconsciously, every day, and they have to result in something that our customers won't want to jettison into the sun. If you're a graphic designer or software engineer, and you want to understand the basics, the starting point: you've come to the right place. And if you think that programming, building circuits, running a data center, writing documentation, or promoting strategy excludes you from that, ask yourself: *who are you working for?*

WHAT'S WRITTEN AFTER THIS PAGE.

This book offers a series of principles for building useful, practical, and humane technology. It's organized into seven chapters that address the problems facing interaction design, and how designers can adapt and respond to them.

Ultimately, it should teach you how to frame any new project, and how to revise a current product so it's easier to use. My hope is that in confronting these problems, we can change our attitudes, question our premises, and make something better in this world.

The first half of the book covers the unique slang in which technology can communicate, and suggests how to make products that can better adapt to us.

The second half discusses the way that we perceive technology, and the cadence that we adopt when we're deeply immersed in productive activity.

Slang:

Successful products are confident, empathetic, and kind.

In technology:

- Consumers drive progress more than corporate customers.
- Utility and simplicity encourage adoption more than style.
- Forgiving interfaces cause less frustration, helping people adopt a positive, stronger relationship with the product.

1.1. *Never lose sight of what you believe in.*

Products reflect the values and desires of their creators—and those with a sincere interest in design tend to make better designed things. It takes hell-bent resolve to design well. It's not enough to say you want good design: you have to love it and demand it.

Your attitudes come from your beliefs about what constitutes beauty and quality. You probably formulated these early on. Don't forget them. Write them down and post them someplace ridiculously conspicuous. They'll keep you on track during times of uncertainty.

1.2. *Design on your own terms.*

Many great products provide some improvement on what already exists. This can take many forms: simplicity, features, speed, or finding a new way to do an old thing. At the same time, we design based on what we know, and it's hard to realize totally novel possibilities. In technology, there's a fine line between inspiration and rote copying.

Because our field progresses so fast, it's hard to find the time for inspiration. Many of us take the easier route: looking at what already works, and copying those designs with slight modifications. In technology, there are *leaders* who figure out ways to advance the medium and make existing things more useful, and there are *followers*, who borrow from the leaders to make what they believe are safer bets.

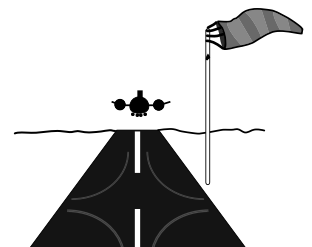
Copying is now so frequent that great new products which do new things have become the exception, not the rule. But copies neglect the subtler features of successful products. For example, music players copied the iPod between 2002 and 2007, but ignored the integration with iTunes, and placed less emphasis on perfecting such a connection. Shortly after the MacBook Air became popular, PC makers began copying its industrial design. After Square came out, other established companies released credit card readers and associated mobile apps of their own.

The interplay between leaders and followers is reminiscent of the “cargo cults” that appeared after World War II. During the war, the United States and Japan shipped supplies to small Pacific Island nations that served as slapdash bases. Runways were built to accommodate the airplanes.

Once the war ended, there was no more use for the bases, so the Americans and the Japanese left. But the isolated societies were fascinated by this new technology, and they built runways of their own, complete with

1.

Empathy & Kindness



air traffic control, ground crews, and so on. “Airplanes” were built out of straw. But the real airplanes never landed. They had perfectly emulated the whole pomp and circumstance of an airplane’s arrival, for aircraft that would never appear.

The systems necessary to build, fly, and land an airplane were missing. The entire *context* was ignored. The operation had no integrity. None of the engineering and infrastructure problems were addressed. And why not?

1.3. *Design interactions like you’re typesetting writing.*

Interactions are narratives. Hopefully, they augment the stories of our lives in a beneficial way. Can you think of anything you’ve used that accomplishes this? Or can you think of something that you wish did so, but failed in some way? *What* way? What would bridge that gap?

Writing is the art of communicating ideas, and typography is the art of displaying those ideas in a clear and readable way. When it functions well, typography can improve writing, communicating with equal importance. In the 20th century, many influential authors—David Foster Wallace, Richard Brautigan, e.e. cummings—used inventive typography to their advantage.

Returning to interaction design, a **task** is a user’s perceived goal. In order to communicate this goal, someone works with an **interface**, which is the surface appearance of a product: its buttons, menus, text fields, and so on. Writing is to typography as tasks are to interfaces. People communicate their tasks to the interface as **input**, and the product’s job is to interpret and respond to that input. The separation between a task and an interface is similar to the difference between ideation and execution in writing and typography. It’s up to the designer to provide clarity and balance to the resident slang.

In an email client, for example, one task is to reply to an existing message. Another task is to compose an email from scratch.

1.4. *Write a mission statement.*

Before you can make anything, you have to determine what you want to make. What do you stand for? How do you build consensus with other team members?

A **mission statement** begins this process by answering the most basic questions: what you do, for whom you’re doing it, and why. As with any big question in life, it shouldn’t be taken lightly or composed in a vacuum. Develop it into something substantial and satisfying. This provides consensus and guides you through ambiguous times. Refer to the mission statement in the development process, to determine which decisions fulfill it and why.

1.5. *Take the time to get it right.*

The beginning of any product’s development is stressful, and it’s easy to make mistakes without noticing. Stopgap solutions pervade products that weren’t designed well early on, but in technology, it’s impossible to build well on a

shoddy foundation. The patchwork of fixes is untenable in the long term, because each fix requires exponentially more effort.

Fortunately, the *first* product in its class is rarely the *best* one. Early releases are overshadowed by those who took the extra time to make a product easier and more enjoyable to use.

Also, releasing too quickly risks more mistakes. The earlier that mistakes are made, the more damaging they can be, because new features are then built on shaky foundations. Such faults can spread epidemically, taking over new versions of the product. Put another way, solutions that work *well enough* aren't the same as solutions that work *well*.

It's easier to discard initial failure and start anew than it is to fix every part of a large project. But even when starting over, we take away some knowledge of how to do it right the next time.

1.6. *Products should be fast and reliable.*

A product is only as good as the code and hardware that powers it. Nobody wants to use something that's buggy or unreliable. While it sometimes helps to slow someone down to ensure a lack of error, they should always feel a sense of accomplishment in what they're doing. Any cognitive interruption is problematic, and any unintentional data loss is disastrous.

In general, performance sacrifices are unacceptable, and readable, beautiful code is less ideal than a perceptibly fast product. Steven Frank, cofounder of Mac software company Panic, emphasizes this:

It's not just that the iPhone has fancy woo-woo transitions and purty graphics; [its quality] runs all the way down the software stack. For example, when I tap on something, I don't have to hover for five seconds wondering "now did it get that tap, or do I have to do it again?" This is something other platforms are still struggling with. When we say you have a bad experience, this is the sort of thing we mean. It has little to do with features, and everything to do with core functionality.

An interface's appearance matters, of course, but so does its **behavior**—which is the way that products respond to input.

Performance is behavior. When we talk about good interactions, we should focus as much on the way that something is coded as the way that it looks. For example, the quality of search results matters just as much as how they're displayed. A form's validation mechanisms matter just as much as its layout. And the algorithms that comprise an online store's recommendation engine matter just as much as how those recommendations are presented.

A beautiful concept with poor technical execution remains unfinished, one that never came alive. Focus on both ends; concepts don't matter without execution.

For more on cognitive interruptions, see 6.1.

Steven Frank, untitled, stevenf.com, <http://stevenf.tumblr.com/post/218293148>.

1.7. Develop for intermediate users.

Let's consider people at three broad experience levels: beginners, intermediates, and experts. These three levels exist with varying proportions in all products. Someone's experience level depends on how many features she uses, how easily and quickly she works with those features, how extensively she customizes the product, and how willing she is to try out new tools.

Beginners have rarely or never worked with a given kind of product. They likely have no experience—or only limited experience—with your interaction model. They haven't figured out how to work with your product *or anything else like your product*. If they do have any experience with similar technology, they have little to no productive output on it.

Experts are exactly the opposite, of course. They know most of a product's features, love figuring out new things, and likely have passionate opinions about their favorite tools. They have substantial experience with using a specific product, and with technology at large. They know what makes a product good and what makes a product bad *for them*, and they can articulate why. They have no trouble figuring out new products. If they develop products of their own, it's likely that they're an expert as well.

Intermediates are the hardest group to describe. They represent a broad range of experience levels and they work with a wide range of products. They have passing fluency with whatever products they choose to use. They tend to adopt routines around these products, and are more hesitant to embrace new ones on their own. They know all of the basic functions of a product, and they learn more advanced features when they provide an essential benefit.

Beginners and experts usually represent minority segments. In many situations, designing for intermediates will please most people. It's easiest to prove this by showing why you shouldn't target either beginners or experts.

DEVELOPING FOR BEGINNERS.

When you develop for beginners, you have to overcompensate by labeling most form elements, adding confirmation prompts, and offering a comprehensive help system. But these features don't make products simple.

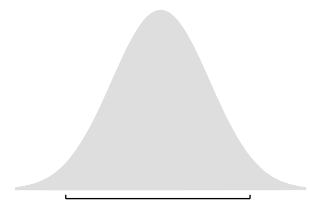
There are ways to teach beginners such that they become intermediates after enough use. Documentation is one way. Brief instructions on how to get a product up and running—with descriptions of basic tasks—can guide beginners through a setup process. And you could create a prompt that appears on a product's first launch which explains certain elements or walks beginners through basic steps. None of these methods appreciably slow down users of other experience levels.

DEVELOPING FOR EXPERTS.

When you develop for experts, on the other hand, almost nothing needs to be documented, and the interface can be organized in any way that makes

Experts tend to gravitate around a small set of products that work well for them. Beginners use few products in the first place.

According to Alan Cooper, *About Face* (Wiley), p. 42, these three experience levels form a bell curve with intermediates at the hump:



a shred of sense to you. You know that experts will tolerate it, that they'll be able to work with any level of complexity—but obviously developing for experts alienates beginners and intimidates intermediates. Targeting experts requires minimal empathy.

DEVELOPING FOR INTERMEDIATES.

Targeting intermediates assumes that people are capable of picking up a device, understanding it after a short period of time, and that they don't need their hands held at every step. Intermediates have some knowledge of technological environments, but don't operate within them as if they were a fluently spoken second language. Products designed for this segment can be simple enough for most laypersons and powerful enough for most experts.

Most people are smart enough to figure things out, but many still need encouragement and support. Finding some ground between the extremes results in something that can be useful for everyone.

Interaction design—and, by proxy, quality user experiences—became popular in part because technology became an essential part of *all* of our lives, not just experts'. In turn, this is reflected in the most popular technologies of our time; for example, smartphones have abstracted away their file systems, because they represent too high of a learning curve for most people to tolerate.

1.8. *Satisfy as many people as possible, and stop there.*

Our desires and preconceptions vary infinitely, and there's no way to make something that satisfies everyone. While this sounds like a major issue, it's easy to overcomplicate: running focus groups or market research to keep on top of things, or building too-complex products with features that appease vocal minorities. Don't be everything to everyone.

Imagine you're running a store. Some customers buy things, others leave, and a handful return what they bought. If people didn't want what you offered, it doesn't necessarily reflect poorly on you.

People decide to buy something because of how it looks at the time of their purchase, not how it might change in the future. As this applies to your product, every modification in every version risks alienating the customers that are likely to be your most ardent supporters. It's okay to say no to feature requests—even ones that appear common—if you think they'll detract from others' experiences.

Ultimately, you have to offer something great if anyone is going to stop by your store. You have to operate with the faith that customers will select the products that fit them best. The world has worked like this for hundreds of years, and those who work in our field shouldn't expect otherwise.

Obviously, you need to listen to your customers, or you'll build a bad reputation for yourself. But you still have to adhere to your own principles. What are the best ways to find balance?

Alan Cooper, *About Face* (Wiley), p. 249:

If we want users to like our products, we should design them to behave in the same manner as a likable person.

1.11 has more on giving great service and retaining business.

1.9. Constraints are inherent to design.

All good designs meet limitations. While frustrating in the moment, constraint is one of the most useful qualities of any design process. Constraints are the walls that determine the product's shape, guide its character and spirit, and strengthen it as we build.

For more on iterating a product carefully, see 5.3.

1.10. Make a great impression before a sale.

Consider the perspective of someone who isn't an expert in technology and doesn't keep up with its trends. He's standing in a store, trying to pick out a printer. Here are the names of the products that he can choose from: E260DN, NX515, H470, MP620, C309A, J6480, MX700, P2055DN, HL-2040. While these printers are from different manufacturers, their model numbers are equally inscrutable, and it's impossible to associate manufacturer with model number.

Our notional customer has only each model's price, features, and *maybe* his friends' recommendations. If he has the latter, "get the P2055DN" sounds ridiculous (try saying it out loud!), and the model number is impossible to remember—and it may not even be the kind of printer that he's looking for.

Nothing about these printers is unique or special. But our notional customer wants to make one part of his life, and he's stuck in a fluorescent-lit Brutalist warehouse of a store looking at a bunch of undifferentiated cardboard boxes. In six months, the NX516 will come out, and nobody but the manufacturer will care. As a result, our customer ends up purchasing something that appears to be crafted with little care for how he thinks and acts.

If you go into a grocery store, it's usually obvious what the products do. Products are named clearly; a can of soup usually looks like a can of soup. People are around to help when you have questions. Figuring the whole thing out is much simpler than buying a new computer or camera. So why is the computer store any different?

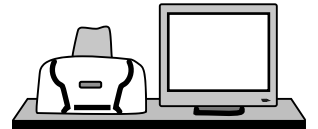
A product isn't just a physical device or its software. All parameters of a user's experience must be addressed—and the most important time to do this is in the first impression. The first impression presents a wide range of considerations, from the packaging to the setup to the initial state.

Brand products well, so they seem more special than just any printer, camera, or music player. Give them names that people can pronounce. Make good packaging. Make setup effortless and fun. A product isn't done until each of these aspects receives the same amount of care.

1.10.1. The product's marketing should reflect its quality.

Like the aforementioned inscrutable model numbers, marketing is at risk of being too technical and inhumane. Given how marketing usually provides the first impression, it's surprising how frequently it's disconnected from the product.

These are the model numbers of real printers, out today. Never mind that MX700 is also the name of a computer mouse.



This customer may be in a brick-and-mortar store, but the same ideas apply to e-commerce.

Marketing should clearly communicate the product's utility. Understanding the symbolism behind a logo and product name aids both you and customers. It reminds you of your original mission and helps you to create an appropriate product. It's easier for customers to understand and remember.

Marketing shouldn't promise a specific lifestyle. Showing a bunch of nubile coeds in a convertible barreling down a street doesn't tell people anything about technology. Showing a product for a half-second at the end of a commercial doesn't tell anybody anything about it. Employing a viral hype scheme with a slow reveal tells people a story that is ultimately divorced from their actual experience with the product. Focus on the product itself. Show somebody using it. Show its utility.

Marketing shouldn't exaggerate performance. Instantaneous transitions promote inaccurate expectations. A product should be advertised honestly. If you're unsatisfied with the product's performance when it's filmed for a commercial, then your customers probably will be too. Fix it.

Marketing shouldn't overstay its welcome. Once you've made the sale, get out of the way.

1.10.2. *The product's setup process should exemplify its quality.*

A setup process should be fast, simple, unambiguous, and effortless. In the case of software, many operating systems offer default installation frameworks. Use them when they're available: don't waste time on an installer that needs to be learned.

The setup process shouldn't obstruct other tasks, forcing unrelated applications to quit or taking up so many resources that it renders the system unusable.

Products should be installed in a maximum of five steps:

1. open the setup program or turn the product on,
2. confirm the intent to install,
3. confirm the install location,
4. authorize with an administrator password, and
5. confirm a complete installation.

For many products, it's possible to eliminate one or more of these steps.

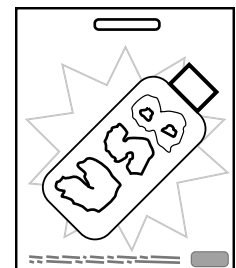
Streamline the process for more complex products by encouraging default installation procedures. This reduces the number of steps and keeps you focused on your product's most common configuration. Customization, if it has to exist, should be an optional quasimode.

1.10.3. *The product's packaging should exemplify its quality.*

Ideally, a product's packaging should reflect the product as an extension of its own design. No matter what form it takes, though, it should be humane and unobtrusive.

This is just one example of product setup, and only the first step is mandatory. Setting up new hardware should involve nothing more than plugging in a couple of cables and pressing the power button.

For more on quasimodes, check out section 6.5.4.



For example, vacuum-sealed clamshell and side blister packaging are far more difficult to open than many alternatives. There is no reason why someone should have to take a knife to a product's packaging. Nobody should have to expend significant effort to open a box. The problem is even worse for people who are young, old, or disabled. Standard cardboard boxes and snap-shut clamshells (without the sides welded together) are better options at reasonable cost.

Forcing someone to destroy packaging also removes its potential for reuse if they return the product. Packaging should permit reuse by both the retailer and the customer.

1.10.4. *The product's manual should exemplify its quality.*

When people buy something new, they want to use it as quickly as possible. Many products come with quick start guides that help people get up and running quickly, while others ship with a 400-page tome they call a user manual.

The quick start guide should include all the documentation that people need to operate a product. Nobody has the time to read an enormous manual, which is a waste of paper and an intimidating deterrent.

1.10.5. *The product's initial state should invite people to interact with it.*

Good products invite you to play with them. They promote the impression that using them is easy. This impression is often formed at the first screen that you see: the **initial state**, a blank, untouched slate with no data entered or behavior triggered.

The initial state conveys the product's slang, implying what to do with it. Here, people make snap judgments about how trustworthy and useful a product is. They'll try to figure out what kind of tool it is. What can it do? What is it supposed to do? Then they'll try to work with it, triggering behavior on buttons or text fields. How the product responds to those first stimuli implies the way it'll behave in the future.

What does the initial state look like? What does someone see as they try to orient themselves to the slang of something new? What format is the invitation to act presented in: is it on a marketing site, or is it part of an email?

Invitations can be *explicit*, accomplished with unambiguous copy that guides you through opening steps. Or they can be *implicit*, accomplished by highlighting various elements—like an arrow pointing to a larger-than-usual text field. In either case, your first impression should prompt a specific action.

For example, your goal on an e-commerce site is to buy a product, so getting products into a shopping cart is an important step. Explicit cues can then guide people toward interesting deals or new products.

Or consider a registration form with many required fields. Highlighting required fields implicitly draws attention to them, while a progress indicator explicitly comforts someone in a form spanning multiple pages.

For more on how to predict next steps, see 6.1.2.

What's your product's first impression? What does it ask people to do? Is it constructive? Is it kind?



1.11. Provide great service.

Writer Diana Kimball, on how she interacts with people:

But still, I do my best to set the tone of every interaction at “extremely considerate.” I try to do this even when the initial tone has been set at “less considerate,” either by the other person or—more often—by circumstance. It is amazing how often the other person will respond in kind. You can actually live in a slightly more pleasant world by doggedly attempting to be considerate with everyone you meet. By not unnecessarily impinging on their time; by thanking them when you’re grateful, apologizing when you’re sorry, and smiling if the mood strikes you.

Diana Kimball, “Trying,”
dianakimball.com, <http://www.dianakimball.com/2008/09/trying.html>.

Service is the way you treat any past, present, and future customers. Good service isn't about making sure nobody sues or slanders you; it's about making friends who tell others about what you do. It's about making people happy.

Something, at some point, is going to dissatisfy one of your customers. Someone, at some point, will hesitate to buy into what you stand for. In every situation, good service helps you cement relationships and humanize your business. It extends the experience beyond the simple transaction of money for goods. And it's the best way to make people into fans.

If your work involves interacting with people at any point, you should serve them as well as possible. Good service has to be honestly fought for. Nobody wins fans by cheating; favor comes naturally from making a good product and genuinely caring about people.

Service is a culture. No matter what, if you practice it, define it, and train employees to do the same, then service can permeate your organization and affect your customers' perceptions. You have to make it so service is easy,

encouraged, and rewarded. This can manifest itself in many different ways, but it should always exist in some form.

1.11.1. *Define, measure, and reward good service.*

There's no way to provide good service if you don't define what it means for your company. There's no way to determine whether you're providing good service if you don't measure it. There's no way to sustain a culture of good service if you don't reward your colleagues for providing it.

DEFINING SERVICE.

Figure out specifically what you do to satisfy customers. You can establish service generally in a mission statement, but mission statements are empty promises without a practical plan to back them up.

Good things to integrate into a service plan include:

- Figuring out what's wrong promptly.
- Apologizing for any problems that arose.
- Fixing issues unquestioningly.
- Going out of your way to make sure customers are happy.

MEASURING SERVICE.

Service can be measured in many ways. What you choose to measure depends on what types of services you provide.

Some good starting points include:

- Calling customers back to ask them about their experiences.
- Sending out surveys.
- Recording all complaints to determine the biggest problems.
- Recording all compliments and trying to figure out which parts of your business are succeeding or failing.
- Tracking and timing the turnaround of support queries.
- Figuring out patterns: who places repeat orders of a specific product, who comes back for a wide array of products, who places more support requests than usual, or who refers others to your business the most.

REWARDING SERVICE.

Develop incentives for employees. Make them part of your daily, weekly, or monthly routine. When applicable, tie employees' bonuses to the quality of service that they provide. If you make sure that service is measurable, you'll be able to figure out who's providing it well, and you'll be able to cite exceptional instances.

1.11.2. *Underpromise and overdeliver.*

Managing expectations is a big part of great service. But when you're dedicated to making people happy, you have to fight the temptation to promise them things that may be impossible to fulfill.

If you underpromise, though, you set expectations that you can meet or exceed. Underpromised goals are grounded in what you're already good at. Set deadlines too far, financial projections too low, feature sets too thin, and product launches too modest. If you keep your estimates reasonable, customers have no reason to believe otherwise. And when you overdeliver, customers feel special because you're going above and beyond their expectations.

1.11.3. *Take responsibility for your mistakes.*

It's painful to admit when we've screwed up, but we all have to do it at some point. On the bright side, solving problems can foster trust more effectively than the absence of problems.

Don't be afraid to take the blame, even when you don't think the problems are your fault. Part of good service involves wanting to understand why the customer is upset, and doing whatever you can to fulfill their needs. Address problems quickly and take immediate, unflinching responsibility for them.

1.11.4. *Encourage feedback.*

You don't need someone on call 24/7, but you should provide a way for people to submit questions and comments. Ensure that there are open, unobtrusive feedback channels.

On a website, a feedback mechanism can be placed on every page. In an application, feedback can reside in a contextual menu item. On mobile devices, a feedback link can be placed in the application's preferences.

1.11.5. *Don't systematize your conversation.*

Architect Christopher Alexander, on first impressions:

Have you ever walked into a public building and been processed by the receptionist as if you were a package?

This should be asked about your service. You can't afford to treat frustrated customers as machines.

Don't route phone calls through a labyrinthine system where people must diagnose their own problems. It's up to *you* to figure out what's wrong. Send queries to real humans, who can determine the problem much faster and more accurately than your customers can, and handle it accordingly. Support staff should work without quotas or time limits.

You may object that this doesn't make economic sense. But that isn't the point of making people happy, and goodwill is harder to measure than sales.

It is also a big part of making good products. For more on managing expectations, see 4.1.

Christopher Alexander, *A Pattern Language* (Oxford University Press), p. 705.

While not directly related to software, Christopher Alexander has influenced software development theories considerably, and is returned to frequently in this book. Section 2.6 applies his idea of modular patterns to interaction design. His books *A Pattern Language*, *The Timeless Way of Building*, and *Notes on the Synthesis of Form* (Oxford University Press) are pretty fast reads despite their length; and while they concern architecture and urban planning, they remain tremendously inspiring.

Though it costs more to hire extra support staff, or provide free stuff in apology, the word-of-mouth benefits and repeat business more than make up for it. The intangibles matter.

If you establish a reputation for good service, customers will be much happier to give you their business again and again—and it feels good to know that you’ve done the right thing.

1.11.6. *Answer questions quickly.*

People don’t want to waste their time. Questions submitted electronically should be replied to in under 36 hours. Phone calls shouldn’t take more than an hour, except when seriously dire troubleshooting is necessary. Unanswered questions should be prioritized based on how long they’ve waited in the queue.

And even if your answer is “I don’t know,” explaining *why* you don’t know comforts them. It implies you’re interested in solving their problems, and it opens the door for future support calls or second opinions from other team members.

1.11.7. *Establish a case thread for each problem, and archive cases for each customer.*

Even though you should enact your own system of service—a way to file issues and collect data on recurring trends—you don’t want to appear too rigorous. Having an unstructured support *front end*, where customers aren’t routed through a maze of questions, means your support staff is personally responsible for organizing their concerns.

Every new issue should be the beginning of a **case thread**. Threads have a one-to-one correspondence with customers’ problems. They should be updated every time you communicate with the customer.

All of a customer’s threads comprise his or her **case history**. The customer’s case history should be consulted at the beginning of every contact, to better understand the context of their problem and what progress has been made.

Customers never see this infrastructure, of course. Case threads and histories are for your own benefit: they help prioritize repeated problems, and they isolate customers who are having significant trouble.

1.11.8. *Link threads to staff members.*

Likewise, each case thread should be associated with a support team member. Staff members should own whatever threads they initiate, because the first contact with a customer will yield most of a case’s significant information. Moreover, people are comforted by establishing a relationship with a specific person, and your staff will be more invested in solving complex problems, rather than diverting them around the organization.

1.11.9. *Treat your colleagues as well as your customers.*

Coworkers should treat each other as well as their customers; even if they may not be best friends outside work, they should put any differences aside in the workplace. Internal issues should be addressed with just as much respect and sensitivity as customer complaints.

1.12. *Don't let popular opinion dictate the interface if it conflicts with your mission.*

Targeting a small group of people—a specific demographic, or specialists in a field—will result in a more focused product. Design decisions are then made with someone specific in mind, rather than trying to please everyone. This strengthens your customer base to weather competition, and they won't revolt at your slightest misstep.

Know when and how to say no. You can decline requests for additional features if they conflict with your beliefs or don't reflect the product's mission. If your business isn't perfect for every customer, somebody else's may be.

People need specialized products. Consider the numerous photo editing programs with varyingly complex feature sets. Some people just want to crop and change an image's white balance; others require more complicated functionality like layering and masking. Both sets of requirements are valid, but they require separate products. Targeting specific people helps you to focus your product and whittle away anything unnecessary.

1.13. *Stay small and autonomous.*

Small teams emphasize the individual. We work better and happier when we connect with others, and that's much easier to do in a small team.

There's an increasing trend among design firms to be small and independent. Many more of us are finding success as freelancers. Even within large companies, managers are organizing their workers into small teams, so their companies can *feel* small. As you grow, ensure that small teams are preserved as well as possible.

1.14. *Create products in the name of utility first, progress second, and money third.*

It's essential to find ways to make money, of course, but that should always be a secondary motivation to making something great. Products should be useful above all else. Resist the temptation to alter the product—even if the changes promise to make you more money—whenever those changes would compromise its utility.

It's tempting, but wrong, to taint the goal of utility with other objectives. Your integrity is tied to your product's integrity.

The different ways to encourage workplace cooperation are as variant as company cultures. One example is hashed out at Michael Lopp, "B.A.B.," *Rands in Repose*, <http://randsinrepose.com/archives/2010/03/19/bab.html>. For more on the ramifications of workplace culture, see Shanley Kane, "What Your Culture Really Says," *Pretty Little State Machine*, <http://blog.prettylittlestatemachine.com/blog/2013/02/20/what-your-culture-really-says>.

For more on defining and targeting a limited set of customers, see 1.8.

1.15. *The more successful your product becomes, the greater the need for humility and simplicity.*

Ideally, your product is wildly successful, with sales beyond your expectations and stratospheric consumer trust. Suddenly, many constraints disappear: you're able to upgrade to a larger office, hire more employees, and have more resources. The risk of losing the humility that paved your way to greatness suddenly pervades your life. Caving to that temptation threatens the great qualities of simplicity, utility, and delight that made your product successful in the first place.

Humility is not as obvious or as measurable as a good interface. It's ethereal, ambiguous, derived from values and behavior. And the only diagnostic you can run on it is your own capacity for reflection.

Many things change as an organization grows, but humility shouldn't be one of them. It should be a core principle that governs all of your dealings. Out of humility comes trust in your colleagues and the incentive to keep doing great things.

Simplicity resists scaling. As your product becomes more successful, the incentive to add features and please more people becomes ever greater. The need to preserve simplicity becomes more pressing because the resistance increases from many fronts, in ever greater magnitude. It takes effort to protect simplicity, but it will be repaid with an iconic, durable product. Does it feel whole? What's missing from it? What can be taken away? What can be modified? How would it look and behave *then*?

The **character** of an interface is the way that it communicates with you, as well as the way that it allows you to interact with it. It should be consistent and unified in all parts.

Consistency aids clarity in every part of the product.

2.1. *Apply a consistent layout with consistent negative space.*

All products begin with a blank canvas: no text, no buttons, no graphics. An **element** is anything that alters a blank canvas. Buttons, switches, speakers, headphone jacks, keyboards, scroll wheels, scroll bars, paragraphs of text, labels, input fields, icons, inline help, graphics, title bars, drop-down menus, borders, and individual pixels are some elements that we encounter in modern technology, and we haven't begun to exhaust the iterations of their forms.

Controls are elements that trigger behavior when prompted by the user.

Layout is how elements and groups of elements are arranged with respect to one another. When people begin to interact with a product, they try to establish familiarity with the locations of various elements. If an element changes position after a step, and feedback isn't provided to clearly express that change, people have to expend unwanted effort on finding each element again. Because of this, layout should be as consistent as possible.

Jan Tschichold defines layout as a hierarchy, which makes sense in graphic design as well:

Every part of a text relates to every other part by a definite, logical relationship of emphasis and value, predetermined by content.

Meanwhile, Paul Rand values the subtle relationships between the elements of an interface (or poster, book, advertisement, etc.):

To believe that a good layout is produced merely by making a pleasing arrangement of some visual miscellany (photos, type, illustrations) is an erroneous conception of the graphic designer's function. What is implied is that a problem can be solved simply by pushing things around until something happens.

We face the same issues in everything we make: grouping, context, and meaning. Excess frill or semantic ambiguity can send mixed messages. Kenya Hara hits on the goal:

A designer creates an architecture of information within the mind of the recipient of his work.

If two elements exist at the same time, they automatically exist in a relationship with each other. When you add a third element, each element interacts with one another, producing six distinct relationships. The complexity of this becomes daunting on paper, but we have the mental capacity to differentiate between, and infer relationships among, many elements at once.

To complicate the matter further, though, the negative space between elements matters as well. Margins should express an internal language that

2:

Consistency & Character

Jan Tschichold, *The New Typography* (University of California Press), p. 67.

Paul Rand, *A Designer's Art* (Yale University Press), p. 4.

Kenya Hara, *Designing Design* (Lars Müller Publishers), p. 156.

implies the location and significance of each grouping. Layout organizes both the existence *and absence* of elements.

Layout issues have been a focus of print design for centuries, thanks in part to the development of **modular scales**, which are mathematically constructed systems of proportions that determine the relative measurements on a page. The units of a modular scale are indivisible, and not of a consistent multiple. Bookmakers use modular scales to compose the margins of books such that they develop proportion, balance, heft, and tension. If it's worked for them for so long, why not apply that principle to an interface's layout?

Regardless of the type of content, good layout principles remain the same. These days, designers and typographers can place text blocks and titles on their canvases near-instantaneously, without any thought that their actions may have implied meaning—but doing so makes for a careless product.

Fortunately, modern trends promote more sensitive considerations of white space in layout. Many designers in both print and interactive media advocate using modular scales and typographic baselines to give text blocks rhythm. These features improve visual balance in any interface.

Let's imagine you're trying to develop an application that views any sort of published content on the web. This could include blog posts, images, video, or status updates. The application probably offers different views of that data: perhaps posts that mention the user or link back to her website, or updates from a specific feed or person, or a group of feeds. Or, if you're viewing an image gallery, the listing could correspond to a specific event, instead of sorting chronologically.

These are all sensible ways that people want to view data. And more likely than not, that data will be readable if it's displayed consistently from view to view. The title of a post, the post itself, the date that it was published, the page title, navigation, and so on should all appear in the same locations. And while the data types vary, each type should be laid out and sorted in a consistent way.

Consistency trains people to expect the position and appearance of every element in a layout. When a layout is consistent, it's easier for them to learn how to use the product, making it easier to develop a lasting bond with it.

2.2. Apply consistent behavior.

As mentioned in 1.6, a product's behavior is the way that it responds to input. Combine elements with their corresponding behaviors, and you have a complete specification for an interface.

Inconsistent behaviors bring about similar problems as inconsistent layouts do. But sometimes the solutions aren't readily apparent, or they take too much effort to build consistently, or it's difficult to change the attitudes of the product's developers.

The basics of behavior can be defined rigorously.

One example of a modular scale is the *Fibonacci Sequence*, where each term is the sum of the two preceding terms: 1, 2, 3, 5, 8, 13, 21, 34, and so on into infinity. The ratio of the n th term to the $n-1$ th term (so $34/21$, for example) approaches the golden section ϕ , or approximately 1.618, as n approaches infinity.

Robert Bringhurst, *The Elements of Typographic Style* (Hartley & Marks Publishers) and Le Corbusier, *Le Modulor* (Birkhäuser Basel) discuss modular scales at greater length.

Jan Tschichold, in 1930, from *eye magazine*, "Faith in asymmetry," <http://www.eyemagazine.com/review.php?id=151&rid=718&set=780>: "White space is to be regarded as an active element, not a passive background." Even today, it's worth considering negative space as an element unto itself.

6.3.3 discusses how to use grid systems and typographic baselines to enforce more elegant layouts.

PATHS.

People conduct a series of discrete **steps** to accomplish any task: pressing a button, navigating between form fields, entering data, etc. In succession, these steps comprise a **path** to complete the task. Multiple paths comprise one entire **interaction**, or a relationship between a person and a product, from the time when it is picked up (or opened) to when it is put down (or closed). Finally, a user's **experience** is the sum of all of their interactions.

The branches of a product's paths can become very complicated, attaining a recursive, fractal complexity. If there are many ways to begin an interaction, then you automatically have to support, and account for, those many different use cases. And as use cases branch out, you may have to account for many *common* paths. It makes sense to simplify the number of paths as much as possible, so the product is easy for people to learn and for you to maintain.

MONOTONY.

In situations where you must create different paths to accomplish the same task, they should behave identically. The best way to ensure consistency between similar paths is to pare them down as much as possible—ideally to only one path. An interface is **monotonous** if it has a one-to-one mapping between paths and desired tasks.

Monotony has a strong connection to **modes**, which are any kind of setting that someone can trigger to activate a different set of behaviors in an otherwise identical interface. Jef Raskin, who coined the term “monotonous,” elaborates:

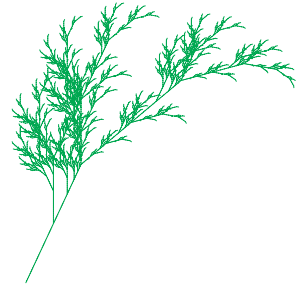
*Monotony is the dual of modelessness in an interface. In a modeless interface, a given user gesture has one and only one result: Gesture **g** always results in action **a**. However, there is nothing to prevent a second gesture, **h**, from also resulting in action **a**.*

The past few years of technology have favored monotony, but the prevailing trend has been against it. Designers and developers assume that people want freedom in their choice of path, so they provide a broad array of ways to complete a task, with preferences encouraging still more. While one can argue that reducing the number of paths is constricting, that doesn't matter if the remaining paths are obvious and learnable. It falls on the designer's shoulders to make paths easy to use in the first place.

Fundamental tasks should always be monotonous; that is, each task should have only one corresponding, unique path (commonly referred to as a **one-to-one mapping**).

Monotony is often hard to implement because of functional constraints, the elements' context, and the desires of other team members—but if elements are conveyed with as little behavioral ambiguity as possible, and if people come to a consensus about what a product should do, then monotony is feasible.

I call these *paths*, which may seem novel, but the idea has been in interaction design for a little while now. Alan Cooper calls the same thing *command vectors*, defined as “distinct techniques for allowing users to issue instructions to [a] program”: Alan Cooper, *About Face* (Wiley), p. 551.



Jef Raskin, *The Humane Interface* (Addison-Wesley Professional), p. 67.

Your keyboard's caps lock key is the most classic example of turning a mode on and off.

It's easier to enforce modelessness and monotony if you follow the precept of the UNIX Philosophy: *Make each program do one thing and one thing well*. For more, see Doug McIlroy, “Basics of the UNIX Philosophy,” <http://www.faqs.org/docs/artu/ch01s06.html>.

Modelessness is a bit easier to pull off on smartphones, due to their smaller screen and full-screen posture.

For more on the constraints afforded by a product's context, see 2.8. For more on posture, see 5.3.

Before you can design a monotonous interaction, you have to have faith that people will successfully try to learn your product. It's a lot easier to have that faith if you make the product simple. So it's a self-fulfilling cycle: making the product simpler gives *you* the confidence that people will understand it more easily, which in turn reassures you that making it simpler was the right choice in the first place. You have to have the right mindset to reach this conclusion, and it's just as easy to take the opposite road towards embracing complexity.

Designing monotony is a crucial but difficult task. Sometimes it requires resolving conflicting desires between colleagues: one wants the product to accomplish a task in X way, and another wants the task to be accomplished through Y, so they assume it's okay to compromise by using both methods—that both are somehow right. Two of the hardest parts of interaction design are recognizing that this line of thinking doesn't scale to more complex products, and having the courage to say no to these additional paths. Consensus decisions lead to less clarity than taking the initiative to say that *this* is the way things should be, offering research and testing as proof, presenting a grounded and practical solution, and expecting the rest of the team to trust that you aren't screwing things up.

AN EXAMPLE.

Let's take a look at a word processor and figure out some ways to cut down the number of steps and paths that it requires. The first thing to do is create a new document. The operating system has a FILE menu, so we move to it with our mouse and click. Then there's a NEW option. One more focus. A submenu then opens with the options DOCUMENT... and DOCUMENT FROM TEMPLATE.... Select DOCUMENT...—one more click—and a modal dialog prompts us to specify the parameters of our file: page size, margins, and whether you have facing pages. That requires another click, too.

All of these are reasonable things that we might want to modify, but they can also be changed after we've created the document, within a preference pane. We decide to change none of them, and hit OK to create the document. That makes four steps: clicking FILE, then NEW, then DOCUMENT..., then OK. And while there's a keyboard shortcut that reduces the number of steps to two (it still pops up the modal dialog), this is the minimum number that it takes to create a new document without knowing the shortcut.

Eliminating the dialog is an immediate and obvious way to shorten the path by one more step. At this point, we have a few options, depending on what trade-offs we're willing to make.

1. We can make both NEW DOCUMENT and NEW DOCUMENT FROM TEMPLATE primary menu items, which would increase the length of the primary menu, but keep the user from having to hover over to a submenu.

A modal dialog appears over the layout you're working with, rendering it unusable until prompted. For more on modelessness in general, see 6.5.

2. If the program has a toolbar for frequently-used functions, one of the two NEW options can correspond to an icon on the default configuration for that toolbar.
3. Upon opening the application, a (hopefully non-modal) dialog could pop up that prompts someone to either create a new document or open an existing document. The new dialog may even teach them the keyboard shortcut to create a new file: “To create a new document in the future, hit command-N.”

These are just three options, and more than one of them can be built. All of them reduce the number of steps to two or one—a 50% or 75% improvement—and they do so in a way that doesn’t sacrifice the program’s functionality.

This isn’t some situation that I fabricated to prove the magic of this process. This is how you create a new document in the software that I’m using to typeset this book. It’s a finished product used by thousands of people, on its ninth major version. There’s still considerable room for improvement, so that people remain in the flow of their original tasks, facing as few distractions as possible.

And it’s just one feature of dozens that could be improved this way. It’s a question of *sensitive editing*: of cutting out what doesn’t absolutely need to be there. We do this with our own writing all the time, shortening it for clarity and impact. Why not do the same with technology?

LANGUAGE.

Interfaces can be read for their underlying meaning. As people interact with a product, they subconsciously process its creators’ intent.

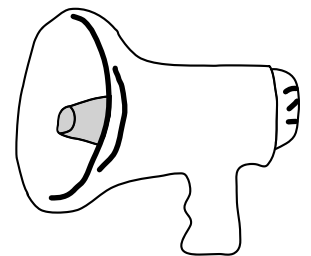
Language means more than just well-written copy. **Language**, as applied here, is the presentation of any stimulus—whether visual, auditory, or behavioral.

A product should have a consistent *visual* language. Functionally similar prompts, outputs, and elements should appear in the same places, displayed in the same ways, with the same character to them. Conversely, separate paths and separate functions should be distinguished.

A product should have a consistent *behavioral* language. Any elements that behave the same should look the same, and any elements that behave differently should look different. Functions that do not pertain to the task at hand should be disabled or invisible, and their disabled state should be immediately evident. Pliant elements should look different from unpliant elements.

EXAMPLES OF INCONSISTENT BEHAVIORAL LANGUAGE.

- *Similar behavior but different appearance.* A proprietary content management system has multiple ways to create new content: either through a



This is often referred to as a “ghosted” button or menu item, made lighter than the other options to imply that it isn’t clickable. For example:

pliant unpliant

green + link in the upper-right of most main content areas, a much larger (and more three-dimensional) + icon in a tab above the content area, or a text-based drop-down list that outlines many common functions, including an option to add new content. This leads people into believing that each way of creating new content could be slightly different, or that there isn't any way to create new data on a page lacking one of the three functions.

The best way to solve a problem like this is to keep the smaller green text link in the same place, on *every* page where content can be added, and to eliminate all instances of the other two designs. It takes up less space, is functionally unambiguous, isn't image-based, and doesn't require translation between languages. The function of the + would correspond to the type of data that's being viewed or edited at the time.

- *Similar appearance but different behavior.* Based on elements' appearance, people form expectations about their behavior. For example, you shouldn't have two square green buttons with rectangular-looking iconography next to each other if they perform different functions, such as to add or edit an item. If an element works differently, it should look different.
- *Unambiguous pliancy.* Rather than pop up an error message on controls that someone isn't supposed to access at a given time, make the element in question unpliant.

2.3. *The words in an interface are part of the interface.*

Design extends to copy, from advertising to branding to the product itself. Copy is part of the interface, and every word is a design decision. Words are used in form labels, page titles, menu names, controls, dialog prompts, and product names. The user experiences words as much as underlying functionality, and word choice should be as straightforward and elegant as the other parts of the product.

2.3.1. *Adopt a kind tone and a hopeful tenor.*

Tone can't be faked; it's a clear indicator of the product's character. The copy should be as kind and humane as the product is. It should promote a sense of hope that the product is capable of filling a customer's need.

2.3.2. *Copy should be descriptive.*

Copy should elucidate the function of an interface, rather than describing how great the product is. Words should be edited to favor practicality, each word relevant to someone's goals.

MailChimp's style guide *Voice and Tone* is a great example of how to enforce tone in a large product. For more, see <http://voiceandtone.com>.

2.3.3. Copy should be non-technical.

Messages should be sensible, natural, and written in complete sentences when appropriate. Technical error messages, with meaningless numbers or stack traces, are condescending.

A *stack trace* is a report of the active parts of a given computer program, frequently used for debugging.

2.3.4. Write real words.

So much copy, especially copy written for the web, urgently needs to be cleaned up by people dedicated to the task of creating and editing the product's mission and vision. Copy is not an afterthought: it's the first thing people look to when they try to understand what *any* product is about.

Writing "lorem ipsum" or "copy goes here" text may make for a nice-looking wireframe, but it doesn't impart any sort of utility. It's also lazy: it takes less time to copy and paste from a block of lorem ipsum than it does to think about writing real content. Real content helps you envision the product's real-world use.

In the past couple of years, the profession of *content strategist* has formed to fill the blank slates created by information architects and interface designers. For more on this practice, you should read Erin Kissane's *The Elements of Content Strategy* (A Book Apart).

2.3.5. Write in either the plural first or second person.

Make sure you use the right personal pronoun, and try to use it consistently throughout. First person involves "I." Plural first person (also called "grammatical person") involves "we." Second person involves "you."

Consider some sample feedback messages, written in different tenses. In each case, one of these messages sounds less awkward than the other two, and the singular first person sounds like the product is anthropomorphized.

I generally prefer the plural first person, but depending on the situation, you may need to use the second person.

- First person: "I'll help you create a new document."
- Plural first person: "We'll create a new document."
- Second person: "You'll create a new document."

- First person: "I didn't understand your input."
- Plural first person: "We didn't understand your input."
- Second person: "You didn't enter a recognized command."

I would use the plural first person here, because the second person is too accusatory.

- First person: "I sent your message."
- Plural first person: "We sent your message."
- Second person: "You sent your message."

- First person: "I couldn't find any updates for the applications in your library."
- Plural first person: "We couldn't find any updates for the applications in your library."
- Second person: "There are no updates for any of the applications in your library."

However, the second person works better here.

2.3.6. Apply consistent, descriptive copy.

Make sure every word in the product is vetted for consistency, and make sure that developers are provided with guidance for writing labels and feedback.

CONSISTENT LABELS.

Form labels should be consistent. If two different forms ask for an address, one should not read “Address” and the other “Home Address.”

- Use either OK or SUBMIT, not both.
- Use either CANCEL or START OVER, not both.
- POST, SEND, SUBMIT, and SAVE all connote different actions.
- Depending on which country was entered, a lookup procedure could be used to set the right nomenclature, from POSTAL CODE to ZIP CODE; as well as among STATE, PROVINCE, and TERRITORY. Use POSTAL CODE if you can’t change this term dynamically.
- Submission buttons should describe the action you’re performing and not be generic. For instance, PLACE ORDER is a more suitable button than SUBMIT.

CANCEL is more familiar than START OVER.

CONSISTENT FEEDBACK.

Feedback should always have consistent syntax and structure. If you’re formatting one message in a particular way, it stands to reason that every other message should read similarly.

For instance, if you write ARE YOU SURE? in one confirmation prompt, then you should not write CONFIRM: Y/N in another.

Feedback involves behavior as well. If you’re deleting a row from a table, one delete function shouldn’t fade the row out while another slides the row out of view. Each of these functions should do one or the other.

Ideally, though, there should be only one delete function in the first place.

CONSISTENT ERRORS.

Error messages should be formatted clearly and consistently. Here’s a set of real error messages, quoted from finished products and stripped of obvious identifiers, that show room for improvement. We can reword these so they make more sense, don’t condescend to anyone, and offer an opportunity to provide feedback.

Error. The operation completed successfully.

Internally confused messages send the wrong impression. If the operation were indeed successful, the word “Error” has no place in the message.

The application [application name] quit unexpectedly. [The operating system] and other applications are not affected. Click Relaunch to launch

the application again. Click Report to see more details or send a report to [the developer].

Error feedback after every crash is redundant. Usually, people either want to automatically report errors whenever they occur, or don't want to report errors at all.

Realistically, though, they don't care.

The application's preferences could provide someone with a setting to deal with errors: for instance, to automatically send *all* error reports to the developer, which would remove the need for a prompt every time something goes wrong.

Ideally, though, such a setting could apply to all applications in the operating system, with a way for software developers to collect all error reports for their respective products. In what way would such an option differ from this error reporting system, besides removing the continual intrusion?

Sorry, [application name] crashed unexpectedly. If you were not doing anything confidential (entering passwords or other private information), you can help to improve the application by reporting the problem.

What if the user *was* indeed doing something confidential? Is there a way to remove or obscure identifying information from the error report to protect their privacy? If sensitive information is absolutely necessary, is encryption a possible compromise?

If so, then the first clause and parenthetical explanation of that sentence could be removed. Strive to be concise; wordy responses waste time.

Beta-only warning message—not to be localized: UMESSAGE buffer full! Should never happen. Generate fewer messages.

This error message appeared in a completed, post-beta product, but post-beta customers have no need for it. If developers care about error output during beta, they should code a way to toggle verbose reporting in the source code, for when the product ships.

“Should never happen” is a sentence fragment that should be rewritten as a sentence.

Localization is a term solely used by, and relevant to, software engineers. A shipping product should generate error messages that are *only pertinent to customers*, not developers.

People have no idea what the term “UMESSAGE buffer” means—or why it matters. It should be defined or reworded, for example to: “An internal error has occurred. The message buffer is full.”

An unhandled error occurred in the GUI, further errors may be reported. -6

Errors should never be referred to by their number, even when accompanying descriptions. Diagnostic numbers help developers, not users.

While you likely know what a GUI is, most people don't.

What does it mean for an error to be “unhandled?” At press time, the powers that be haven't admitted “unhandled” into any English dictionary.

Why is the sentence broken with a comma, rather than a period or semicolon?

Worst of all, why announce that further errors may be reported? It gives you no assurance that the product works. If further errors are possible, is it time to force the program to restart?

A dialog box is open. Please close the dialog box before continuing.

Modal dialog boxes should be kept to an absolute minimum to begin with, and the need for a dialog box about a dialog box shouldn't exist in any application.

The message implies a recursion that makes no logical sense. Messages should follow the logic of the application, which people should be able to figure out.

A system error has occurred. Please contact the support team.

This message doesn't describe the nature of the error, and the product doesn't automatically report it.

A colleague sent this error message to me. He's on the support team. When diagnosing his own software, he frequently encounters this message for a variety of problems. So who supports the support team?

[application name] did not shutdown tidily. Check /home/user/.appname/logs/save for diagnostic log files and consider reporting them to the [application name] team if this is the result of an application error. Also check the Wiki (see the Help menu) for "[application name] Disappears"

This message appears when you open a program after a system crash. It assumes that you have significant experience in system-level debugging and research. Few people will expend the effort that the message requests. Programs should recover gracefully after a force quit. Error messages of this type shouldn't exist, because they're unhelpful and misleading.

Why does the error message refer to a hidden path, requiring a special search on the command line?

What's a log file? What's the difference between diagnostic log files and other kinds of log files? Are they saved in different locations?

How can people determine what kind of error constitutes an application error?

Consult the Wiki? While many people know what a wiki is, many more don't—and most won't care. Regardless, using the term "help wiki" clarifies its role.

Why and how does the application "disappear?"

Usage errors abound. What does it mean to shut down "tidily?" Why is "shutdown" one word? Why is the final sentence missing a period?

And again: why does the program lack a way to automatically report the error?

Catastrophic failure.

This was the entire message. Had I read this and not known better, I'd think a sinkhole was about to open underneath my computer and swallow it forever.

Sweat the details in every error message. You may think that a misplaced semicolon or two exclamation points ultimately makes no difference in how people perceive your product. You may think that picking out a few absurdly bad error messages is too satirical for serious critique. But complaints about poorly-written error messages abound—and these took little effort for me to find.

Detailed copy editing is important in any field where words exist to be read. Your customers will care about what you have to say to them. Meet them at *least* halfway.

For an amusing take on the subject, see Ben Zimmer, “Crash Blossoms,” *The New York Times*, <http://www.nytimes.com/2010/01/31/magazine/31FOB-onlanguage-t.html>.

CONSISTENT HELP.

Like everything else in your product, help should be consistent in its language, voice, and tone.

Information architects and content strategists help make the copy on websites consistent, readable, and navigable. People expect the same of the web's help files and support documents: they should be easy to search, common problems should be prominently displayed, and their display should be consistent from entry to entry.

When developing software, your operating system will likely already have a system in place for help documents. Use it. Don't make it a placeholder for where to find the *real* help—put the actual information there, and format it in the right way.

2.3.7. *Apply a consistent tone.*

Nobody intentionally writes with inconsistent tone, but it can creep into products when they become large enough that more than one person is charged with writing for it.

Even if you don't consider yourself a good writer, it's possible to train yourself to write good copy. All it takes is being sensitive to what your product says.

Tone is a matter of attitude. Writing, however, is a *craft* that is practiced—one learns to write like learning to play a musical instrument, or mastering a trade. Nobody is born a great writer, but many dedicated people have made themselves into great writers.

Let's look at some copy that may unintentionally give the wrong impression. Syntax errors have been some of the most common types of errors in technology since the dawn of personal computing.

They take many forms, from the terse:

- *Syntax error.*
- *Inappropriate entry.*

- *Overflow*

to the verbose:

- *No standard web pages containing all your search terms were found. Your search—[entry]—did not match any documents. Suggestions: Make sure all words are spelled correctly. Try different keywords. Try more general keywords. Try fewer keywords.*

to the technical:

- *Incorrect form: must be in form [entry].*
- *-bash: [entry]: command not found*
- *Not Found: The requested URL [entry] was not found on this server.*
- *[Application name] isn't sure what to do with your input.*

to the personified:

- *I'm sorry, I don't know what to do with that.*
- *No matches. Try being more specific.*

to the humorous:

- *Uh oh. Something very bad has happened. We're working on it.*
- *We couldn't find any results matching [entry]. We give up!*
- *This wave is experiencing some slight turbulence, and may explode. If you don't wanna explode, please re-open the wave. Some recent changes may not be saved.*

Independent of the content of these error messages, they imply different moods. Some are indifferent. Some are funny. Some are comforting. Some are useless. Some are needlessly verbose. Some don't even convey the right thing; the one where “something very bad has happened” appeared when I typed gibberish into a search blank and hit enter. How is that “very bad”?

Each of them, however, reflects on whomever wrote it. When multiple people are working on a product, each runs the risk of inconsistent tone by subtly projecting his or her intentions onto the copy. This has to be accounted for, although it's hard to effectively critique.

If you grouse about the connotations of a single word or phrase, you're viewed as splitting hairs, fighting battles over too-small details. But these things *do* matter, and they *are* worth fighting over. All successful products concern themselves with smoothing out the details of copy. As with all other parameters of your product, neglecting the copy is a big problem when your trustworthiness and reputation are at stake.

2.4. *Apply consistent graphic design.*

Because trends change over time, any specific rules about graphic design will be obsolete in short order. But no matter when you're designing, you're still building a consistent interface, not a Dadaist poster. All optical parameters

should appear unified, balanced, and coherent. They should convey ideas with as much clarity as possible.

There are countless ways that graphic design can be inconsistent, but some common problems are easily solved:

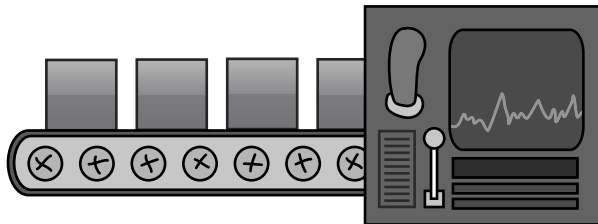
- All icons should look like they're created by the same hand.
- Don't mix typefaces at random. Better yet, use as few typeface families as possible.
- Text should have consistent letter spacing and leading.
- Use no more than three shades of the same hue when establishing a hierarchy.
- On websites, represent links with only one color.
- Favor the plain and familiar over the flashy, the posterous, the absurd.

2.5. *Don't create separate products with feature sets geared towards beginners or experts.*

At some point, *all* of your customers use your product for the first time. Still, it's condescending and denigrating to assume that the customer is a beginner. Establishing separate interfaces for different experience levels is needlessly complex, time-consuming, and doesn't effectively educate people.

Beginners tend to become intermediates quickly, because nobody invested in using a product wants to remain unaware of how it works. However, few intermediates become experts, because most people don't see the point in memorizing keyboard shortcuts or extensively customizing their tools. The vast majority of people are intermediates, and you should design with this group in mind.

In short, this means two things. First, all the fancy customization options you're building into your preferences pane will be used by a tiny minority of people. Second, your default settings will be used by the overwhelming majority. Take the intermediate path and make sure that your application is learnable, by default, without the need for training wheels.



2.6. *Reuse code, elements, and behaviors.*

It may seem like a tall order to enforce consistency in every single aspect of the product, but fortunately there are many ways to encourage it. Modern

Discard this rule if you are, in fact, making a Dadaist poster. For more on color and graphics, see 3.4.

For more on selecting typefaces and understanding their historical traditions, see Robert Bringhurst's *The Elements of Typographic Style* (Hartley & Marks).

There's one exception to this: any interface that people deliberately use for brief periods of time, like a kiosk or ATM. But the majority of products are designed for longer periods of use than these.

For more on targeting intermediates, see 1.7.

software development favors consistency and reuse, from graphic design to code.

Systematic reuse isn't *essential*, but convenience, maintainability, and consistency depend on it. You made it once; why expend the effort to make it again? Why not design it right the first time? It's faster, easier, and less expensive.

Software engineers embrace object-oriented programming (or OOP), applying reuse notions to code. Just as the Industrial Revolution gave us mechanical reproduction as an efficient way to improve our standards of living, object-oriented programming provides ways to make programming more efficient.

One pillar of object-oriented programming is the concept of a design pattern. The term “pattern” is coined in Christopher Alexander's books *The Timeless Way of Building* and *A Pattern Language* (Oxford University Press), which apply patterns to the design and construction of houses and towns in a modular, efficient, and humane way. While a series of patterns is the central tenet of both books, Alexander never formally defines the term, so I'll take a stab at it. A **pattern** is the definition of a recurring problem; the discussion of its essence and implications; and an abstracted, reusable, visually expressed design solution. Patterns interact with one another to create an entire system called a **pattern language**.

Elements are patterns of relationships that coalesce into working features. Place an element on a blank canvas and it alters the canvas; place *another* element there, and it alters both the canvas and the other element. The use of patterns is a good way to simplify, aggregate, and manage the relationships among elements, groups of elements, and their behaviors. They help you identify problems before they become too difficult to fix, and then they abstract those problems to give you a better perspective for approaching future tasks.

If you're interested in a good pattern language for interaction design, check out Jennifer Tidwell, *Designing Interfaces* (O'Reilly).

The relationships between elements is further discussed in 2.1.

HOW TO CREATE PATTERNS.

If you ever repeatedly encounter a certain kind of problem and find yourself solving it in the same way each time, it may be time to employ a pattern. There are many libraries of interaction design patterns, and creating patterns for yourself is easy as well. Establish a toolbox of commonly reused ideas—controls, processes, front-end appearances—and consult it regularly.

AN EXAMPLE.

Take a website that requires you to login. Historically, login has taken many forms. Login can exist for first-time users who'd have to create an account—or for veterans, who have no use for the account sign-up process anymore. You can log into the site through the front page, by following a link to an interior page, or after the session has expired.

So there are many problems—namely, how to log into a site from many different places, with many different intentions. Each entry point can be resolved in different ways. For the sake of consistency, using a pattern will work much better and make the product that much easier to learn. Let's make a pattern and apply it to this situation.

- **First**, *define the problem*, accounting for as many situations as possible. Defining the problem shouldn't take more than a few sentences.
- **Second**, *outline why the problem occurs* and list the situations in which it commonly occurs. Start discussing potential solutions. This should be written as concisely as possible, and shouldn't be more than a page or two.
- **Third**, *propose an abstracted, reusable solution*, which shouldn't take more than a paragraph. It may be useful to include sample graphics, wireframes, or pseudocode—but in most situations, you should have an essay that straddles the line between functional specification and mission statement. It says *here is how we will address this problem*, in any situation where it occurs.
- **Last**, *draw a diagram expressing the solution*. If it can't be expressed visually, it isn't a pattern.

Pseudocode is a rough, written description of how code should behave.

Returning to our example, here's how our pattern would look when finished:

The customer needs a way to authenticate from any part of the site.

First-time customers typically enter through the front page and sign up for an account there. We don't need to account for the tiny percentage who enter through other pages, but veterans who already have accounts need a way to easily log in from any page, including the front, and start working as fast as possible.

Login and account creation should be kept separate. Two separate groups of customers access each section. Once someone has created an account, she shifts from the group needing to create an account into the group needing to log in using their new account.

LOGIN: 808

PASSWORD:

GO

Once logged in, customers should be automatically directed to the original page they were trying to access. The login form should look identical on the front page and interior pages. Lastly, account creation should be kept slightly more prominent on the front page, because that's how new customers will begin to interact with the site.

In this example, the bold statements describe the problem and solution, respectively.

I've kept the solution general. If you were writing this for yourself, you might want to be more specific, describing the specific layout of a login form, its cosmetic features, and its behavior (i.e., whether caps lock is detected, whether the form automatically completes its fields, whether the login requires a second authentication step, whether there's a REMEMBER ME check box, whether account creation requires a confirmation email, whether a CAPTCHA is needed, etc). There's a lot to keep track of, with many various solutions for what looks like a simple feature on the surface. Write what best describes and solves *your* problem.

HOW TO USE PATTERNS.

Existing patterns are easy to find: search online for the term “design patterns,” and you'll find pattern libraries and books in many subject areas.

If you've already made your own patterns, or if you're consulting libraries of patterns that you find inspiring, you may ask yourself how they can be used when they amount to piecemeal, if detailed, requirements in a functional specification.

- **First**, *compile enough patterns* that they fully describe the product you're making. This sounds daunting, but the majority of the work is done for you by other libraries. Your problems probably aren't as unique as you think. Chances are that other people have experienced them, and written and grouped patterns to solve them. These provide a great starting point that you can customize to your own situation.

Once you have your patterns together, you can use the rest of this process to design each step. Repeat it for other steps.

- **Second**, *list every element and its corresponding actions*. In the case of the login pattern earlier, the front page and interior pages may require slightly different organization. *Label each element with a number*.
- **Third**, *group elements which affect each other directly*. For example, consider whether a login field and a password field would affect each other. One element may belong to many groups. *Label each grouping with a letter*.
- **Fourth**, *prioritize each group* based on its importance. You should have only three to five priority levels here—low, medium, and high will do.
- **Fifth**, *sketch mockups that correspond to each letter group*, in descending order of priority. Assume that these groups comprise separate interfaces. It's smart to work through this step on paper, so you can erase and rewrite quickly.

This is where your pattern library comes in. If the library is well-curated, then you should have answers for many common problems. Organizing your interface in this way frames it as a collection of patterns, because patterns are most applicable to common, sensible agglomerations

This is how it's done in *A Pattern Language* as well.

A CAPTCHA, an acronym for “Completely Automated Public Turing test to tell Computers and Humans Apart,” is a means of differentiating humans from robots when filling out forms to create accounts, leave comments on blogs, buy tickets, etc. Usually it involves solving a simple math problem, or typing a scrambled-up picture of words.

This section is heavily inspired by an essay by Ryan Singer, “An Introduction to Using Patterns in Web Design,” *Signal vs. Noise*, <http://37signals.com/papers/introtopatterns/>. For an application of this process to the creation of a building and a town, check out Christopher Alexander, *Notes on the Synthesis of Form* (Harvard University Press), where the idea of patterns is expanded on at great length.

of elements. Apply your patterns as they pertain to each group, and see how they affect your original concept of the interface's layout and behavior.

- **Last**, *combine each sketch into a rough design*. Move the sketches around so they form a layout that makes logical, hierarchical sense.

Patterns move you from idea gathering to a promising wireframe that can be useful and beautiful once designed, coded, and polished. A less tangible benefit results as well: you can use patterns to defend your decisions in a more forceful, trustworthy way.

2.7. *Employ natural mappings between controls and functions.*

The relationship between elements and behavior is defined through a **mapping**, which connects a series of elements to each element's respective control. For instance, a light bulb is mapped to a light switch. Easy enough, but how does this scale? A row of light switches may turn on various lights, or groups of lights, in a large room. How do you know what switch affects what group?

Natural mappings are a subset of mappings where a set of functions is described by an *identically organized* set of controls. Natural mappings are a fundamental way to break down cognitive barriers between customer and product.

Consider a stove's burners and knobs. In the diagram at right, the burners don't naturally map to the stove: the burners are arranged in a 2×2 square, while the knobs are arranged in a 1×4 row. This layout is common to modern stoves.

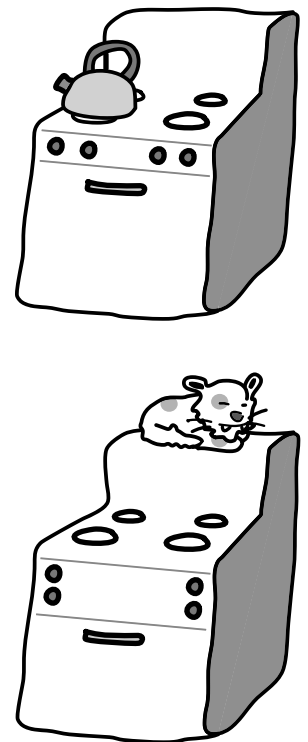
The knobs can be reorganized without losing much surface area. In the below diagram, there's no ambiguity as to which knob controls which burner. One benefit of the new arrangement is we can remove the labels next to each knob, since the function of each knob is now self-evident. This would cost about the same to build as a poorly mapped stove, and it makes for a safer stove that wastes less natural gas, because people will make fewer errors.

Natural mappings can be used in more complex functionality as well. Take the columns of an audio mixing console, for example, where each one corresponds to a different input channel. If each column is labeled correctly, there's no ambiguity as to what treble knob changes what input. Adding more channels is much easier when organized this way, and it's much easier to learn when the mapping is used universally across brands and types of mixers.

2.8. *The context of your interface should prescribe its norms.*

All objects have context. A product's context should dictate its appearance and behavior.

The term "natural mapping" was coined by Don Norman in his seminal work *The Design of Everyday Things* (Doubleday Business). He devised the stove example below.



A **platform** is the operating system or device on which your product runs. A platform acts as the frame for your product. A game console is a platform; a controller or game is designed according to its constraints. A smartphone is a platform; applications are made for it. The personal computer is a platform; operating systems are written for it, which are themselves platforms; programs are written for the platform of the operating system. A **system** is an interconnected set of platforms—operating system, input devices, central processor, screen, etc.—that you interact with.

A product's context prescribes the way that it should work. An element's context prescribes the way that it should appear and behave. The subservience of a product to its platform and system is its **normative context**.

For example, if software is written for a given operating system, it should use conventions similar to other programs written for that operating system. It's likely that the operating system's developer has published a set of guidelines to aid you. If the product is a peripheral for a computer, it should connect to the computer with the appropriate protocol. If it's a website, many specifications exist to dictate good coding, usability, and accessibility practices, and you should develop with them in mind.

People call products “Mac-like” or “Windows-like” depending on how well they fit their parent operating system. Designers and customers alike want the programs they use to look and behave like their respective systems, and they don't want to enter a separate, proprietary world with every new application that they open.

How to design *like* your platform is impossible to define generally, because it differs from platform to platform. But as a platform gains traction among developers, and programs are lauded as examples of quality design, a body of knowledge emerges about how best to adapt to the ethos of your platform's stewards.

Four months after programmers were allowed to develop third-party software for the iPhone, John Gruber wrote:

Figure out the absolute least you need to do to implement the idea, do just that, and then polish the hell out of the experience.

He then set forth five principles that apply to “nearly every iPhone app designed by Apple, and ... the ones [he likes] most from the App Store.”

In subsequent years, mobile device principles have dovetailed with the constraints that are imposed by each platform's creators. When Apple significantly changed iOS's interaction model in June 2013, they released a new edition of their human interface guidelines recommending specific tweaks that developers could perform to their applications. Within a week after its release, iOS developers began laying out what needed to be changed in their applications.

Another example: in the web's early days, the HTML and CSS specifications were the subject of considerable debate. A decade or so later, people began using JavaScript frameworks to build application-like functionality. Further debates emerged. Object-oriented CSS became popular, with two

Sometimes the product is itself a platform, but most things you make are subservient to one.

The World Wide Web Consortium (W3C) dictates most coding specifications on the web. For more information, check out <http://w3.org>.

Human interface guidelines are published by hardware and software developers to dictate the norms of applications running on all common devices and operating systems. Search for the human interface guidelines for your specific operating system to read more.

John Gruber, “iPhone-Likeness,” *Daring Fireball*, http://daringfireball.net/2008/11/iphone_likeness.

Three of these principles—minimizing the number of on-screen elements, making elements large enough to be selectable, and avoiding preferences whenever possible—are echoed in 3.2, 6.1.1, and 5.6, respectively. For example: Khoi Vinh, “Requiem for a Back Button,” *Subtraction.com*, <http://www.subtraction.com/2013/06/14/requiem-for-a-back-button>; and Mills Baker, “One Interface to Rule Them All,” *No More*, <http://nomore.metaismurder.com/post/52905285820/one-interface-to-rule-them-all-ios-7-future-apple>.

competing methods sparking even more debate. The number of major browsers shrank, but new challenges appeared with each subtle tweak to the most popular rendering engine.

In all parts of technology, new ways of doing things emerge, and then ways to do those things *well* emerge shortly after. The tools briefly precede the craft. Then new tools arrive. Given the recent pace of technological development, the whole process repeats so quickly that we don't have the chance to step back and notice it. Ishmael Reed concludes his novel *Mumbo Jumbo* with this: "Time is a pendulum. Not a river. More akin to what goes around comes around." Indeed.

Ishmael Reed, *Mumbo Jumbo* (Scribner), p. 218.

EXAMPLES OF GOOD FIT IN CONTEXT.

Responding to normative context gives your product **good fit**, or the perception that it's a natural extension of the platform and system. Good fit responds naturally to common behavior and allows a task to proceed as comfortably as possible. If a product has good fit, people will enjoy using it.

For example, new ergonomic keyboards tend to have good fit, while thick, boxy, old keyboards have bad fit. One leads to repetitive stress injuries much faster than the other. Likewise, brick cell phones of the late eighties have substantially worse fit than current smartphones, because of their size, weight, and feel.

In web design, the proliferation of smartphones and tablets in a wide array of screen sizes has catalyzed a technique called **responsive web design**, which detects the width and resolution of your browser's viewport and serves a layout that best fits your context. For example, a smartphone's layout may be only one column with relatively large type and buttons, but a desktop or 10" tablet layout may safely contain two or more columns. Images may change, too, with larger images served to high-resolution displays. And navigation models may change to fit the context: sidebars or header bars make sense on a desktop computer, while a menu icon could be tucked away on the smartphone layout.

For more on responsive web design, read Ethan Marcotte's *Responsive Web Design* (A Book Apart).

Beyond the web, native applications fit the context of our devices better. Smartphones and tablets especially favor native applications because of the many small affronts that appear after prolonged use: choppy scrolling, a lack of one-touch installation, and access to lower-level operating system commands. As a result of this, people tend to significantly favor native applications over their web-based equivalents.

The thinking behind this paragraph is indebted to Rob Foster, "Tiny Little Knives," *Mysterious Trousers*, <http://mysterioustrousers.com/news/2013/5/6/tiny-little-knives>.

Finally, special-purpose devices tend to work well for certain tasks. On the one hand, current e-readers display books well and have seamless integration with large bookstores; on the other, single-purpose digital cameras are becoming obsolete in favor of cameras built into our smartphones.

One of the best descriptions of fit comes from Christopher Alexander. In *A Pattern Language*, he describes a pattern to build a seat that's attached to a wall:

Before you build the seat, get hold of an old arm chair or a sofa, and put it into the position where you intend to build a seat. Move it until you really like it. Leave it there for a few days. See if you enjoy sitting in it. Move it if you don't. When you have got it into a position which you like, and where you often find yourself sitting, you know it is a good position. Now build a seat that is just as wide, and just as well padded—and your built-in seat will work.

Christopher Alexander,
A Pattern Language
(Oxford University
Press), pp. 925-6.

Common sense, but rarely applied; we tend to put these sorts of decisions in the hands of those who may not fully understand what we need.

WHAT TO DO ABOUT IT.

Because an operating system prescribes the norms of its applications, it's easy for people to determine whether a program looks and behaves like others. Furthermore, familiarity correlates with trustworthiness. No matter how well the program is designed, if people think that it doesn't fit with the operating system, then they'll conclude that the program is *poorly* designed.

Don't try to squeeze something into an operating system where it doesn't belong; instead, change the product so that it fits better. Hardware *and* software should be integrated in design and function.

Software can have extremely poor fit. For example, cross-platform programming languages promise easy and cheap implementation, but they can result in incoherent aesthetics, bloated code, and degraded performance. Many products have been developed with elements and behavior that belong to *no* system, with characterless graphics and unacceptably long response times.

Cross-platform technology fails because no context *can* prescribe its norms, since it refuses to adapt to its context. It tries to please everybody, and in doing so pleases nobody. Exceptions to this rule occur only by accident.

Very few people regularly encounter platforms with small market share anymore: everyone chooses from a tiny handful of web browsers, operating systems, and hardware. As a result, an increasing number of developers shun cross-platform technology, instead developing products for only a couple of the most popular platforms.

While I'm not fond of monopolies, and while it can be argued that imposing the norms of an operating system is isolationist and stifling of the designer's means of expression, maybe this sort of consensus is what's needed for the most reliable experiences, reducing the risk of a breakdown between developer and customer. Norms affect one's experience with an entire platform—and in doing so, they honor the intent of the platform's developers.

2.8.1. *Use traditional elements instead of custom ones.*

People appreciate familiarity in their interfaces. When they know where something is and what it does, it's easier for them to complete tasks.

For more on how cross-platform applications tend to fail, see Alex Payne, "Shortchanging Your Business with User-Hostile Platforms," *al3x.net*, <http://al3x.net/2011/01/15/user-hostile-platforms.html>.

Conversely, every time a non-traditional element appears, people have to expend mental effort to understand its purpose and function.

Your product's context determines its aesthetics. If an operating system provides pre-made buttons to drop into your application, then creating a different kind of button will confuse people. When developing a website, use actual HTML for text, instead of text saved as an image.

Avoid creating elements that have already been created for you by the platform's creators. Standard elements are the building blocks of your product's visual and behavioral language.

2.8.2. *Replace traditional elements only if there is a functional purpose for doing so.*

Often, you'll find that your platform and system don't have the ability to sufficiently describe your product's desired features. In this case, it's reasonable to create new elements.

Appearance is not the same as function, though. Don't replace traditional elements to fit your branding; only do so if you need to create new utility that the system can't sufficiently provide on its own.

2.8.3. *Information is molded by where and how it is displayed.*

Context determines the way that your product displays *qualitative* information (anecdotal evidence, words, ideas) and *quantitative* information (numbers, data). In each case, your platform may suggest an optimal scale, layout, behavior, typography, or data-ink ratio. Following these guides makes data more readable and honest.

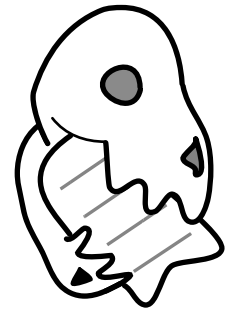
That said, the normative context of information is sometimes the product itself, not the product's platform or system. Ignoring the product affects information's scale and meaning. Good interaction design involves understanding how and where to mold information to fit its layout.

In the case of data plots, there's considerable research favoring certain design standards to make information more readable and honest:

- The *cause* should be on the x-axis, and the *effect* should be on the y-axis.
- If you're working with Latin text, labels should read from left to right, like normal text in a book—not vertically or upside-down.
- The width-to-height ratio should lie between the golden ratio (approximately 1.414 : 1) and the golden section (ϕ : 1, or approximately 1.618 : 1).
- Most quantitative plots should be wider than they are tall, and they shouldn't be excessively short on one side.

If your data won't fit well, it's better to rework the design to fit the data. Don't carelessly put the data somewhere that it doesn't belong.


The interactive data displays on journalism sites are great examples of designing to fit data well. Hovering over areas of maps pops up information



The portions of *Cadence & Slang* about information display owe an enormous debt to the work of Edward Tufte. His four books, *The Visual Display of Quantitative Information*, *Envisioning Information*, *Visual Explanations*, and *Beautiful Evidence* (Graphics Press) have redefined modern perceptions of data display.

about given locations. Hovering and clicking graphs explains the details of a narrative. Almost all of these displays are separate from any sort of *text*-based story; many are posted as their own features.

Or consider stock quotes on financial trackers. In most cases, financial data display is the entire product; no frame is necessary for the data, because *the entire page* qualifies as quantitative information, organized and displayed to imply a narrative. Sliders exist to alter scale; contextual flags explain sudden changes in value. As someone hovers over parts of the graph, market capitalization, closing price, and share volume dance in a sidebar.

On the opposite end are **sparklines**: intricate, word-sized graphics that can fit into a paragraph of text, such as . These describe appropriate, short-term time narratives in significant enough detail.

Displayed in the right way, data is capable of telling these dense and complex narratives, and it deserves the same attention as any other part of your product.

2.9. *Input devices should fit the function and the person equally.*

I can't run the 100-meter dash as quickly as an Olympic sprinter can, but I can probably hold things with a stronger grip than my 88-year-old grandmother. My vision is about 20/45, which is worse than my sister but better than my father.

Human factors is the science of fitting what we use to human capabilities. The human factors of your platform's hardware should fit well with your customers and the tasks that they're trying to perform.

Touch screens are less appropriate for precise drawing than stylus tablets, especially for the fat-fingered. Qwerty layouts are more appropriate than Dvorak for the vast majority of people who haven't bothered to learn Dvorak. And many pointing devices—mice, trackballs, etc.—can't be used by handicapped people.

AFFORDANCES.

Psychologist James J. Gibson defined an **affordance** as “an action possibility inherent to a given environment or object” or, more concisely, what something allows you to do with it.

We go through our whole lives taking advantage of various affordances without consciously realizing it. Think of a cantilevered handle on a tea kettle that automatically opens the spout when you pick it up. Or shovels with long handles and wide blades that give us more leverage to move heavy loads of snow. Or custom-molded grips for vegetable peelers. Or a corkscrew, redesigned so it grips the bottle and pierces the cork with almost no effort.

Affordances allow us to work with the world more easily. They adapt objects to us. But affordances that work for some people may not work for others: for example, if you design a steep staircase, an adult can climb it more quickly, but a small child cannot.

ProPublica's style guide is tremendously helpful for any kind of interactive data display. It's available at <https://github.com/propublica/guides/>.

For more on sparklines, see 6.4.1.

James J. Gibson, *The Ecological Approach to Visual Perception* (Psychology Press).

For more on affordances, especially as they pertain to software, see 6.1.1.

Make sure that your product affords the right things for the right people. If you're designing hardware, ask yourself if the buttons are easy to press; if the device is easy to grip; if things are laid out and formed in a way that ergonomically and naturally fit our capability. If you're designing software, ask yourself if buttons are easy to find and click, if it's possible to run through forms quickly, if the layout implies functional clarity.

2.10. *Provide for disabled users.*

Don't alienate people because their needs are incompatible with your priorities. Using a full-page Flash application for your website excludes the blind, who rely on screen readers. Using 8-pixel-tall text with no obvious way to change its size doesn't help people with vision impairment. These are only two of the many ways that the world can further marginalize those with disabilities.

Many agencies try to make technology more accessible. In the United States, the Americans with Disabilities Act (ADA) provides legal incentive to create accessible products. Section 508 pertains to both software and hardware, with guidelines that are applicable to anything you create. Other countries have adopted similar measures.

We owe it to ourselves and our society to develop and support accessible interfaces. Simplicity and clarity get us close, but they're no substitute for a continued, conscious effort.

Section 508's official site is <http://www.section508.gov>.

For more on international accessibility guidelines on the web, a list exists courtesy of the W3C at <http://www.w3.org/WAI/Policy/>.

3.1. Find the simplest complete solution.

Products should be simple. Simple products are easier for developers to maintain, and easier and more fun for people to use. By reducing the number of required steps in a task, incrementally disclosing information, eliminating redundant content, and hiding complexity where it's impossible to eliminate it entirely, we move closer to the ideal of instinctual usability.

There is an essential tension between **simplicity**, which is the thoughtful reduction of unnecessary elements, and **completeness**, which is to have enough function to be useful. Products must remain functionally complete through their fundamental tasks. The product must do precisely enough, and its creators must know where and when to stop.

You can't easily make a complex product into a simple one, but you can try to convey the feeling that it is. The *impression* of simplicity is powerful on its own, and functions well as a compromise.

It's also possible to oversimplify, so that products lack utility. Removing *essential* features will force people into unexpected situations, leaving them unable to understand how to operate the product. Removing *inessential* features illuminates and strengthens paths.

3.2. Be clean.

Here, something is **clean** when the ratio of ink or pixels dedicated to *function* is maximized relative to that dedicated to *form*. But since this can be qualitative, it's hard to formulate a process to enforce it in interfaces. The well-trained interaction designer knows it when she sees it.

In the world of information display, Edward Tufte coined the terms *data-ink* and *chartjunk*, advocating to maximize the data-ink ratio by erasing and lightening axes, eliminating (or lightening) trend lines and grids, and removing anything inessential to perceiving the information. Technology would benefit from adopting the same sentiment.

We can go further, though. It's useful to differentiate between true cleanliness and perceptual cleanliness. In real life, we assume a room is clean when it has well-scrubbed counter tops, vacuumed carpets, inoffensive odor, and lack of clutter. But if you hide a bunch of crap underneath the flap of your sofa, the room only *looks* clean.

Cleanliness should pervade your product, front to back. The code *and* the appearance should be clean. If code is reused, take advantage of object-oriented programming to repurpose it. If code can be made more readable and elegant without negatively affecting the product's performance, clean it up.

Elements should be unambiguously placed. The layout shouldn't be crowded, or have too much white space. Using traditional elements gives the perception of cleanliness by virtue of familiarity. Clean products have instantaneously recognizable function.

3:

Simplicity & Clarity

The simplest complete solution is first referenced in Alan Cooper, *About Face* (Wiley).

3.7 expands further on this idea.

JUSTIFYING THE EXISTENCE OF ELEMENTS.

Everything that you add to a product should be justified in terms of its appearance, location, and behavior. All elements should have **roles** associated with them: in other words, they should serve some functional purpose for the product. Periodically audit every detail of the layout for what it does to help:

- *“This vertical divider is here to separate the search form from the login form.”*
- *“This button is here because the user needs a mechanism to reset their custom page settings.”*
- *“This text area is here, in this size, to encourage feedback.”*

If you can’t justify an element’s existence, remove it.

TRADITIONAL ELEMENTS.

Use the elements that your normative context prescribes. This can help ensure a feeling of cleanliness because the creators of your platform—be they the programmers of your operating system or the designers of your hardware—likely put considerable thought into creating basic, familiar elements that get out of your way as much as possible.

For more about traditional elements, see 2.8.1, 2.8.2, and 2.8.3.

NEGATIVE SPACE.

Negative space should be consistent and generous. Let the elements on your canvas breathe, so they can differentiate themselves from each other as much as possible. This provides less ambiguity and more clarity.

For more on negative space, see 2.1. For more on arranging elements consistently and simply, see 6.3.3.

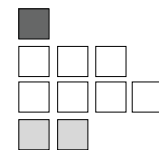
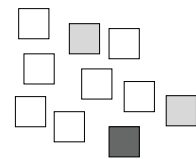
MEANINGFUL HIERARCHY.

The meaning of your interface is implied by the arrangement of its elements. Establishing a hierarchy of what’s important on a page indicates what features matter the most. It also defines elements better, conveying the product’s broader theme. Conversely, downplaying or hiding an element can imply that it’s less important.

In any product, hierarchy can be established by emphasizing (or de-emphasizing) specific elements. This leads your eye in a more focused way than, for example, a layout where all the type is the same size, face, and weight.

Hierarchy can also be established by grouping elements, and then emphasizing (or de-emphasizing) the groups.

On websites, a hierarchy can be established on a global level with primary navigation, and on a per-page level with layouts that follow reading patterns.



Scattered and grouped elements.

FEATURE CREEP.

Feature creep is what happens when unnecessary features are added to a product before it ships. Many stakeholders may try to push through various concepts for an otherwise simple product, and designers compromise by implementing all of them.

Simple products are easier to learn, and cluttering them is difficult to undo once any new feature has gained traction. Simple products must resist feature creep in any form, and you have to be critical of any suggestions that may not suit the product's core goals.

3.3. *Eliminate excess graphics.*

Graphics are tremendously useful for establishing and reinforcing branding, offering a consistent experience, and promoting a fit and finish in the final product. But graphics shouldn't replace traditional elements unless there's a functional reason for it; and graphics shouldn't overwhelm the product, because that risks distracting people.

Interactions should be kept separate from graphic design. Only use graphics if they fulfill a functional purpose, contributing to the product's interactive character. Remove any graphics that don't aid in someone's interaction.

Disputes between stakeholders and designers should favor the conventional and familiar, the tried and tested. In a world where everyone is trying to shout loudly, simplicity and elegance speak forcefully.

THINNER AND LIGHTER LINES.

Frequently, borders, grids, and graphs are too thick or too dark. Lightening and thinning them directs attention to the content, which encourages people to understand and interact with it.

BACKGROUND AND FOREGROUND.

Your background connects with all elements at once, which drastically affects impressions. Calming the background of an interface draws attention to information that you want people to see.

CUSTOM BUTTONS.

Buttons are one of the most common and important controls, because they often confirm input or move you to the next step of an interaction. You should follow your platform's context when creating them. The shape and size of buttons should be determined by their context first, and their relative importance second.

Be sensitive to using iconography on buttons. Icons should universally imply the right function. For example, using an icon of a floppy disk to

1.15 discusses why resisting complexity is important as your organization scales.

For more on this, see Jared Spool, "Experience Rot," *UIE*, http://www.uie.com/articles/experience_rot/ and Whitney Hess, "No One Nos," *A List Apart*, <http://alistapart.com/article/no-one-nos-learning-to-say-no-to-bad-ideas>.

Paul Rand, *A Designer's Art* (Yale University Press), p. 213:

In the frantic hope of "standing out," he tries to shout out, outcolor, and outglitter his competitor. He approves gaudy color schemes, oversized or misshapen lettering embellished with outlines, double or triple shadows, pseudo-Victorian decorations, and other exhibitionistic devices.



describe a save command no longer makes any sense: nobody uses floppy disks anymore, and many people have *never* used them. There are many suitable alternatives, including a text label.

Combining text and icon in one button is a good way to train people on the meaning of the icon, while providing a larger target for them to trigger. It may be useful to provide the option to view just text or just icon in a product's preferences.

LEGAL COPY.

Keep legal copy out of the way whenever possible. Move it into a separate page, away from the interaction. Word it in plain, simple, unambiguous language whenever you can.

If your product requires authorization of a contract at setup, don't force someone to scroll to the bottom, or require a minimum time limit before the OK button becomes pliant. Instead, alert them that account creation implies acceptance of the terms, and link them from there.

ADVERTISING.

Ideally, a product should have some way to make money on its own. Only use ads in sparing, limited circumstances. If ads must exist in your product, advertising is necessarily part of the interaction. If you alter a product with any advertising, you are making design decisions. Advertising should be trustworthy, pertinent, not subversive, and separate from the interaction.

Advertising is sensitive ground: people hate being marketed to, especially if it gets in the way of their tasks. People know when they're being marketed to rather than being given actual content. It's vitally important to maintain the trust of your customers when advertising to them—and especially when advertising products outside your control. Poorly executed advertising builds frustration and distrust as it wastes people's time—and one day, the negative impressions will build up enough that they will stop using your product.

LOGOS AND BRANDING.

Branding can include your product's logo, icon, tag line, or word mark. Each of these should imply the core functionality of your product. For example, if you're creating an application for gardeners, maybe your logo's colors should be green, or in bright primary colors that connote flowers in bloom. Or it can be something different or muted, so it can stand out from competitors.

No matter what, you should keep your branding as simple as possible; the best logos are intensely expressive with only a few elements, and can survive harsh printing conditions and poor screen resolutions. Many successful brands don't even directly discuss the product itself.

For more on logo design, take a look at Per Mollerup, *Marks of Excellence* (Phaidon).

On the other hand, poorly executed branding could contract the message you're trying to convey, or it could render badly on the platforms and systems you're trying to target.

3.4. *Employ a neutral and balanced color palette.*

Color should be employed with the same care as any other aspect of the product. Just as you shouldn't use ten shades of the same color when only three shades would do, the colors you *do* use should be neutral and harmonious with each other. Bold colors and stark contrasts should be used sparingly. Eliminate any colors that clash, that perturb the eye in ways unrelated to the product's function.

Consider the ways that color can vary:

MANY COLORS.



BRIGHT COLORS.



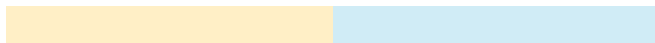
DISSONANT COLORS.



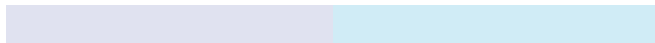
CONSONANT COLORS.



MUTED COLORS.



FEW COLORS; FEW HUES.



Which of these palettes is easier on the eyes? Which is harsher? What do these colors say when placed next to one another?

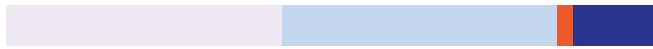
Alan Cooper, *About Face* (Wiley), p. 162:

A program may be bold or timid, colorful or drab, but it should be so for a specific, goal-directed reason.

Or, for example:



versus



These contain the same four colors in different proportions. The narrower orange bar implies a role of an accent color. This mutes the color palette, with more sparing saturation. The amount that each color is used matters as much as its value.

3.5. *Omit needless words.*

Words should only exist if they fit the product’s purpose. Long, opaque words should be replaced with short, simple ones. Excess words should be removed.

One way to omit needless words is to speak as much of the interface out loud as possible. If any sentences or fragments sound clunky or awkward to you, someone else will probably think so too.

Another way is to surround any problematic fragments in brackets. Then, go back through the product and find ways to improve those phrases.

Also, consider writing the interface in a plain text editor before wireframing it. This places greater emphasis on the precise words you use, stripping out any other distracting elements.

At first, it’s likely that you’ll frequently pare back long chunks of text. But as you practice this skill, you’ll find yourself writing more concisely and clearly—and striking out less.

Let’s rewrite some of the error messages from 2.3.6 to be as concise as possible:

The application [application name] quit unexpectedly. [The operating system] and other applications are not affected. Click Relaunch to launch the application again. Click Report to see more details or send a report to [the developer].

Sorry, [application name] crashed unexpectedly. If you were not doing anything confidential (entering passwords or other private information), you can help to improve the application by reporting the problem.

Each of these could be reworded as “[application name] quit unexpectedly. An anonymous error report has been sent.”

Beta-only warning message—not to be localized: UMESSAGE buffer full! Should never happen. Generate fewer messages.

“Sorry, but the message buffer is full.”

An unhandled error occurred in the GUI, further errors may be reported. -6

If you ever took a writing class, you will probably recognize this rule. This comes verbatim from William Strunk and E.B. White, *The Elements of Style* (Longman); in many ways, “omit needless words” is a distillation of that entire book.

Strunk says writing should have no needless words or sentences “for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts,” which is so pertinent to this book that it’s almost clairvoyant. Strunk also advocates the active voice over the passive one, and positive writing over negative—true when writing for any products.

More good writing techniques are in William Zinsser, *On Writing Well* (Collins).

See 2.3 for why a positive tone is essential in writing for technology.

Rewording this to “Graphical error” or “Rendering error” is more descriptive and accurate.

[application name] did not shutdown tidily. Check /home/user/.appname/logs/save for diagnostic log files and consider reporting them to the [application name] team if this is the result of an application error. Also check the Wiki (see the Help menu) for “[application name] Disappears”

This has no relevance to a customer’s interaction, and should be eliminated entirely.

3.6. *Keep transitions short.*

Animations keep an interaction fluid from step to step, teaching someone what to expect as they proceed. When executed well, animations can be usefully expressive, but there are many ways animation becomes distracting. Keeping them short—under 100 milliseconds, and at a high frame rate—helps ensure that they won’t get in the way. They should move from the first state to the next without any irrelevant content in between.

It’s telling that some of the best interaction design and user experience companies have begun hiring designers with animation or video experience. Operating systems now use animations to minimize and close windows, scroll through menus, and track progress. Smartphones employ animations to open applications, delete items in a list, and email photos. While animations aren’t strictly essential, they provide utility and beauty.

3.7. *Eliminate chartjunk.*

As implied in 3.3 and 3.5, the rule “omit needless words” applies equally to graphics. So it is with data, too. Data should be approached in the same way as the rest of your interface: a spare layout forms the frame that dictates the context of the information within.

Data is often considered boring, but it’s only boring if you have a boring story to tell—or if you tell it boringly. Presented with clarity, data tells stories. Sometimes, the stories are just as riveting as those set to prose.

Edward Tufte coined the term **chartjunk** in *The Visual Display of Quantitative Information* (Graphics Press) for any elements that don’t aid in comprehending data. Eliminating chartjunk leaves the designer with the essence of data, called **data-ink**: a scale, data points, and some implied relationship.

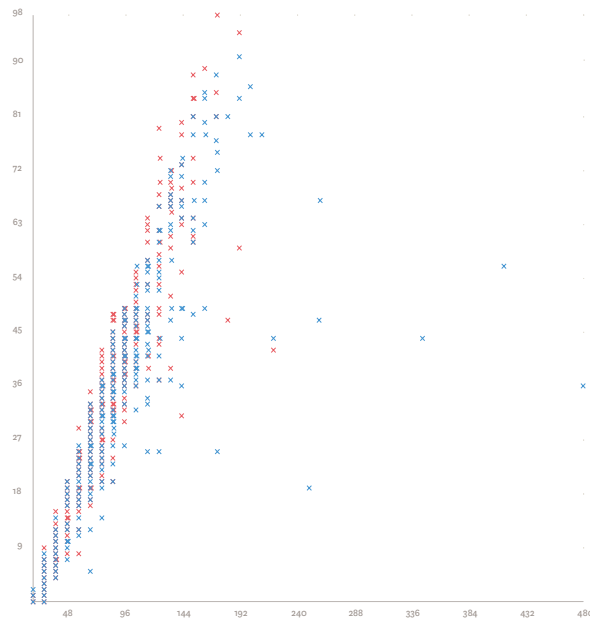
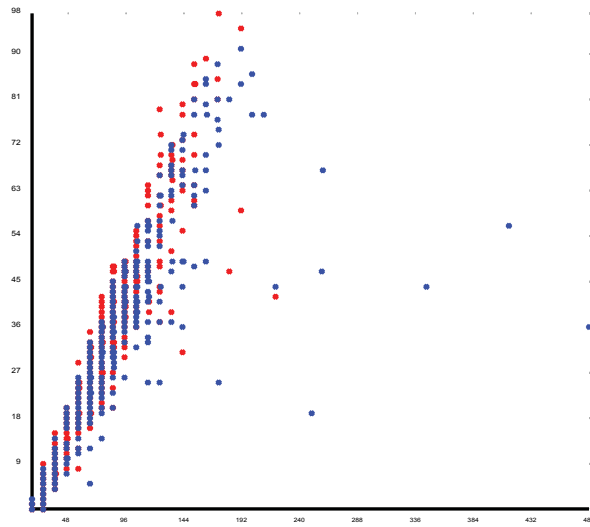
Eliminating chartjunk mutes color; it erases liberally. This provides many benefits. Data becomes faster to display, easier on the eyes, and more immediately understandable.

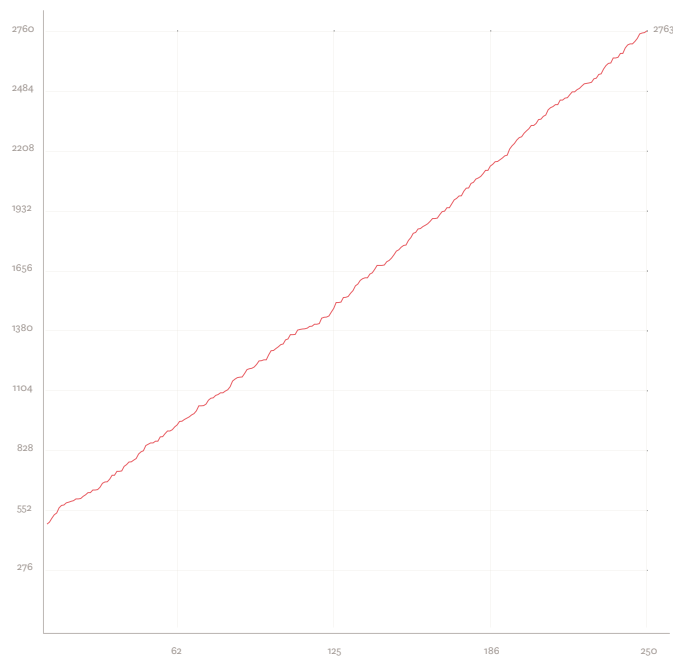
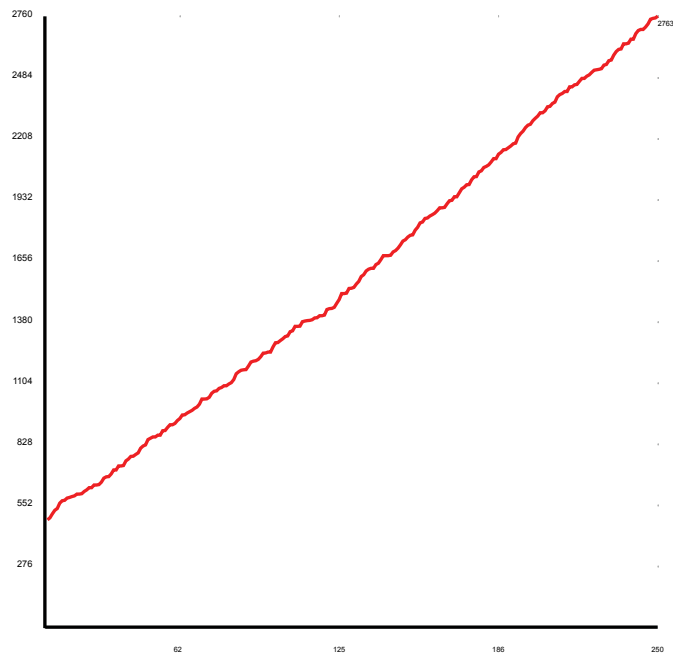
For example, Tufte describes a map of Napoleon’s army as it invaded Russia in the 17th century as “the greatest statistical graphic ever drawn,” as it detailed the position of the army and its dwindling size as the temperature dropped below freezing.

For context, see Frank Jacobs, “Vital Statistics of a Deadly Campaign,” *Big Think*, <http://bigthink.com/strange-maps/229-vital-statistics-of-a-deadly-campaign-the-minard-map>.

Data-ink, first mentioned in 3.2, is any element that serves the purpose of expressing data. Lots of graphs use extraneous elements or stylistic flourishes to convey data, which reduce the display’s *data-ink ratio*.

Tackling *chartjunk* in practice. Here are some example graphs, pulled from Brian Venn, “Render dynamic graphs in svg,” *IBM developerWorks*, <http://www.ibm.com/developerworks/xml/library/x-svggrph/>, with my redesigns:





3.8. *Replace metaphors with abstractions.*

Technology is reductive. It simplifies the real and the tangible into something that is both machine- and human-readable. This process is called **abstraction**. It begins with something real, something that's *only* human-readable: handwriting, placing a phone call, or turning on a TV. Then you remove elements from it, moving towards the essence of the original idea. It stops somewhere in between, though, where the original concept fits the machine's context in a way that remains recognizable to us.

Consider the essential functions of a given device. For example, what does a TV remote *need* to do? Turn the TV on and off, certainly. Change the channel and the volume. Beyond these essential functions, anything else can be argued as excess, including more complicated ways to accomplish each of these tasks.

Metaphors are anathema to this process. Using a picture of a desk to represent a desktop, a three-dimensional rendering of an office for an office application, a snooty butler that fetches search results, or an anthropomorphic paperclip that yells at you every time you create a new document, does not make a task simple. Each of these are distracting to people, who think: "How on earth is a butler going to help me search?" In short, your software should look like software, not like an object in the real world.

It's tempting to think that literal metaphors make a product more familiar and inviting. But there are many other ways to make interfaces inviting, and many familiar tropes of GUI design—how to use a mouse, or the location of functional menus—have already been taken as axioms. Worse yet, real-world metaphors could be interpreted as condescending by intermediate or expert customers—and those are usually the first people who buy a new product.

Making an interface graphically and functionally simple should remove the need for any metaphors. Basic tasks should be abstracted in ways that fit your product's context.

For more on context, see 2.8.

For more on interface metaphors—and the type of metaphor that deliberately changes the appearance of software to look like something else, called *skeuomorphs*, it might be best to read about them as applied to Apple, in John Maeda, "The Future of Design Is More Than Making Apple iOS Flat," *Wired*, <http://www.wired.com/opinion/2013/06/the-future-of-design-is-more-than-making-apple-ios-flat/all/1>.

3.9. *Eliminate title popups and splash screens.*

Splash screens waste time. Nobody wants to see a prompt that introduces something they know they've already opened. Additionally, many platforms and systems have built-in ways to show your program or website is loading, which removes the need for you to implement that yourself.

Some platforms require that your product have a startup screen. If so, you should display the product's blank slate, before any information has been entered.

3.10. *Only require account creation when it provides a clear benefit to the customer.*

Sometimes products will ask you to register at first launch. Others will have you create an account on the company's website.

Account benefits should connect with how the product works. If you're releasing a GPS tool that tracks people's whereabouts, it makes sense to require an account to tie their identity to their progress. Or if you're making a web-based task manager, then accounts are the best way to access your data. In either situation, **gradual engagement** is a good way to unintrusively ask for information. Gradual engagement encourages you to ask for information as your customer begins to use the product. If you run a web application, people might hesitate to create a free trial of a paid service. So, you can have people use a demo of it before you ask for their email address and create an account. Or if you are taking orders on an e-commerce site, don't make people sign up for accounts before purchasing; instead, send a link to create an account with their confirmation email, so they can order more easily next time.

If mandatory account creation only benefits your business, then it's just as useless as the splash screen. If you're using accounts to mine demographic data, there are much better ways to accomplish that. For example, surveying some customers every year will give you broad ideas about who is using your product.

3.1.1. *Sketch first.*

You haven't found all the answers until you sketch. Sketching helps you generate many ideas more quickly than any other method.

Sketches should always convey basic layout, but they may need to be arranged as a storyboard in order to show behaviors. Or multiple sketches could show various input states, like the placement of different error messages, form validations, or confirmation prompts.

The best sketches double as paper prototypes for rapid usability testing. This can take place with as much formality as needed: recruit people, take them through the screens, and have them try to complete a task.

For large projects it's sometimes necessary to annotate sketches, calling out specific elements and adding descriptions of their intended behavior. Most of the time, however, if a sketch is done well and the design is clear and understandable, then it shouldn't need any further explanation. Regardless, individual frames of a storyboard should have captions so you can stay organized.

Anyone can sketch an interface. Useful sketches express a solution to the design problem at hand and provide a clear understanding of the project to anyone not involved with it.

Sketches should be conducted as quickly as possible, but don't sketch so fast that you won't remember what an interface does when you look at it a month later. Also, don't sketch so fast that you have to spend undue effort to explain your decisions to others. A good sketch should be tidy, and it should convey its functionality without requiring further explanation.

For more on gradual engagement, see Luke Wroblewski, *Web Form Design* (Rosenfeld Media), as well as "More on Gradual Engagement," *LukeW Ideation + Design*, <http://www.lukew.com/ff/entry.asp?1130>.

For more on how to conduct usability testing on a budget, see 4.1.11.

3.12. *Create real interfaces second.*

Simplicity applies to your design process. Creating real interfaces as quickly as possible reduces the number of steps in your work, and it builds consensus more quickly.

Functional specifications and wireframes don't accurately reflect the final product. Clients rarely know how to critique them, and people never see them. A working prototype is the most useful kind of functional specification, and it should be sought out. Building something real gives more form to your ideas—something with accurate shape and behavior, to solve future problems.

See also Meagan Fisher, "Make Your Mockup in Markup," *24 Ways to Impress Your Friends*, <http://24ways.org/2009/make-your-mockup-in-markup/>; and David Cole, "Designers Will Code," <http://irondavy.quora.com/Designers-Will-Code>.

4.1. Understand expectations.

Harvard Business School professor Theodore Levitt:

People don't want a quarter-inch drill, they want a quarter-inch hole.

4.1.1. People view interfaces in terms of what they need to accomplish.

For a long time, **user-oriented design** was the most accepted practice of making products easier to use. People believed it was important to understand potential customers before they could understand what technology to make. Designers paid attention to their customers, augmented with demographic research, contextual inquiry, and stakeholder interviews.

This is still important, but the starting point has changed. Many of us are now creating **goal-oriented design**, where designs are framed in terms of the customer's goals. We dictate the steps that people should go through to complete a task, and design from there.

A major premise behind goal-oriented design is that *people don't care about what makes technology work*. When someone sits down with a computer, for example, they think "I need to send an email," not "I need to click these buttons in order to send an email." And when people see your product, they should say "this helps me deal with this problem I have."

They don't know what goes into making a computer or a phone. And they're right: ultimately, it doesn't matter. The computer or the smartphone is a tool—and technology is all about tool-building.

Many experts believe that technology should be an end in itself—and they tend to be the ones to develop new things. But this thinking corrupts the intent of a product, adding undue complexity to our lives. Why *would* people think about how a CPU is made? Why would *anyone*? What does it matter to our neighbors, to our families?

4.1.2. Know your audience.

Even with clearly articulated goals, you still can't make relevant products if you don't know your customers and what they want. We create things based on what we know. And often, products are created when we have too little insight into customer needs.

Consider the dot-com boom and bust from 1998 to 2001. Most technology businesses were based in San Francisco, a dense urban area with a rigorous street grid. Many of the resulting products focused on home delivery or e-commerce. The party line—you can get *anything you want* online!—was marketed to the entire country, even though most cities were not as dense or walkable, or as likely to support a significant delivery infrastructure, as San Francisco. Many services in the first iteration of the web were incompatible with the way that most of America works.

4:

Managing Expectations

4.1's quotation is from Clayton M. Christensen, Scott Cook, and Taddy Hall, "What Customers Want from Your Products," *Harvard Business School Working Knowledge*, <http://hbswk.hbs.edu/item/5170.html>.

More broadly, Christensen coined the term *jobs-to-be-done* as a way of framing good market fit. For more on jobs-to-be-done, see Stephen Wunker, "Six Steps to Put Christensen's Jobs-to-be-Done Theory into Practice," *Forbes*, <http://www.forbes.com/sites/stephenwunker/2012/02/07/six-steps-to-put-christensens-jobs-to-be-done-theory-into-practice/>.



Design problems like this one can be solved through **ethnographic research**, a way to determine the cultural, aesthetic, and social values of a demographic by observing and interviewing them. You should go out into the real world, observe people using products in their environment, and ask questions about what they do and why they do it.

When research is done right, you'll quickly discard bad ideas and focus on what fits your customers' needs well. Then you can refine those good ideas into an internally consistent mission. Combined with user testing, this forms the beginning of more ambitious designs.

4.1.3. *Know your technological context.*

The tools people use today will be obsolete tomorrow. Desktop computers are currently undergoing a massive shift, brought about by touch-based interfaces on tablets and smartphones. Twenty years before this book's publication, most people didn't even use the web.

Figure out what exists right now, what people like, and what people consider outdated. Good designers understand what's happening right now; great designers understand what's likely to happen in the future.

4.1.4. *Know your expectational context.*

Our perspectives are the sum of our past experiences. When someone uses a great product, that experience will color their subsequent experiences. And when they have bad experiences, future impressions of similar products are may be tainted by that negative impression.

As our devices' capabilities improve, our expectations also change about what they can provide. Once people have worked with a better product than one they're used to, going back to the original product tends to make them more frustrated than if they had stuck with the original product.

Finally, young people tend to figure out new interfaces faster than older people, because they have fewer preconceptions about how an interface should work.

All of these factors need to be accounted for, so we can make products that fit people's needs as well as possible.

4.1.5. *Review current research.*

New research is constantly being generated, both within and outside of academia, and the more you review, the more insight you'll have into your own work, as well as how other people design. Design-related books help, of course, but related topics in cognitive psychology and social anthropology will provide valuable insights as well. Seek out the biggest library in your metro area. Spend a few weeks there; bury yourself in books, online databases, and recent academic journals. And if you don't know where to start, your local librarians are there to help.

Arguably, they still are. From 2013, George Packer, "Can Silicon Valley Embrace Politics?," *The New Yorker*, http://newyorker.com/reporting/2013/05/27/130527fa_fact_packer?currentPage=all:

It suddenly occurred to me that the hottest tech start-ups are solving all the problems of being twenty years old, with cash on hand, because that's who thinks them up.

For more on real-world research, see 7.7.

Many private-sector companies, such as Xerox, Procter & Gamble, and Microsoft, have in-house research departments that publish their new findings to the design community at large.

4.1.6. Avoid theoretical research.

A lot of research disconnects designers from customers, talking about notional future desires, and notional interfaces that could fit those desires. This is a great way to generate new ideas about interactions. It inspires future design methods, and it's helpful to frame complex problems. But it's not so good for designing products that you want to ship today.

For example, touchscreens existed for many years before the iPhone was announced, and many research articles explored issues around implementing touchscreen computer interfaces during that time. But touchscreen research didn't matter for people developing mobile phone applications, whereas the iPhone's release mattered quite a bit.

Theoretical research should be treated with careful skepticism. Make sure to understand all of the research's attendant limitations, and rigorously test your implementations.

4.1.7. Avoid group research.

Focus groups frequently suppress the will of the minority or the timid in favor of those who talk first or are most gregarious. Sometimes people quash edge cases, such as expert features that might benefit only a small subset of users. Sometimes people push edge cases as if they were common. Regardless, large groups acknowledge group needs while downplaying individual beliefs. One-on-one interviews are always preferable.



4.1.8. Interview stakeholders.

Stakeholders are the people with any business investment in the product's success. If you work as an in-house designer, your stakeholders are any coworkers affiliated with what you're designing. If you're a consultant or freelancer, your stakeholders are your client's employees.

Stakeholders have strong opinions on the project's financial, technical, and logistic constraints, and they can assess whether your ideas fit the company's mission. Taken in sum, their opinions represent the company's goals and challenges, and the relationships between the two.

You need to actively listen when interviewing anyone. For more on this from an engineering manager's perspective, see Michael Lopp, "You're Not Listening," *Rands in Repose*, http://www.randsinrepose.com/archives/2012/08/28/youre_not_listening.html.

It takes practice and fine-tuned diplomacy to negotiate with stakeholders. Organizations are often thorny, complicated, and full of power struggles. The people you work with could be several levels of management away from those who give final approval. Or the decision makers could be close by, but they stay silent until after the work has been done.

No matter how it's configured, you need to understand the business you work with. Interview as many people as it takes to gather a complete portrait of the way the organization works—ideally one-on-one.

SOME STAKEHOLDER ROLES.

Every organization has a unique set of roles, but here are some of the most common ones:

- *Executives* run the company. They're the best to talk to about the company's culture, the large-scale vision that inspires employees. They're also good to consult about the structure of the company and its competitors.
- *Product managers* keep design and engineering on track. They can also act as **subject matter experts** who are well-versed in not only your product, but the context that surrounds it: the industry, its competitors.
- *Engineers* write, test, and debug the product's code. They have a good sense of what technical challenges may face the product's creation, as well as what issues you may face in building out additional features in the future.
- *Quality assurance* (QA) people test the code that engineers write—and, in organizations that prioritize design, they'll also test your recommended interactions.
- *Designers* determine how the front end will look. They'll be able to talk about the history of the brand, and what cosmetic decisions they've made in the past. Designers should also have a fluency in how the product is built, and if you're building anything technical they should have a basic understanding of coding practices.
- *Marketers* brand the product. They'll know a lot about the state of the product's brand identity, as well as information about your customers and their needs and desires.
- *Salespeople* pitch the product to customers. They'll know what drives purchases, why customers enjoy the product, and why they may gravitate towards competitors.
- *Support* follows up with customers after the product is purchased, solving any problems they may have. They will have a good sense of the most common problems that people face, as well as an understanding of any problems that your design may create.

Each of these roles provides their own set of challenges. The best organizations have people who overlap substantially in skill sets, but everyone requires compassionate and even-handed responses to their views. In the end, the best thing that you can have on your side is practice.

It helps to conduct stakeholder interviews with two people: one to lead the interview, and one to record a transcript.

Ask stakeholders what their role is, who the product is intended for, what constraints they're operating under, what they want to do for the project, and what the product should accomplish.

Many of the answers to these questions may be different from what you expect. Moreover, *everyone has opinions about design*. Your job is to weigh these opinions and draw informed conclusions from them.

Specific questions vary from role to role:

- When interviewing executives, ask about the timeline, as well as what trade-offs they're willing to make if the timeline can't be met.
- When interviewing product managers, ask about the product's demographics, what differences exist between user roles, and the product's desired features in the short- and long-terms.
- When interviewing engineers, ask what technical decisions have been made, what the product's underlying architecture is (if one already exists), and how the current product works in basic terms (if it exists).
- When interviewing quality assurance people, ask how they work with engineers to find and fix bugs, what significant problems they've found in the past, and how they'll test this particular product.
- When interviewing marketers, ask who they believe the company's customers are, and how that might change; how the product fits the company's business goals; what the product's brand identity or style guide looks like; and who the company's biggest competitors are.
- When interviewing salespeople, ask who they're going to pitch the product to, what differentiates your product over a competitor's, why people stop using the product, and what prospective customers ask about most frequently.
- When interviewing technical support, ask what the most frequent problems have been (if any), if they have any ideas for fixing those problems, and how they plan to triage future cases for this product.

4.1.9. Interview users.

With stakeholder interviews, you gather information to create a portrait of the company making the product. With **user interviews**, you seek to understand what people want to do with the product. Not all stakeholders are users, and good designers know how to fit the square peg of user needs into the round hole of a company's goals and capabilities.

Unless you're developing a product of considerable scope, you don't need to conduct more than a half-dozen user interviews. More than this will provide redundant or misleading information.

While this is probably far fewer in number than your stakeholder interviews, they're just as important. You're there to create a product that fills their needs. They're the final arbiters, the ones who tell you whether you did your job well.

RECRUITMENT.

Recruitment can be daunting, and you should always cast a wider net than you think you need. First, you have to determine the criteria by which people can be recruited; then you have to make sure you find the right people to fit those criteria, which can be very exacting and particular; then you have to make them show up in person, which presents logistic challenges. For this reason, many firms outsource recruitment to third parties.

Interviews shouldn't last more than an hour—and if you're recruiting from the general public, you should probably offer some sort of honorarium to compensate them for their time.

CONDUCTING THE INTERVIEW.

Focus on understanding what tasks each user wants to complete. You should ask what they do most frequently with the product, and what tasks are the most important to them. How do these tasks affect other people? Very few tasks are conducted in a vacuum.

If you're redesigning an existing product, ask about their current setup and figure out its benefits and shortcomings.

User interviews are opportunities to vent. Be patient and listen to what they have to say about the product's drawbacks.

We wouldn't have an interview in the first place if we knew what our customers wanted. Interview people as if they're the teachers and we're the students. Towards that end, it's extremely important to be non-judgmental and humble. If they use our products in unexpected ways, we should be sympathetic to that and encourage them to expand on their own needs.

When it comes to asking questions, our first impulse is to project our opinions. We want to agree with them; we want to clarify their points. But this colors their views, and it doesn't allow us to fully understand what they have to say. We have to suppress this impulse and encourage people to provide their own thoughts.

Questions should be open-ended, asking them what they are trying to do, why they feel the way they do, and what steps they think are required to perform the task. Questions should be deliberately naïve, inquiring into the greater context of their profession. That way, you don't make any assumptions about terms or processes that you may not know.

We should ask for more detail when the user describes anything that we don't fully understand; it's always better to ask for clarification than to assume you can fill gaps in later.

Finally, questions shouldn't ask people for specific solutions. Finding the right answer is *your* job.

4.1.10. Judge stakeholder and user opinions equally.

If there's a conflict between what stakeholders want and what customers want, then you should consider compromises that fulfill the desires of each equally, as if each group is represented by a separate person, and you're trying to make both as happy as possible.

4.1.11. Test your product at every stage of completion.

You can only ensure that your product works right if you test it with real people. No amount of research can replace the need for frequent, well-executed testing. Because you can test products at any level of functionality or fidelity, testing should start as soon as a prototype is put together.

Formal testing takes considerable money, effort, and time to conduct. In most cases, it takes so long that it diverts your focus away from making a great product. Hiring a full usability team, recruiting subjects from the general public, and buying fancy eye-tracking equipment are luxuries afforded by only the largest companies—and only if they really want to do usability testing, or if they have enough time to do it well. People are usually too busy for formal testing—but fortunately, **discount usability testing** offers little in compromise and considerable benefit in agility.

One of the fiercest advocates for economical testing methods is Jakob Nielsen. In the mid-nineties, he coined the terms *discount usability* and *guerilla HCI*. He was the first to run usability tests on sketched prototypes to isolate major problems as early as possible. He introduced these ideas twenty years ago, but they were poorly received at first.

Since then, discount testing has become respected in many organizations. It's affordable. It can be conducted faster. It offers immediate, actionable advice.

Only a tiny handful of interaction designers ever conduct professional usability testing. They might do it once in graduate school, and then dismiss it as a fluke of infinite resources, unbound by the constraints of time or budget. But you can't forgo testing entirely. Done economically, testing becomes feasible and tremendously helpful.

USABILITY TESTING ON A BUDGET.

When most people refer to **usability testing**, they're talking about giving people a series of tasks to accomplish with a product at any stage of completion, then timing their responses, gauging their reaction, and asking them to explain their impressions.

Kim Goodwin says that this “is the product designer's equivalent of asking a patient where it hurts.” Stakeholder and user interviews are discussed in much greater detail in chapters 5 through 8 of Goodwin's reference tome *Designing for the Digital Age* (Wiley).

You should only interview stakeholders *as* users if they're the *only* users.

Jakob Nielsen, “Discount Usability: 20 Years,” *Alertbox*, <http://www.useit.com/alertbox/discount-usability.html>.

Note that this is different from user interviews, which should take place *before* prototyping. But the principles that apply to interviews—not asking suggestive questions and listening carefully—still apply.

For example, if you're making an email client, you may ask someone to check, compose, and send an email. If you're writing a scheduling application, you may ask them to create some events and check for time conflicts.

Usability tests prove the effects of design decisions quantitatively by timing the user, counting a task's steps, or assessing the number of errors they make. If the time (or number of steps) decreases to complete a task, we say that the product's usability has increased by that much. Effective tests need a way to ensure consistency between test subjects, as well as a way to measure success.

Test your product early, when you can iterate it rapidly. Testing earlier beats testing later by such a magnitude that it makes no financial sense to forgo it. It helps refine your mission, and it allows you to incorporate feedback into your project immediately. You can come up with many designs, too; testing allows you to choose the best one and build upon it.

Most professional usability tests involve at least ten subjects, but you can test with 95% certainty using as few as three. You can go through a months-long recruitment process, or you can call on anyone with an intermediate experience level. You can test in a dedicated usability lab with a specialized observation room, or you can set up a computer in your office's conference room. You can write a 50-page deliverable that recommends a wholesale overhaul of a mostly completed product, or you can write a single page of suggestions for something that's still in the early stages of development. Economical alternatives to rigorous testing provide enough useful results that the significant added cost usually isn't worth it.

PAPER PROTOTYPES.

Section 2.6 discussed the value of sketches as a fast and inexpensive brainstorming tool. Readable, clean sketches can be used as slapdash prototypes to test significant usability problems before any code is written.

Paper prototypes help test hierarchy, layout, and rudimentary behavior. They're especially useful for products with more structure and static content, like small to mid-sized informational websites.

To evaluate paper prototypes, you need someone else to guide the test. The guide will swap out various prototypes as the participant proceeds through each step. It's often useful to mark out the monitor's area on the table with four strips of tape, to discourage anyone from moving the prototype around. Then it proceeds like any other usability test: you have them perform a specified series of tasks. Time how long it takes, record any noteworthy successes or errors, and count how many steps it took to accomplish the task.

If you think that applying usability testing to unrefined sketches is too reductive to be useful, it's worth remembering that the earlier you test for usability, the better the product's final state. Paper prototypes work near the beginning of the design process, revealing any major layout issues.

Steve Krug's *Don't Make Me Think* and *Rocket Surgery Made Easy* (New Riders Press) cover discount usability testing at length, including how to conduct a test and common issues you'll encounter. For more on how this applies to our whole field, see Cennydd Bowles & James Box, *Undercover User Experience Design* (New Riders).

Printing out graphical mockups or wireframes accomplishes the same thing, but takes more time.

Heuristics are metrics by which the simplicity, clarity, and utility of an interface can be evaluated. **Heuristic evaluation** is a kind of usability testing that determines how well a product fulfills a series of heuristics, offering suggestions for how to solve any issues.

As with conventional usability testing, you can use heuristic evaluation to judge design effectiveness at any level of fidelity, from sketch to fully working product. The main difference is that heuristic evaluation tends to be conducted with experts (or other designers) who are familiar with your product. Experts can evaluate an interface for smaller details, like performance hits or subtle rendering errors.

Between three and five people should evaluate the design, to collect multiple opinions and a comprehensive analysis of the product. But if constraints prevent your testing with three people, testing with one person is still better than testing with none.

Heuristic evaluation should be performed by those familiar with the craft, but this doesn't mean you need to spend thousands of dollars hiring a consultancy to do it. It's easy to train yourself and others, especially because there's a lot of flexibility in what constitutes a good heuristic evaluation. Some characteristics are common to all good evaluations, though:

- Each evaluator should evaluate the product using the same set of heuristics.
- Each evaluator should review the product alone.
- Each evaluator should review the product at least twice. The first review is for layout, context, and overall tone; the second is for specific elements or behaviors.
- Reports can be organized in two different ways. Evaluators can list each heuristic and describe what parts of the interface do (or don't) fulfill each. Or they can organize it by element, and describe what heuristics do (or don't) apply to each.
- Each evaluator should create a written report of their findings. Write a summary of these reports, along with any decisions that you've made as a result.

You can train many stakeholders to be evaluators if they know the product especially well, and it's easy to work with the people that you have.

Unless experts are your only target group—which is unlikely for the vast majority of products—heuristic evaluation should complement usability testing with real people. Fortunately, usability testing is usually easier, faster, and cheaper than heuristic evaluation, and it's more appropriate for checking products at every stage of development.

Testing with more than five people will provide a negligible increase in useful data. For the research supporting this claim, check out Jakob Nielsen, "How Many Test Users in a Usability Study?"

Alertbox, <http://www.nngroup.com/articles/how-many-test-users/>.

All of this works best when you have a rapid, iterative development process. You start with something rough, and then build on it quickly, in small increments. Along the way, evaluate your decisions through user testing and heuristic evaluation.

These are necessary steps in a sound development process. They push you to make something better, they open your mind to alternate solutions, and they allow you to second-guess your original plans and change course quickly.

In software development, the best practice for rapidly iterating is called the **agile development model**, which favors frequent iterations and upgrades on a daily, weekly, or fortnightly basis. Agile creates a workable product as quickly as possible. It also exposes and fixes flaws efficiently.

Embracing agile is a procedural and attitudinal shift away from the traditional **waterfall model** of conducting quality assurance and vetting large, feature-rich builds with many stakeholders.

4.1.12. *Understanding expectations is a never-ending process.*

Expectations age quickly. Figuring them out requires constant adaptation. Technology changes too quickly for our desires to stay still.

Research is ongoing, and success is a pendulum; we struggle as we gain and lose ground.

Find peace in this. Know that you'll never know everything. Seek joy in the attempt.

Rapid iterations hinge on starting small and building the product carefully—which, of course, plays into notions of simplicity. Starting small is discussed in detail in 5.3.

The agile development model has considerable support and widespread traction, but some good starting points include:

- Mike Beedle et al., “Twelve Principles of Agile Software,” *The Agile Manifesto*, <http://www.agilemanifesto.org/principles.html>.
- Agile Alliance, <http://www.agilealliance.com>.
- James Shore, *The Art of Agile Development* (O'Reilly Media), 2007.

4.2. Manage expectations.

Robert Pirsig, on a motorcycle repaired using an aluminum can:

*I was seeing what the shim **meant**. He was seeing what the shim **was**. That's how I arrived at that distinction. And when you see what the shim is, in this case, it's depressing. Who likes to think of a beautiful precision machine fixed with an old hunk of junk?*

Robert Pirsig, *Zen and the Art of Motorcycle Maintenance* (Bantam), p. 55.

4.2.1. Feedback should be helpful.

Providing feedback and instructions can help correct errors before they're made. Links can be added after form fields, directing people to inline help. A paragraph of copy with instructions can precede a form, or copy can be placed in a sidebar for someone to follow as they enter information. And while form validation is preferred, adding "Entries should be in the form of..." blurbs is better than nothing.

Products should provide subtle but firm encouragement that errors are easy to overcome and working with the product remains feasible. Good products understand people's shortcomings, addressing them before they pose problems. To customers this comes off as effortless, but to the designer it's simply a matter of being prepared.

4.2.2. Address common forms of error.

Nobody can design a perfect interface that accounts for every problem. When errors occur, the resulting feedback should be concise but descriptive, pointing people towards potential solutions to their problems without any technical jargon.

In *The Design of Everyday Things*, Don Norman talks about the different categories of errors that people can make, categorized as *slips* and *mistakes*. **Slips** occur subconsciously, when we unthinkingly proceed through otherwise routine tasks. **Mistakes** happen when you consciously misinterpret a product's basic function: you think it does one thing, but it does another instead. Norman credits mistakes to your "choice of inappropriate goals."

Don Norman, *The Design of Everyday Things* (Doubleday Business), pp. 106-7.

Norman discusses slips in much greater detail, but it's worth summarizing them:

- *Capture errors* are when you perform one task frequently, and then you perform a different task as if it were the frequent task. For example, one time I took the morning off work to go to a doctor's appointment. Rather than take the bus to the doctor's office, I woke up and took the train downtown to my workplace. The process of getting ready resembled what I do before work every day, and so I interpreted the morning's events as requiring that I go to work.

I am not a morning person.

- *Description errors* are when you apply an action to a process that appears similar, but isn't what you wanted. Description errors tend to happen when two objects are near each other, like every time you press an adjacent key on a keyboard, flip the wrong switch in a long row of switches, or right-click when you meant to left-click. The original example is helpful: "A person intended to put the lid on a sugar bowl, but instead put it on a coffee cup with the same size opening."
- *Data-driven errors* occur when people perceive some sort of information and intend to communicate something different in response, but instead recite the original information. For example, let's say I'm instant messaging a friend and they give me their new phone number. At the same time, I'm on the phone with the electric company, and they ask for my phone number. Instead, I provide my friend's.
- *Associative activation errors* are when you respond to similar stimuli identically. Another example from Norman: the phone rings, so you pick it up and say "Come in" as if the caller had knocked on your office door. Alternately, I've replied to emails in the past with "Thanks for calling!"
- *Loss-of-activation errors* are when you simply forget what it is that you meant to do. You walk into your bedroom and wonder why you did so. You spend five minutes scratching your head, searching your room for objects you may have meant to pick up, until—ah-ha! You meant to turn the lights off.
- *Mode errors* occur when you respond to a product that is in one mode as if it's in another, like entering a password when your caPS LOCK IS ACCIDENTALLY ENABLED.

Don Norman, *The Design of Everyday Things* (Doubleday Business), pp. 107-8.

For more information on modes—and why they cause more trouble than they're usually worth—see 6.5.

All of these errors can occur in your product, and your design should address the behaviors that might produce them. You also have to account for the possibility that someone will completely misinterpret how to use your product: that a *mistake* will take place. Making elements and layouts as unambiguous and simple as possible helps to compensate for that.

4.2.3. Encourage undo instead of confirmation.

Products need a way to account for error. **Undo** reverses the action of the most recently completed step, while **confirmation** provides a second step before completing an action.

Confirmation requires an extra step every time a task is performed, but undo requires an extra step only when an error is committed. When scaled to every step of a product, enabling and encouraging undo can dramatically speed up someone's interaction. Confirmation prompts can usually be replaced with undo functions.

Undo works especially well when you can quickly backtrack across multiple steps. Undo histories can be recorded, to preserve entire paths. Undo history should also be preserved between openings of a file, or instances of

an interaction, so the history doesn't erase when you close the file or quit the program.

4.2.4. *Streamline input.*

Don't create a long path when a shorter one will do.

REMOVING UNNECESSARY SCREENS.

Sometimes, entire layouts are excessive. Two pages can often be combined. Some pages can be removed entirely.

Many websites present long forms across multiple pages; for instance, e-commerce sites will prompt for shipping information on one page and billing information on another. Sometimes it's sensible and appropriate to differentiate between similar types of input—text fields of the same size next to one another, for example—because it reduces perceived complexity and the chance of error. But in time-sensitive situations, like purchasing tickets to a popular event, it can slow people down and interrupt their flow. Make sure you can justify the trade-offs in adding pages.

In this case, you could make the text fields a different size, or separated further on the page.

PROMINENT NAVIGATION.

Google once said they wanted people to spend as little time on their site as possible—because they could quickly find the information they wanted.

Navigation can speed up our interactions by being prominent and unambiguous. For example, ebook readers have prominent backward and forward buttons, to shift through pages easily. Newer word processors and spreadsheet applications have an all-in-one toolbar that controls most functions with a maximum of two steps. These are means of wayfinding, and the same principles of simplicity and clarity apply to them.

This claim peppers the Internet, but one mention is from Paul Nelson, "Interview: Esteban Walther, Google head of travel Europe," *Travel Weekly*, <http://www.travelweekly.co.uk/Articles/2007/02/08/23713/interview-esteban-walther-google-head-of-travel-europe-8-feb.html>.

CONTEXTUALLY APPROPRIATE NAVIGATION.

Just as navigation should be prominent, it should also adapt to where you are. It should mark your place, knowing where you've gone and how you arrived there.

If the navigation is a list of primary sections, then navigating to one should result in a list of that section's sub-pages. If needed, a **breadcrumb trail** shows linked titles of each page in the site's hierarchy, to show where they came from and how they can get back there. People sometimes prefer breadcrumb trails to the back button, especially when they enter a site through a third-party link. That said, back buttons can often include the previous step as a label, especially on smartphones.

Sub-navigation raises some interesting layout questions. Many layouts initially provide no room for sub-navigation. Adding it too prominently

could disrupt the layout, with a high risk of distraction. But hiding the sub-navigation until it's needed runs the risk of it never being discovered.

It's possible to include sub-navigation in every menu to begin with, but that doesn't scale to larger products, and it won't convey where someone has just navigated.

There are many potential solutions, but all of them pose trade-offs. Make sure you test navigation thoroughly before launching, because it's difficult to shift from one to another in a finished product.

CRUD INTERFACES.

CRUD stands for “create, read, update, delete,” which are the four tasks that are permitted whenever a product has to work with information. In short, *allow input wherever output exists*.

This is especially pertinent for administrative interfaces.

4.2.5. *Remember the user's place.*

Thoughtful products remember a person's actions; intelligent products learn a person's habits. When someone stops interacting with a product, it should save the most recent state. Reopening it should, by default, put them back at that state, rather than a blank slate.

Ignore this rule if someone prompts a program with some sort of input, like if they want to open a file. The program should open the file, but preserve the other saved state in the background.

4.2.6. *Never provide the ability to reset a form.*

A reset button never does anything that refreshing the page, an undo function, or manually clearing the form fields can't also do. Inserting this button—often next to the submit button, which does the exact opposite thing—increases the chance of a significant error.

4.2.7. *Controls should show the desired function when it's unclear that they toggle between multiple states. Otherwise, they should show the control's state.*

When a control toggles between multiple states, it should convey that selecting it would trigger the displayed behavior. This should only work for toggles that don't clearly convey the idea of “toggling,” such as buttons and links. Switches and switch-like controls should instead convey their current state.

For example, if you're expressing a control as a button, the button's text should read what you expect that button to do. A button that turns on a light bulb should read “on” when the light is off, and “off” when the light bulb is on.

But you typically don't turn light bulbs on with a button; you use a switch. The switch conveys the bulb's current state; if the bulb is off, the switch reads "off," and vice-versa.

As a result, most two-state toggles should be represented as switch-like controls. Be careful when using controls that conceal possible options.

4.2.8. Work should be automatically saved, and saved work should be easily recalled.

Automatic saving preserves work and saves time. The product can periodically save time-stamped backups as someone works, while providing minimal, unobtrusive feedback each time one is saved. Scaled to longer periods of time, this can replace the need to save manually, with no intrusion and relatively little disk capacity.

There should also be some way to recall backups, and to merge existing work with old drafts. This could be integrated into the FILE menu of many editors as a RECOVER command, showing the list of time-stamped versions and offering the opportunity to open them as if they were separate documents.

Automatic save and recovery should be turned on by default, unless the average document's file size is too large or disk space is prohibitive. If automatic save has been implemented, manual saving should preserve an instance of the original file.

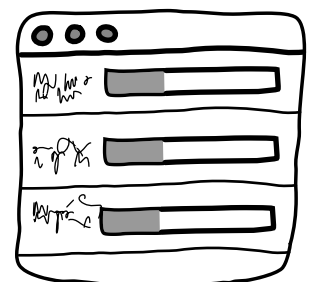
4.2.9. Monitor progress to fit the person, not the process.

Displayed progress is a natural mapping between an interaction and its feedback. While there are many different ways to accomplish this, progress bars are the most common. For computational tasks, progress bars should be a function of time taken. They can be weighed based on the results of previous tasks, which might be more accurate.

When progress doesn't naturally map to time taken, the interface toys with your expectations. Progress bars that jerk erratically towards their finish teach you nothing about the process. It's endlessly frustrating when a progress bar hangs on some massive procedure at 99% for ten minutes. If you can't display an accurate progress bar, show different information, such as percent completed or specific tasks completed, or omit it altogether.

SIMULTANEOUS PROGRESS BARS.

Many tasks run simultaneously. Progress bars should be laid out such that concurrent tasks can convey feedback simultaneously. As a result, layouts should afford the addition of more than one progress bar.



HIGHLIGHTING COMPLETED TASKS.

Showing a flat list of ongoing tasks is another way to imply progress on computations that have many discrete parts, especially in software installation or account creation. Then as tasks are completed, they can be highlighted to indicate progress in some way.

For example, as the computer works on a task, it can fade each list item from black text to some other color. As it *completes* each task, the type can switch from normal to bold. This offers a comforting, if imprecise, way to show progress.

ESTIMATES IN COPY.

Copy can suggest estimates based on your internet connection's speed, or your computer's processing power. This helps you estimate the duration of a task, allowing you to take further action if needed.

Done

Done

In Progress

Pending

Pending

Pending

18% (about 3:42 remaining)



4.3. Elevate expectations.

Author and actor Stephen Fry:

Don't you sometimes long to be CEO of a company like Sony Ericsson, Samsung, Nokia or Microsoft? So that you can say to your coders, your designers, your development teams and your software architects: "Not fucking good enough. I haven't said 'Wow' yet. I haven't gasped with pleasure, amusement or admiration once. Start again. Not fucking good enough."

Stephen Fry, "Gee, One Bold Storm coming up...", *The New Adventures of Stephen Fry*, <http://www.stephenfry.com/2008/12/11/gee-one-bold-storm-coming-up/>.

4.3.1. Good interfaces bring technological progress.

Technological progress reframes everyone's expectations, setting a product as the new standard. Progress makes the old seem uncool, boring, and frustrating. Progress is a display of courage.

When most companies make a successful product, their first impulse is to fiercely protect it, avoiding any substantial changes. But this impulse is governed by fear. It's greedy and unsustainable. Standing still in the world of technology takes less effort, but can be a barrier to success.

It won't be very long before someone else with more confidence than you makes the next great thing. Even when it looks like you're dominating the market, it can't hurt to keep improving.

That said, progress could seemingly come out of nowhere. Disruptive businesses are often initially perceived as shabby or déclassé by incumbents. For more, read Clayton Christensen, *The Innovator's Dilemma* (HarperBusiness).

4.3.2. Create free interactions.

Sometimes it makes sense to add features that have no functional consequence, but nonetheless cause something perceptible to happen. They can subtly indicate a limit—like if a person selects something that's been disabled, or if they reach the end of a list. Free interactions always stay out of the way

One solid perspective on how to change existing products comes from Marco Arment, "Side effects of developing for yourself," *Marco.org*, <http://www.marco.org/392848093>.

of the primary task; it could be entirely possible that you never encounter one during daily use.

No free interaction is essential, but they're easy to include and they add tremendous value to a product's personality. They're also good ways to convey common errors.

4.3.3. *Provide delight and wonder.*

Your design should provide a clear vision for the way that a product should behave, giving it a personality all its own. Behavior becomes more than how a product responds to a stimulus: it allows us to predict other ways that something can work, without yet knowing how.

Good experiences make life memorable and beautiful. They provide a reason to go to work every day. They help us adopt constructive, enjoyable routines. And it's your job to create them.

This section is inspired by Chris Noessel, "One free interaction," *Cooper Journal*, http://www.cooper.com/journal/2009/01/one_free_interaction.html—which, in turn, is inspired by the "snapback" scrolling feature on iPhones.

Cadence:

5.1. Reorganize complexity.

Simplicity resists catering to edge cases, but professional software is cluttered with small, one-off quirks that cater to small minorities of people. Taken to extremes, advocating for simplicity casts judgment on people's habits, asking them to embrace software that doesn't do quite what they want.

When faced with necessary complexity, designers tend to begrudge it or embrace the challenge that it poses. The latter approach is more optimistic: it provides for consensus, and it frames complexity as something to work *with*, not *against*. Complexity should still be useful and in line with the product's core functions.

Some products can't be simplified any further without adversely affecting a significant number of people, so we need a way of dealing with complexity that respects its occasional necessity. In the long run, complexity is much more difficult to maintain than simplicity, and it should only be implemented with great care.

We gain a sense of accomplishment by solving many small tasks; conversely, we're often daunted when we face one enormous task. Finding ways to reorganize complexity can meaningfully frame common problems, teaching us to solve them in the process.

INCREMENTAL DISCLOSURE.

The most basic way to deal with complexity is by slowly revealing it across a series of steps.

On the web, using multiple pages for a form is a good method. When you have to collect a lot of essential information from new customers, it helps to segment form fields across multiple pages. As an added benefit, progress can be saved and tracked more easily.

The initial screens of many products disclose essential features. Subsequent screens incrementally reveal more information about the company or the product's function.

Gradual engagement techniques, like not requesting an email address or password for up-front account creation, manage this effectively while getting people excited about your product. Gradual engagement also provides a better first impression, showing the product's real-world use instead of a sign-up form.

Drop-down menus, tabbed navigation, and accordion forms also address the problem of complexity by hiding many additional functions until people need them. Accordion forms are particularly helpful, with each field set staying compressed until you need to fill it out.

And in the real world, we all know that the long lines for roller coasters are kept far out of public view and segmented so that they look like a series of short lines. Managing expectations this way keeps ridership high, and people are less frustrated by the wait.

5:

Perceived Accomplishment

GROUPING.

John Maeda summarized one way to deal with complexity on his personal blog:

A complex system of many functions can be simplified by carefully grouping related functions.

With drop-down menus, extra functions are not only hidden, but grouped with related functions. After enough time, some conventions become widely accepted: for example, what's under the FILE menu in one application usually belongs under FILE in another, as the configuration of that menu has persisted for many years.

In what other ways are related functions grouped? Can patterns be used to create new groupings?

MODULARITY.

Modularity is the division of a large system into many smaller, repeatable systems. Computer hardware is a classic example. The memory, processor, hard drives, or graphics card in many desktop computers can be replaced with different components.

With software, third-party plugin technology extends the functionality of many applications, including photo editors, web browsers, and text editors. On smartphones, applications are each represented by a single icon on a consistent, gridded screen. And on some operating systems, movable “widgets” tell us the time and weather conditions.

Complicated forms can be filled out in a modular way. For example, filing taxes online doesn't require filling out every single form the government provides—only the ones that are essential to *your* needs.

MOBILE FIRST.

Consider designing your application for smartphones before tablets or desktop computers. Developing for smaller screens puts greater focus on core features, allowing you to focus on what really makes your product great. Additionally, mobile platforms are rapidly increasing in market share and audience size, making them increasingly more sensible to target.

Overall, though, paring down the number of elements on a screen will help you effectively manage complexity.

5.2. *Manage the user's locus of attention.*

Listening to music, one pays attention to the aspects of a song that are in the *perceived foreground*. A melody may stand out at one point, for instance, but after time a drum flourish could take its place. Skilled musicians know where their listeners' attention lies, how and when it changes, and how to manage it.

John Maeda, “A... Ah ... Atchoo! Gestaltung!,” *Maeda's SIMPLICITY*, <http://web.archive.org/web/20080613184845/http://weblogs.media.mit.edu/SIMPLICITY/archives/000113.html>.

Grouping is also discussed in 3.2.



For more on developing for mobile devices first, read Luke Wroblewski, *Mobile First* (A Book Apart).

The same is true with interfaces. People shift their attention from element to element as they interact with a product. The **locus of attention** is the point where someone is focusing at a given time. It can follow many paths, each with undefined outcomes. Potential paths interfere with one another, making it hard for someone to evaluate each one, yet still conditioning their mental model. Attention is focused perception—and perception and cognition are the basis of every interaction.

Each step of an interaction presents two challenges:

1. Identify where a person's locus of attention is.
2. Determine where and how to shift it in the next step.

Every visible element affects your attention. You can mute certain elements and highlight others to establish a foreground relationship. You can alter the layout of elements to imply a hierarchy. You can change the pliancy of elements to show which paths are feasible. Natural mappings can link labels to functions. Monotonous interfaces help confidently direct your attention. Modeless feedback directs your attention without distracting you. Eliminating modes decreases the variance of paths, simplifying the interface and reducing usability problems.

In any interface, paths increase exponentially with the number of unique ways to act on a given step. This correlates with the number of different places that someone's attention can be directed.

Each pliant element that you add to an interface introduces a new potential path. From that path often comes a new layout, each with its own set of elements and corresponding set of paths. Complexity is capable of creeping into a product very quickly, and it can pose many daunting design problems if it's left unchecked.

CONTEXT.

Context affects your perception and attention significantly, and products should always fit the context they are placed in.

If you're making a mobile app, for example, you should expect people to use it in all sorts of different lighting conditions, from near-total darkness to blinding sunlight. If you're building a desktop computer, you may design it to be less shock-proof or visible in direct sunlight than a laptop. If you're developing a mobile website for a large outdoor event, it should load quickly, render clearly on a variety of devices, and be able to handle a traffic spike during the event itself.

POSTURE.

Posture, defined by Alan Cooper in *About Face*, describes how much a given product occupies its platform's layout. For example, some applications run best when maximized to the full screen. Others run best as tiny icons in the

For instance, your locus of attention is over here right now, when it should really be on the main text.

A *mental model* is a diagram of common user behaviors, often gathered through ethnographic research or contextual inquiry. For more on mental models, see Indi Young's *Mental Models* (Rosenfeld Media).

Pliancy and monotony are separately defined and elaborated upon in 2.2. Modes are described further in chapter 6.

For more on how path variance increases exponentially, see 2.2.

For more on how platforms and systems determine a product's context, see 2.8.

Alan Cooper, *About Face* (Wiley), p. 127.

menu bar, taking up almost no space. In between are windows of a variety of sizes, pliancy, and modality.

Currently, the most popular music-playing software usually adopts a full-screen (or *sovereign*) posture, attempting to display a considerable amount of information about a list of songs and all the associated ways to manage them, in addition to unrelated functions like smartphone applications, ebook management, and video playback. The designers address this with system-wide keyboard shortcuts for playback, as well as a special “mini player” mode that can better fit on a desktop.

Smartphone and tablet applications also adopt full-screen postures, so it should be clear what application you’re in at any given time. If it’s unclear, someone may misinterpret your application’s state as a mode error.

Different postures affect your attention in different ways. If you’re developing software, what posture does it have? What impressions does its posture make?

FEEDBACK.

Placement of feedback can drastically affect someone’s flow. Some messages can render the rest of the interface inoperable until they’re addressed. Others can stay to the side, away from your locus of attention.

Carefully consider how feedback directs someone’s attention. Take special note of instances where feedback occurs unexpectedly, such as an error.

5.3. *Start small and iterate slowly.*

START SMALL.

It’s a romanticized notion, but many businesses really do begin out of someone’s bedroom or garage. Small business owners can do quite a bit with less money and resources—and these constraints give them a greater incentive to start making money.

In the early eighties, a band called the Minutemen revolutionized punk rock under the idea that they *jam econo*: use few tools, tour on a stringent budget, and do as much by themselves as they can. They wrote an influential double album—45 songs in 81 minutes—showing that skill and talent matter more than tools or means.

Don’t overshoot your abilities at the beginning. You start from an idea. Your tools affect, but don’t necessarily limit, what you can make.

ITERATE SLOWLY.

Andrew Burroughs, on iteration:

When things are designed, the most intuitive path or solution is usually the one that is chosen first. Over time, improvements are made... as a result,

6.3 defines and elaborates on flow. 2.3 discusses feedback at greater length.

Chapter 1 discusses ground-up work on your product, and scaling slowly and carefully, at much greater length.

The Minutemen, *Double Nickels on the Dime* (SST).

Andrew Burroughs and IDEO, *Everyday Engineering* (Chronicle Books), p. 41.

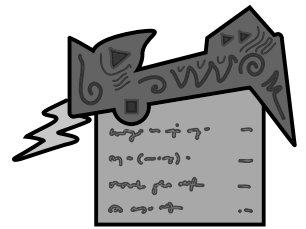
the design of an object often evolves away from that first intuitive idea. This results in interesting contrasts, especially when you see the “before” concepts and those that came afterward, side by side.

Build new things out of what you already have. Evaluate the context of new features before implementing them. Avoid knee-jerk decisions when implementing anything new. Deliberately make yourself wait too long before you begin work on something new. If you’re building with design patterns, focus on implementing only one pattern at a time.

Don’t build new features and fix bugs at the same time. Work should alternate between the two whenever possible, as they require very different procedures and mindsets. Building new features requires upfront planning, user testing, and careful vetting of scope; only the most pervasive and serious bugs require any of these activities. Only do both at the same time when you have to fix critical bugs.

5.4. Build essential functions first.

If you’re making a music player, it stands to reason that you should first build something that plays music well. Building the essential bit of a product provides its center, like the living room of a house, or a town square. With the center in place, you can then create supporting functions: the rooms around the living room, the buildings around the town square, the buttons that control the player.



5.5. Know what your most fundamental tasks are.

Word processors are good for writing documents. Music players are good for playing music. Email clients are good for reading and writing email. CD burners are good for burning CDs.

Imagine your product has only one function. What would that function be? The answer affects your company’s mission and your product’s goals, and it helps your team build consensus and maintain focus.

5.6. Don’t mandate a specific format for information.

Information can take different forms in different areas of the world. Fortunately, it’s easy to account for most variations. Products should be smart enough to anticipate the possible variations in input, for *all* relevant input.

For example, a United States phone number entered into a form can take many different formats: 312-555-1212, 312.555.1212, 1-312-555-1212, +1 312 555 1212, (312) 555-1212, 3125551212, etc. All of these are valid phone numbers. Your product should accept them, rather than enforcing a single format.

Creative solutions exist for standardizing input. Entry can be validated and automatically completed on the fly, as a part of natural feedback. For example, if you’re entering a telephone number into a form, your form can supply the bare essentials of a phone number, () – , with the cursor

after the opening parenthesis. As someone types *only numbers* into the form, it skips past the appropriate punctuation.

Entry that's validated on the fly is called **active validation**—as opposed to **passive validation**, which checks a field once someone focuses away from it. In either situation, validation removes much of the risk of error, as well as the need for server-side validation functions.

There are many different types of validation, from postal codes (some countries include letters in their postal codes; others do not) to states and provinces (e.g. Illinois v. IL). No matter what, though, you should trust that people know the right way to enter information.

5.7. *Require only essential information.*

At least one field is required in every form, or you wouldn't have people fill it out in the first place. But which fields should be required?

Form fields should only be required if they're essential to ensure passage from one step to the next. For example, one's email address should be all that's required to start an account on many websites, as passwords can be created over email later.

Only ask for mailing addresses if you have to ship something. Only ask for phone numbers if you have to call someone. Only ask for additional comments if they're essential to the form's purpose.

If forms contain a mix of required and optional fields, required fields' labels should be suffixed with a *** (e.g. PASSWORD ***).

Finally, if someone hasn't filled out the required fields, any feedback that information is missing should be carefully worded, sympathetic, and unobtrusive, and it should direct them to the right place to correct their information. Ultimately it's *your* responsibility to ensure that people complete forms as you want them completed; it's bad business to blame someone else for your form's shortcomings.

5.8. *Sort features into categories.*

In 2.6, I discuss repurposable design patterns that can organize a layout's elements by their purpose and function. Patterns apply to features as well. Features can be organized into sets of categories, which then determine layout and navigation:

1. **First**, *list all the features that you want to include*, how they will be visually expressed, what design patterns could affect them, and the way that they are supposed to behave.
2. **Second**, *consider which features make the most sense together*, and group them accordingly.
3. **Third**, *prioritize each group* based on its relative importance to the product. You should designate only three priorities here: low, medium, and high.

4. **Fourth**, *name each group*. If you can't come up with a name for the group, reconsider the grouping. Have you included too many things in it?
5. **Fifth**, *sketch each group*. Do this with whatever tools you'd like, but keep the process fast.
6. **Last**, *combine each of these sketches into a rough design*. Move the sketches around so the layout makes hierarchical sense.

Sketching features out is a good way to determine whether they've become too complex to safely roll out to intermediate users. Once you have a sense for how many features will suffice, then you can go through the more rigorous process of developing a pattern language in 2.6 and figure out how specific elements work together.

5.9. Use preferences where appropriate.

Preferences are modal controls that govern more than one potential interface behavior. Complex products often incorporate preferences, so people can customize their experience to best suit their needs.

People use products differently, and often the split between groups of common uses is more like 50/50 than 90/10. For these situations, if it wouldn't make sense to build two separate products, preferences are a suitable answer.

Many platforms have consistent conventions for where preferences are found. Since preferences are modal, they should go in the most familiar place that your context dictates, which is usually out of the way of the rest of your product's functions.

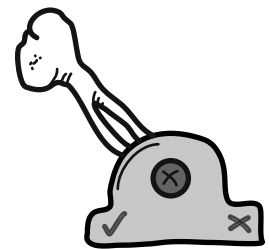
Be less conservative with preferences for products that exclusively target experts, as these users are accustomed to tailoring products to their own highly specific needs.

Preferences should roughly scale with the size of the product, and they should affect a subset of the product's features—not all of them.

5.9.1. Take great care in determining defaults.

Most people won't bother to reconfigure their products, and most novices won't even know that preferences exist at all. Default preferences are a product's most common settings, making them the most important.

For a long time, the most frequently visited website on the entire Internet was the default home page in Internet Explorer. Most passwords are among a list of only twenty common ones, like *password* or *123456*. If the majority of your customers will use your product in its default state, you shouldn't make the product dependent on preference customization. Make sure the product's defaults are the product done right.



For more on bad password practices, see also Sean Gallagher, "Born to be breached: the worst passwords are still the most common," *Ars Technica*, <http://arstechnica.com/information-technology/2012/11/born-to-be-breached-the-worst-passwords-are-still-the-most-common/>.

5.9.2. *Preferences shouldn't interfere with use.*

No product's operation should be contingent on its preference settings. If people don't ever want to customize their product, they shouldn't have to. It should work fine out of the box.

5.9.3. *Preferences are features.*

Preferences are subject to the same rules of simplicity as the rest of your product. Don't use them as an excuse for adding unnecessary features or settling internal debates.

6.1. Products' cognitive cost should disappear after frequent use.

Above all else, products should get out of a person's way. The best products become instinctual after enough use; response to any product should become reflexive as soon as possible. People already work towards this ideal, but they should succeed *because* of our designs, not *despite* them.

When using the web, people scan content as quickly as possible, remembering very little of any page. We put a lot of stock in our first impressions, and we are rarely charitable when we experience poor performance or crashing behavior in any product.

We expend very little conscious effort on using technology, because we consider it a means to an end. Most of the time, we've already established expectations about how a product works. Then we try to get things done. We should always spend more time on accomplishing the task than learning the associated technology.

6.1.1. Common functions should be self-evident.

Your product's most essential features should be its most obvious. Your product's utility should be evident to people of every experience level. Steve Krug said it best:

Making pages self-evident is like having good lighting in a store: it just makes everything seem better.

THE INVERTED PYRAMID RULE.

Section 5.1 discusses ways to refactor complexity so that interfaces appear simpler. So which functions should you include on the starting screen?

The **inverted pyramid rule**, commonly used in journalism, encourages writers to organize information in an article from most to least general and most to least important. Consider applying the inverted pyramid rule to your interface design: make general tasks the most accessible, and specialized tasks the least.

For instance, an article from the *New York Times* begins:

Barack Hussein Obama was elected the 44th president of the United States on Tuesday, sweeping away the last racial barrier in American politics with ease as the country chose him as its first black chief executive.

If you read *nothing else* in the article, you still know what happened. Someone was elected President of the United States on a recent Tuesday, his name is Barack Obama, he is the country's first black president, and his victory was by a wide margin. This paragraph is rich in detail, but its main points are easy for anyone to understand.

If the reader is still interested, additional details are disclosed in descending order of importance. The article continues:

6:

Cognitive Cost

Douglas Hofstadter, *Gödel, Escher, Bach* (Basic Books), p. 26:

No one knows where the borderline between non-intelligent behavior and intelligent behavior lies.

This point is corroborated by Aza Raskin, "Good Interfaces Create Good Habits," <http://www.azarask.in/blog/post/good-interfaces-create-good-habits/>.

Steve Krug, *Don't Make Me Think!* (New Riders Press), p. 19.

Adam Nagourney, "Obama Elected President as Racial Barrier Falls," *The New York Times*, <http://www.nytimes.com/2008/11/05/us/politics/05select.html>.

Mr. Obama, 47, a first-term senator from Illinois, defeated Senator John McCain of Arizona, 72, a former prisoner of war who was making his second bid for the presidency.

This lists the ages, careers, and locations of both the President-elect and his opponent. And then:

Not only did Mr. Obama capture the presidency, but he led his party to sharp gains in Congress. This puts Democrats in control of the House, the Senate and the White House for the first time since 1995, when Bill Clinton was in office.

This places the victory in context and provides historical background.

From paragraph to paragraph, the details progress from most to least important, and least to most specific. You can read only the first paragraph and get the gist, but you also have the opportunity to learn more if needed.

Likewise with interfaces: essential tasks should be the most obvious. And the more specialized the task, the more hidden it should be, and the more steps should be required to reveal it.

AFFORDANCES.

We perceive our environment “in terms of its possibilities of action.” We ask what the world has for us. When a possibility is evident, we can act on it more easily.

Section 4.1.1 mentions that people perceive technologies as tools for accomplishing various tasks. The same may be said of affordances, which are only as good as our ability to take advantage of them. Affordances can solve many design problems.

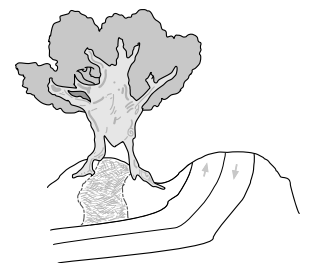
For example:

- The shape of a door handle is a good example of an affordance; if designed well, it conveys whether to push or pull.
- A bridge’s supports position the road bed and keep it stable.
- A manhole’s shape and weight keep it in place, but its perforations allow water to drain through.
- The gears of a bicycle’s drivetrain work with the chain and pedals to create forward momentum.
- The woven structure of a chain link fence conveys a sense of impermeability at a very low materials cost.
- The desire paths worn in grass fields from continuous foot traffic take advantage of the most common way to get between two points, indicating an opportunity to build a sidewalk.

If a feature is going to afford its own use, it has to imply *existence* and *behavior*. It should be in your locus of attention. It has to imply how to use it, and how it will behave once used.

This quote is from Jane Fulton Suri and IDEO, *Thoughtless Acts?* (Chronicle), p. 170.

Affordances were first defined by James J. Gibson, *The Ecological Approach to Visual Perception* (Psychology Press). 2.9 provides more examples of affordances as they apply to human factors.



The task of implying so much can be less daunting than it initially appears. In *Designing Design*, Kenya Hara alludes to a narrow rectangular groove cut in the floor along a house's foyer wall. Visitors rested their umbrellas inside the groove, leaning them against the wall. The groove was not a conventional umbrella rack, but it afforded the exact same functions that a conventional rack would. As a rack, it was functionally complete because people used it unthinkingly and effortlessly—although it might not work quite as well for folding umbrellas.

Fitts' Law helps define the affordances of various on-screen elements. It posits that the time required to move to a given target on a computer screen is a function of the distance to that target. In modern interfaces, however, the edges and corners of a screen afford an *infinite-sized target*—because no matter how far your cursor moves, it remains at that point on the edge. Thus, when you're using a mouse or trackpad, a screen's corners are the most important real estate on any graphical display that's operated by a mouse, and the edges are the second most important. For this reason, many crucial functions are placed in these areas. For smartphones and tablets, your hands are the "pointing device," so you need to consider their orientation relative to the screen when considering where to place important elements.

Fitts' Law is also a function of the size of your target: short, wide buttons are harder to hit if your cursor starts from a point directly above or below that button. While the button's area may be large, the distance required to reach it is too short to move to reliably.

There are other ways to create affordances. Emphasize elements' pliancy, increase their size or change their color in order to draw attention to them. Elements' position also matters: consider moving them towards the center (in the way), the edges (to take advantage of Fitts' Law), or the left side (on the web, to take advantage of the typical "F-shaped" reading pattern).

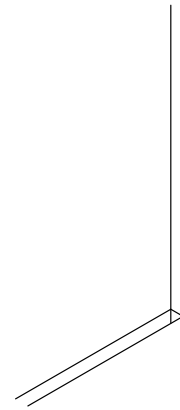
In what other ways can you design a product's critical functions so they're more noticeable and easier to use?

6.1.2. Paths should be self-evident.

A product operates according to a set of rules, and those rules can be obvious or obtuse. When you open a product for the first time, the finite number of things you can do with it provides a framework for further use; for each of these first steps, there's a set of second steps that you can take.

As alluded to in 2.2, paths branch out with exponential complexity per step. The entire set of paths and steps that someone could potentially take is called a **decision tree**. Decision trees can be *wide* or *narrow*, meaning they can have many or few unique potential paths. They are also either *deep* or *shallow*—meaning once you start a path, it either takes many or few steps to complete a task.

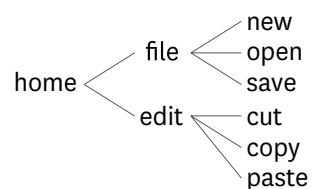
The best decision trees are narrow and shallow. They're easier to implement and simpler for people to think through. Ostensibly "simple" decision trees are still surprisingly large, though. For example, tic-tac-toe has



Kenya Hara, *Designing Design* (Lars Müller Publishers), p. 44.

A great summary of Fitts' Law is available at Kevin Hale, "Visualizing Fitts's Law," *Particletree*, <http://particletree.com/features/visualizing-fittss-law/>.

Jakob Nielsen, "F-Shaped Pattern for Reading Web Content," *Alertbox*, http://www.useit.com/alertbox/reading_pattern.html.



The inspiration for this comes from Don Norman, *The Design of Everyday Things* (Doubleday Business), p. 119.

an extremely simple set of rules, learnable in only a few minutes. The decision tree for a tic-tac-toe game, expressing all unique types of moves by X and O, depicts eleven separate paths with a maximum of seven moves per path. When considering *permutations* of tic-tac-toe games, though, which treats every square as if it's unique, there are over *fifteen thousand* different potential moves. But that's a drop in the ocean compared to a chess board: its decision tree comprises approximately 10^{120} unique games. To put that in perspective, the observable universe contains approximately 1.5×10^{82} atoms.

There are only three kinds of first moves that you can make in tic-tac-toe: a corner, a side, or the center. Now think about all the ways that people can trigger behavior in the first screen of your product. Is it obvious where they should go first? Is your decision tree so wide that people may be confused about what to do next? Is it so deep that you'll lose them midway through the process?

Apply this thought process to subsequent steps. Each step possesses its own decision tree. Is the next step contingent on feedback from previous steps? What are the customer's expectations?

Most people can step back from the present task and suss out patterns in anything that looks like it could be systematic. It's how we're able to figure out something as complicated as a chess game. It's also what we mean, more or less, when we say we're *figuring things out*. The most successful products minimize this process, as they're easy to figure out and use unthinkingly.

6.1.3. *An interaction should proceed as fast as our mental ability to process it.*

Moore's Law states that processor speeds double every two years. But the response time of computers hasn't increased accordingly, because we keep writing software that takes advantage of the extra processing power. Only recently have we begun to optimize what software we currently have, rather than incorporating more computationally demanding features.

The more demanding we make our software, the harder it is for that software to fit the speed of our perception. Customers pay dearly for this: the cognitive cost of slower products affects their ability to work with the entire system.

People shouldn't feel limited by their technology, and they process information very quickly. The period between any two steps should be of negligible length. Interruptions slow us down considerably, and a product's response time forms our expectations for its future performance.

But if we write yesterday's software to work on today's hardware, it runs much faster. And if we were to stop developing new features from time to time, we could focus on making our products faster and more reliable. Maybe if we build our products so they don't hog system resources, they won't collapse under their own weight, either.

John Carl Villaneuva,
"Atoms in the Universe,"
Universe Today, [http://
www.universetoday.
com/36302/atoms-
in-the-universe/](http://www.universetoday.com/36302/atoms-in-the-universe/).

Gordon Moore,
"Cramming More
Components onto
Integrated Circuits,"
Electronics 38(8):
114-17, 1965.

See also R.B. Miller,
"Response time in man-
computer conversational
transactions," *Proc.
AFIPS Fall Joint
Computer Conference*
33: 267-77, 1968.

All of this plays into
optimizing software for
better performance.
1.6 elaborates on this
at greater length.

6.1.4. Support the user's muscle memory.

Muscle memory uses repetition to encode certain physical tasks into memory, eventually allowing people to do things like ride a bike or play an instrument without conscious effort. We develop these skills over time, and eventually we stop thinking about them.

It's very hard to break a habit that's been committed to muscle memory. When an inefficient way to complete a task is committed to muscle memory, it requires considerable effort to undo the damage before anything new can be learned.

Our motor skills come in two categories: *fine* and *gross*. Fine motor skills involve small, precise actions, like typing, brushing teeth, writing, or eating. Gross motor skills require more substantial body movement: running, throwing, kicking, or biking.

Using small devices like a keyboard or mouse requires significant repeat use of fine motor skills. We have to understand fine motor skills if we're going to understand how to put muscle memory to work in our products.

Different aspects of fine motor memory are utilized in varying degrees for a given task. For example, if you haven't written for a long time, picking up a pen and trying to write for a long time often results in cramps and poor handwriting. The same goes for typing or using a mouse; it stands to reason that experts have fewer problems using a keyboard and mouse than novices.

Keyboards, mice, and touchscreens all require the use of different groups of muscles. All of these input methods are widespread now, and practiced frequently. To confound the problem further, switching rapidly between two different input methods (keyboard and mouse, for instance) involves largely separate groups of muscles.

Considerable research favors restricting the amount of input methods to only one. In a desktop computer's case, this is the keyboard, since it involves less range of muscular movement. Keyboard-only interfaces are used primarily by experts, though, since they usually require memorizing an extensive series of product-exclusive shortcuts to perform basic tasks. This means that touch-based devices benefit from having only one interaction model for people to learn.

REPETITIVE STRESS INJURIES.

Repetitive stress becomes an issue when anyone uses a product for an extended period of time. Practicing good ergonomics is the best solution, and that's your customer's responsibility. But we have responsibilities of our own, and we can frequently meet people partway.

By offering keyboard shortcuts, shorter paths, and encouraging simpler repetitions that require less strain, we can reduce the risk of repetitive stress injuries. We can build narrower keyboards, more comfortable key layouts, touchscreens, and interaction models that require only one input method.



A personal anecdote: I type with every finger on my left hand, but on my right hand, I type entirely with my pointer finger. That's how I learned how to type, and I doubt I'll ever be able to change it.

Gregory Shaw and Alan Hedge, "The Effect of Keyboard and Mouse Placement on Shoulder Muscle Activity and Wrist Posture," *Cornell University Ergonomics*, <http://ergo.human.cornell.edu/AHProjects/Mouse/keyboard.html>.

For a synopsis of input methods, take a look at John Gruber, "Where Keyboard Shortcuts Win," *Daring Fireball*, http://daringfireball.net/2008/01/where_keyboard_shortcuts_win.

People of all experience levels can suffer from repetitive stress problems, but—for obvious reasons—experts are at greatest risk.

For more on repetitive stress injuries, consult OSHA's reference materials at http://www.osha.gov/SLTC/etools/computerworkstations/adtnl_matrls.html.

6.2. Manage the user's workflow.

Workflow models are a way to describe steps and paths. Usually expressed as a flow chart or numbered outline of each step of many paths, they're useful for auditing paths to determine where you can safely eliminate steps. They should describe a task's intended goal, note the relative frequency of each path, and call out any actions that are dependent on other actions. Decision trees can also be outlined with a workflow model.

Workflow models can be tied to usability testing by indicating where someone's locus of attention should be at each step. This provides a good starting point for asking people what they're thinking, and seeing if it matches your own expectations.

6.3. Create a rhythm of rapid, brief, and repetitive interactions.

Flow is the name of a psychological state, coined by Mihály Csíkszentmihályi in an influential book of the same name, that refers to single-minded, deeply focused immersion that you can attain when performing a given task. Csíkszentmihályi cites examples of musicians getting "in the groove," as well as athletes competing after considerable practice. Game developers have attempted to leverage principles of flow in order to make more engrossing games.

Encouraging flow is a huge aspiration among interaction designers. Chances are you've experienced this state on a project where, uninterrupted, you were deeply involved in solving a complex problem. It takes time and effort to experience flow, but it is extremely fulfilling. The most successful designs appear to provoke it effortlessly.

Once someone begins to work instinctively with any product, they become fluent in its slang: their intent is connected with the product's output. Conversely, people won't experience flow if your product is too difficult to use—and they'll think more highly of products that they can learn easily.

One of the best ways to encourage flow is by providing a way to repeat similar tasks efficiently, leading someone to adopt a **rhythm**. A single instance of a rhythm may appear inconsequential, but when repeated many times, a larger purpose reveals itself. Andrew Burroughs acknowledges this:

A process's singular actions often appear to have no effect, but when they are repeated over and over again, there is a noticeable cumulative result. Such traces can reveal trends or interesting phenomena that otherwise might not be visible.

No matter what form it takes, rhythm is fragile: it's much easier to stop a rhythm than it is to create one. But we have a tendency to find regularity in any interface, if it's there for us to see.

With or without rhythm, though, flow happens out of habit. Once we work with a product for long enough, its use is imprinted in our memory, framing how we handle future interactions. What matters is how much

Mihály Csíkszentmihályi, *Flow: The Psychology of Optimal Experience* (Harper Perennial).

See also Ian Bogost, *Persuasive Games* (MIT Press).

Andrew Burroughs and IDEO, *Everyday Engineering* (Chronicle Books), p. 135.

This is called *pattern recognition*: the desire to seek out regularity in any facet of life. Sometimes this can work against us, especially in data analysis: see also Jason Cohen, "The Pattern-Seeking Fallacy," *A Smart Bear*, <http://blog.asmartbear.com/pattern-seeking-fallacy.html>.

effort is required to get into a state of flow, and what barriers we can remove to make that process easier and faster.

6.3.1. *Encourage people to invent their own rhythms.*

Rhythms are formed when someone unthinkingly repeats any brief action. You commit an action to memory over time, so you can remember how to do it in the future. In this way, early instances of an action can affect your expectations in the future. Well-designed products form a positive feedback loop that improves output, decreases a repetition's completion time, and can continue indefinitely.

Rhythm trains us, shaping our expectations. Keyboard shortcuts, inline help, visual cues, and keyboard- or mouse-only applications all help to encourage rhythms.

6.3.2. *It should take only one step to shift between two of a rhythm's repetitions.*

Consider switching between successive form fields, moving between rows or columns of a spreadsheet, or switching between currently running applications on a desktop computer. All of these require only one step, which makes it as easy as possible to adopt a rhythm.

Conversely, shifting between applications on many smartphones requires two steps: going to the home screen (or opening a list of recent apps), then selecting a new app. This excess step has the potential to be reconsidered.

6.3.3. *Compose elements with a visual rhythm.*

The best way to enforce a readable layout is by placing elements such that people can guess their locations. If two elements are placed five pixels apart, maybe a third should be placed five pixels apart from the second as well. This subconsciously trains people to scan the layout in a consistent manner.

Towards that end, a **grid** is a consistent vertical or horizontal division of a layout, comprised of a column width, margin width, and number of columns. A **typographic baseline** is a specific line height for enforcing the placement of content across columns of a grid.

Most software and web layouts use grids of flexible height and a typographic baseline instead of a fixed number of rows. That said, try to enforce a horizontal grid and a fixed height when possible—and make the width : height ratio a deliberate value (such as 2 : 1, 16 : 9, or the golden ratio ϕ : 1).

Grids are the best way to bring visual rhythm to any layout. They establish the priorities of various elements, conveying them quickly and clearly, because they are arranged in a systematic and structured way.

Elements can span one or more columns of a grid, but they should be left- or right-aligned to a given column; centered elements risk appearing

For more on flow and memory, check out Jack Cheng, "Habit Fields," *A List Apart*, <http://www.alistapart.com/articles/habit-fields/>.

For years, experts have finely tuned their workflows with keyboard shortcuts and fine-grained preference customization, adopting more complex rhythms than less experienced people.

Switching between applications involves holding down a keyboard shortcut.

See 6.5 for more information on modes.

visually erratic. Here's a four column grid with generous margins that are $\frac{1}{4}$ the width of each column:



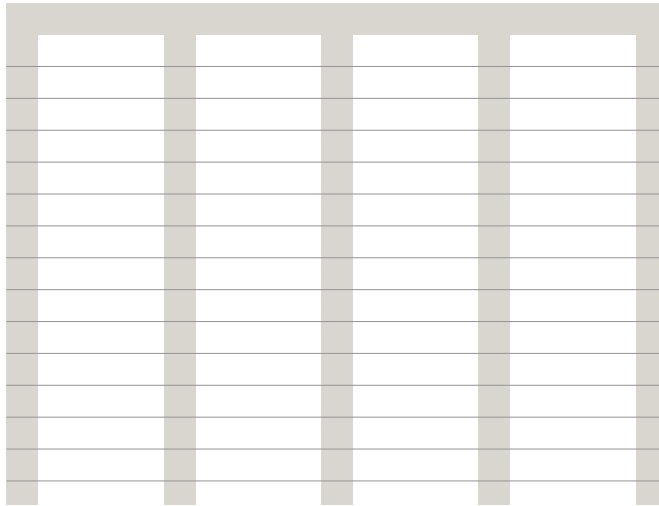
Most designers use translucent overlays to indicate grids when they're designing or wireframing on a computer. When I sketch grids, I use a thick, subtly off-white marker to indicate the margins. That way I can snap elements to the grid without distraction.

While the possibilities are endless, grids come in a few traditional flavors. Choose the right grid based on what elements you have to organize. For example, a web page with five primary navigation elements would prosper with a five-column grid. If you need more than five columns, subdivide each column to yield ten or twenty. Some common grid divisions include 3/6/12 columns, 4/8/16 columns, and 5/10/20 columns.

Grids beyond 20 columns cease to be effective because it's harder to differentiate between adjacent columns as a layout is scanned. Fine-grained subdivisions should be used sparingly; the best grids are simple grids, numbering 6 columns or fewer.

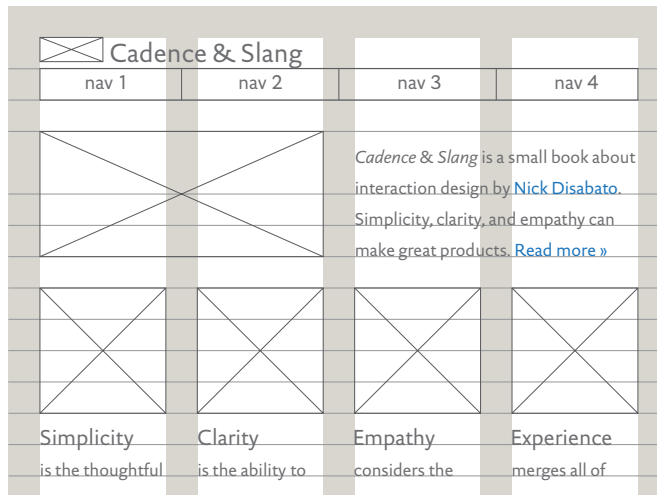
A good baseline is around 120% to 140% of your body text's font size. All elements should be aligned to the baseline.

We can impose a baseline upon our previous grid:



The layout has only one baseline, which makes the text in one column line up with the text in another.

Adding some sample content shows this:



Not only do the text blocks fit the baseline, but images are sized to be a consistent multiple of the baseline as well.

Larger text can fit a taller baseline, or be doubled in size to fit a shorter one. Short baselines are well-suited to long blocks of text, and tall baselines are better for headline– and image-heavy layouts. In either case, though, the baseline height should be consistent across the product.

Cadence & Slang's body text is 11pt with a 15pt baseline. The sidebar columns are 10pt text with a 12pt baseline.

On the web, many frameworks exist to enforce grid divisions and a baseline: plug in the values that you want, and a script calculates the corresponding stylesheet. *Blueprint* (<http://blueprintcss.org>) is an early, influential framework.

Print designers use grids and baselines to make text more readable and layouts more scannable. Pioneered in the mid-twentieth century, grid systems are also used in laying out interfaces, and they're crucial to ease of use.

Josef Müller-Brockmann sums up the sentiments behind grids:

The use of the grid as an ordering system is the expression of a certain mental attitude inasmuch as it shows that the designer conceives his work in terms that are constructive and oriented to the future.

Josef Müller-Brockmann, *Raster Systeme Für Die Visuelle Gestaltung* (Niggli), p. 10.

He continues:

The systematic presentation of facts, of sequences of events, and of solutions to problems should, for social and educational reasons, be a constructive contribution to the cultural state of society and an expression of our sense of responsibility.

Ibid., p. 12.

Grids make elements easier to organize and text easier to read. They impose rigor on the entire layout.

Grids give a sense of familiarity and inspire confidence. They form the rhythm of your layout—to know that if one element is *here*, it stands to reason that the next element will be *there*. And while they're not essential for good interactions (as a consistent tempo isn't essential for beautiful music), they give structure to layout, making elements more readable and findable.

Some designers believe that grids are too constraining, that they stifle creativity. But they're a good frame for creative solutions, and too subtle a constraint to negatively affect your design. Grids form the tempo of your layout; they dictate the sensible placement of elements and do a sizable amount of work for you. Plus, if you find your current grid confining, you can always work with another grid.

1.9 has more on the benefit of constraints.

Grids can be imposed on *any* interface, and you should always use a grid and baseline whenever possible. Their benefits far outweigh the risks of not using them: inconsistent, incoherent layouts are much harder to use, and can quickly become frustrating. This rule should only be broken if you're absolutely sure how and why you're doing so.

6.4. Express quantitative information densely.

Quantitative information can be displayed in any form. The most common ones are graphs and maps, but DNA, train schedules, football plays, and countless other things have taken radically different forms.

Edward Tufte, *Envisioning Information* (Graphics Press), pp. 88-89.

In his book *Envisioning Information*, Edward Tufte proposes the axiom “to clarify, add detail.” The average human eye can distinguish between extremely small distances: 15cm from your face, you can distinguish a gap of .026mm, or about 977 distinctions per inch (DPI, shorthand for how many distinct lines the human eye can see in one linear inch). This is far more

Focus Magazine, “How Small Can the Naked Eye See?” <http://sciencefocus.com/qa/how-small-can-naked-eye-see>.

than any current displays, so more detail will probably be easier to discern than you think.

The technological limitations of most current platforms pose challenges to successfully presenting dense information. The human eye can see up to 3,000 DPI, but the average computer screen remains, as of press time, 130 DPI. Smartphones currently run between around 230 and 320 DPI. Current electronic book readers are around 220 DPI.

Many graphs and charts—especially bar graphs and plotted line graphs—offer only a few points of information, but it’s been proven (most notably by Tufte, in *The Visual Display of Quantitative Information*) that we’re capable of understanding much more. According to Tufte, good information display should provide comparisons between nearby values, show multiple variables in two-dimensional space, and integrate text with images and data. **Information density**—how many data points are expressed in a given area—is important for these goals, but it becomes especially crucial in an interface. Technology has a tremendous opportunity to display large data sets creatively, so people may understand the powerful narratives that they imply.

Well-made print media continues to have a huge advantage over electronic media when it comes to dense, narrative-based information display, and sadly, low-resolution displays continue to be built. People have implemented creative solutions to work around the constraint of bad resolution, from zooming in on visually complicated images to providing multiple overlays on a map or graph.

6.4.1. Use sparklines to provide lightweight, contextual information display.

As mentioned in 2.8.3, **sparklines** were invented by Edward Tufte as a way to describe short series of data in a narrow, word-length space. For example, here’s a sparkline of the 97-64 win-loss record of the Chicago Cubs in 2008, where up denotes a win, a horizontal line indicates a home game, and a red tick means the game was won or lost by two or more runs:



This information is dense enough to describe a set of games with as much meaning as a full article on the subject. Notice the winning streak about two-fifths of the way through, and the textbook fall slump in the last fourth of the season. Compared against other teams’ win-loss records, readers can also determine whether the Cubs made the playoffs. Depending on one’s fan loyalties, this sparkline affects different people in different ways, and the presentation gets out of the way to tell the right story. For example, a Cubs fan might relish the early winning streaks, but a fan of their rival Cardinals might focus on when those wins took place, or they might care about the margins of victory.


The first 72 DPI displays were mass-marketed eighteen years before this book’s publication.

Kenya Hara states that before the insurgency of digital graphic design, Japanese printmakers were trained to draw ten distinct lines between two lines a millimeter apart. Now you can do that on a computer in two minutes. *Designing Design* (Lars Muller), p. 146.

Sparkline created from Bryan Donovan, “Custom MLB Team Sparklines,” *Hardball Times*, <http://www.hardballtimes.com/thtstats/main/sparklines/custom.php>.

They did, but then lost in the division series as expected.

6.4.2. Give graphs context and make them truthful.

Information shouldn't be kludged. Make sure your axes are accurate. Don't put breaks in your axes like  unless you absolutely have to. Make sure the scale of your x-axis is the same as your y-axis when you're comparing two of the same unit (time, distance, weight, etc).

It stretches the truth to blow scales out of proportion. Tufte defines the **lie factor** of a data display as the size of an effect shown in a graphic, divided by the size of that same effect in the data. Ideally, the lie factor should be 1.0, showing a 1:1 mapping between perceived magnitude and data increase. In *The Visual Display of Quantitative Information*, he shows a graph with a lie factor of 59.4, showing a size increase of 27,000 percent to express a data increase of 454 percent. Keep your graphs' lie factor at 1.0, and don't add anything to an information display that distracts your audience from understanding the data.

For more on distracting information, or *chartjunk*, check out 3.7.

6.5. Avoid modes.

In 2.2, I defined a **mode** as “any kind of setting that someone can trigger to activate a different set of behaviors in an otherwise identical interface.” The caps lock key is an obvious kind of mode, but modes exist in software, too.

A mode is one of multiple (usually two) states; a **state** is a behavior (or set of behaviors) that occurs when triggering an element that is affected by changes in the mode.

In his book *The Humane Interface*, Jef Raskin formally defines an interface as:

modal with respect to a given gesture when (1) the current state of the interface is not the user's locus of attention and (2) the interface will execute one among several different possible responses to the gesture, depending on the system's current state.

Jef Raskin, *The Humane Interface* (Addison-Wesley Professional), p. 42.

If a lightbox or dialog box prevents further input in a product until acted upon, it's a mode. Toggling italic text in a word processor is a mode. When you change tools in a photo editing program, each tool is itself a mode.

By definition, a mode's state is outside your locus of attention, which doesn't fit your expectations and can lead to errors. Modes can also restrict the paths you can take by disabling the rest of a layout. This indicates either issues with the code (it can't do the dirty work of handling different inputs) or your layout (it isn't naturally conducive to the most common modes).

Modes are harbingers of deeper design flaws. Good products are modeless, and modes should be eliminated whenever possible. Modes have been viewed as a necessary evil in our field for decades, but they've become increasingly hard to justify when we now have so many ways to eliminate them.

Dialog boxes don't have to be modal, but if they disable the rest of the layout until action is taken, they are.

Alan Cooper, *About Face* (Wiley), p. 430:

A dialog box is another room; have a good reason to go there.

4.2.2 has more on mode errors.

6.5.1. *Feedback should be modeless.*

Feedback becomes modeless when it's built into the layout. Modeless feedback doesn't interrupt the task, and it's well-suited to show any sort of status. You should replace modal feedback with modeless feedback whenever possible.

As displays support higher resolutions, more feedback can be executed modelessly. Icons can convey more than just the file type: half-downloaded files from a web browser, for instance, can have their preview icons replaced with a small progress bar. Hover states and marginal text can convey more information as you hover different parts of a map or data table.

One good example of modeless feedback is the character count and line number in plaintext editors. These data exist inline, without having to pull up a separate word count dialog. And on the web, error and success alerts can be executed with inline copy, rather than a modal pop-up.

The trade-off is that modeless feedback adds elements to your layouts. This can be distracting, especially for novices. Be careful about what kind of feedback you choose to add, and how and when it appears.

6.5.2. *Don't disable the layout with any dialog prompt.*

One of the most common ways to prompt another step is with a dialog prompt: either asking for preferences, or the infamous "Are you sure? OK/CANCEL" confirmation message. But disabling the layout with any dialog prompt or modal overlay requires people to expend excess effort on understanding the new layout and what's now required of them.

Ideally, dialogs should be placed inline. If modal prompts must be used, the rest of the layout should be pliant, and it should be possible to dismiss the overlay by clicking or tapping outside it.

6.5.3. *Remove modal dialogs through layout changes.*

By shifting elements around and approaching behavior differently, you can move once-modal displays out of the way and allow greater freedom of input.

Here are some easy ways to remove modes through layout changes. In all of these cases, modeless dialogs should be applied with consistent appearances and behaviors.

A SECOND COLUMN.

Consider a two-column layout: your primary layout on one side, and feedback on the other. This sort of layout makes feedback dependent on primary input, which could mean that it would change after every step. Two-column layouts like this are more usable, but likely take a little more work to implement, and wouldn't work so well on smartphones.

This may be slower going than I let on here. For more on the sad state of display resolutions, see 6.4.

A good process for designing without modes on the web: Aza Raskin, "Mobile Firefox and Designing Without Modal Overlays," <http://www.azarask.in/blog/post/designing-without-modal-overlays/>.

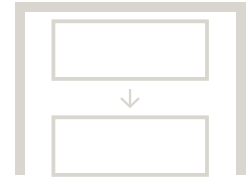
37signals also touches on this. Ryan Singer, "Modal Overlays Beyond the Dialog Box," *Signal vs. Noise*, <http://www.37signals.com/svn/posts/1149-modal-overlays-beyond-the-dialog-box>.

One noteworthy constraint, however: the smaller the screen, the less freedom you have to move elements, making modes harder to eliminate.



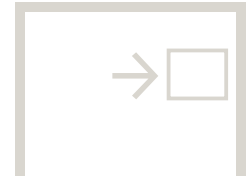
EXPANDING THE LAYOUT.

Consider expanding the layout to fit new elements when they're needed. A group of elements could move out of the way to fit more, or white space could exist preemptively. Both ways provide enough room, with many possibilities for conveying meaning without distraction.



OPTIONAL POP-UPS.

Consider popups that appear over the layout without disabling it. Only use these if they fit the function and layout more appropriately than any alternative, and if you provide an easy way to ignore them.



CREATIVE SOLUTIONS.

Strictly speaking, *any* layout is possible. If the solutions listed here don't work, then something else will. Be creative in solving layout problems. New possibilities will reveal themselves as new platforms and systems are released, with novel interaction models that this book could never foresee.

6.5.4. Replace modes with quasimodes.

Quasimodes are modes that require conscious input to activate and deactivate. Returning to the caps lock analogy: caps lock is a mode, while holding down the shift key is a quasimode, because it changes your keyboard into a different character set. Click-dragging to select multiple items is a quasimode; so is holding down the shift key and clicking on several items. Drop-down menus are a quasimode. Quasimodes are generally hard to find, but easy to learn once they're found.

Quasimodes benefit us because they eliminate the possibility of mode errors. At the same time, quasimodes are constrained because they can generally be used only to control your system, not to input information. Quasimodes also involve a higher cognitive cost, because someone has to think about when to turn them on or off.

Fortunately, most people know what quasimodes are, even if they don't call them by that name. They're very powerful tools for improving your product's functionality, with little development cost and a greatly reduced risk of errors.

7:

Cultural Norms

Different cultures form design slangs that speak well to some people and alienate others. At the same time, certain designs try to appeal to many people, across generations and cultural boundaries.

Your design can speak to as few or as many people as you'd like, but appealing to a lot of people has trade-offs, as does targeting very few. The fewer people you target, the more intimately you can communicate with them, and the more likely that you can pare down features, but there's a higher risk involved in making them happy. The more people you target, the more potential customers you have, but it might be harder to make a great, simple product that satisfies everyone.

Both of these extremes may be undesirable. It rarely makes sense to write products that target only a few people, or products that try to please the whole human race. There's a middle ground that amasses new business and preserves existing customers, but finding that can be difficult.

7.1. *Account for the plurality of cultural etiquette.*

All icons and nomenclature should be vetted for translation into other languages and use in other countries. Different cultures ascribe different meanings to various symbols. For example, in most of the Middle East, the thumbs up hand 👍 means roughly the same as the western world's middle finger. Would you want *that* for your OK button?

This also applies to products written for your home country. The common way to force a program to close on Linux or BSD is by typing "kill"—not exactly the nicest term, especially when you could use adequate synonyms like close, quit, or end.

7.2. *Account for the plurality of religious customs.*

Imagery that's innocuous in one country may be offensive or suggestive in another. For example, the Red Cross is called the Red Crescent in Islamic countries because of the religious connotations of a cross. More infamously, the swastika symbol is used in many Eastern religions, but its association with the Third Reich and Nazism tainted its meaning in much of the Western world.

When using any sort of symbols or iconography, make sure they're vetted to ensure a consistent meaning across cultures. Fortunately, there are quite a few resources online. Offline, Carl G. Liungman's *Dictionary of Symbols* (Norton Paperbacks) is a good source for tracing the history and context of iconography.

7.3. *Account for the plurality of non-Latin scripts and reading direction.*

This book is written in English, which is read from left to right, top to bottom. Other scripts across the world have different reading directions, which

At the very least, you should conduct a web search for the name of your symbol. An archive of public domain symbols is available from the Noun Project, <http://thenounproject.com>.

influences the way that native readers perceive any layout, whether printed or electronic.

Different reading directions affect the way that text is displayed, of course, but they also affect layout. Many websites reverse the left-right orientation of their layouts for Hebrew or Arabic readers, with navigation in the right sidebar, for example.

Thousands of scripts have their own conventions, and alphabets that may seem identical to English have their own quirks. For example, German and French have ß and Œ ligatures, Vietnamese has a vast and complex set of diacritics, and Hungarian hyphenates the common word *össze* as *osz-sze*.

Is your product sensitive to these concerns? Products native to their home countries likely are, giving them an advantage over yours. If you're developing a product in San Francisco that you want people to use in Thailand, then you should work to make it familiar and inviting to Thai as well.

7.4. *Define the tone of the design based on your target culture.*

Within specific cultures, people of different economic classes, interest groups, and experience levels are more likely to sign up for some services than others. As people gravitate around social networks and instant messaging services that address the needs attendant to their cultures, a regional online vernacular begins to take shape.

This is a controversial point—that poor people use one product and rich people another, or novices one and experts another—but enough data has been amassed over the past decade to strongly corroborate it. Frequently, products initially target just one demographic, broaden their reach to others, and then collapse under the weight of being everything to everyone. Products commonly fail when ported to new countries, because their designs don't meet the specific needs of that country's residents.

If we apply the principles in the rest of this book consistently—that simplicity is better, and that products should do less and target small groups of people—then it stands to reason that products should be written for narrow demographics. But taken to an extreme, this can isolate people and keep them from understanding one another. We already speak of “walled gardens” between social networks, but as technology becomes more prevalent, these walls can extend into our personal relationships as well.

One real-life example gives me hope, though. Here in Chicago, everybody uses public transit. The rail line that I ride to work every day is full of a diverse collection of people—all the more surprising in a famously segregated city. And yet we're all using the same thing to get around, because it's designed to be convenient, affordable, and demographically neutral. It's a success by that measure, and it gives me hope that all designers can create things that successfully appeal to many different people.

How do we include new groups of people? How do we understand the needs of others who are much different from us? And what are the broad cultural and sociological implications of what we build?

Hello, world.

שלום

“Hello,” Nick Disabato said.
«Bonjour,» dit Nick Disabato.
„Hallo”, sagte Nick Disabato.

For products developed in Latin languages, navigation should be put on the left sidebar, as people rarely read content on the right side of a page.

“össze” is a prefix for “together”.

For more on adapting your product to other cultures, see Molly E. Holzschlag, “Putting the World into ‘World Wide Web,’” 24 *Ways to Impress Your Friends*, <http://24ways.org/2005/putting-the-world-into-world-wide-web>.

See also danah boyd, “White Flight in Networked Publics? How Race and Class Shaped American Teen Engagement with MySpace and Facebook.” In *Race After the Internet*, eds. Lisa Nakamura and Peter A. Chow-White (Routledge), pp. 203-222.

A draft is available at <http://www.danah.org/papers/2009/WhiteFlightDraft3.pdf>.

7.5. Neutralize the product if different demographic groups will be using it.

We can potentially find demographic consensus by *neutralizing* a product with indistinctive colors, spare graphics, and traditional typefaces. Products with neutral design appeal more easily to a variety of people.

Simplicity could be the answer here, when it's executed with the intent to include others. By adopting a more neutral voice and eliminating any visual cues that could imply cultural connotation, many products speak to a wider range of people—and people more easily project their own values, aesthetics, and ideas onto the product.



7.6. Release separate products if cultural differences are mutually exclusive.

Translating a product into a new language is much easier than changing the product's cultural signifiers. But in order to appeal to an expanding audience, products need to be sensitive to all aspects of a culture.

When deciding how to adapt a product's interfaces for different audiences, changes to inessential features should be considered before the product's core functionality. For example, a product can safely be branded differently in various places; branding is separate from the product's most important features. Edge cases also represent less essential functionality, and they can be safely changed as well. But core functionality should be the *last* thing that you change, and only when it's absolutely necessary.

In situations where you find yourself changing a product's core functions, it may be more useful to release two separate products to cater to each audience. However, be careful to avoid providing so many separate releases that it becomes difficult to update or support them.

7.7. Conduct demographic research.

When you don't know how people will use your product, you have to figure it out. *Ethnographic research*, discussed in 4.1.2, is what designers use to find out how people work with technology. A considerable amount of writing has been devoted to ethnographic research, but it's worth closing this chapter with a summary.

If you're interested in pursuing these lines of thinking further, check out Kim Goodwin, *Designing for the Digital Age* (Wiley) and Indi Young, *Mental Models* (Rosenfeld Media).

Ethnographic research can take many forms. We can observe people using products, we can ask them to complete tasks in an isolated setting (similar in format to usability testing), or we can interview them.

RECRUITMENT.

Recruitment is often the most difficult part. Sometimes it involves an open call for recruitment on your site, asking colleagues and friends in person, prompting your customers, or tracking people down on the street. Sometimes, recruitment involves a more focused search, narrowing down by gender, age, experience level, or country of origin.

Either way, it's difficult to recruit qualified participants who will offer useful information. Because it may require too much effort to do in-house, firms frequently outsource this process to dedicated recruiting companies, who call random people and survey them to determine whether they're a good fit for the study.

CONTEXTUAL INQUIRY.

The process of informally interviewing people when they're completing a real-world task is called **contextual inquiry**.

Contextual inquiry involves *observing* real-world use and *asking questions* about it. Questioning can take place either during or after observation, depending on the circumstances and what kind of observations are made. If someone is visibly frustrated with a product, or about to commit an error, it may make sense to ask them about it at that time, to understand the process that led them to that point in their task. Asking questions during a task may gather more honest responses, but it also takes respondents out of the moment.

Seeing someone interact with a product in a real-world context gives a lot of ancillary information about how they work. You learn about physical affordances, personal customizations, and environmental factors in a way that formal testing simply can't provide. Also, contextual inquiry makes a person feel much more comfortable than if they had to complete predetermined tasks in unfamiliar surroundings. While less scientifically accurate, contextual inquiry can give you a more grounded perspective at a lower cost than full-blown usability testing, focus groups, or prototype creation and revision.

Make sure your interview process fits the task at hand. For example, say you're figuring out how people in a different country, of a certain age range, use mobile phones. Contextual inquiry can help: you go to that country, track down people in the street that are using a phone and fit the age range, and ask them questions about what they use, how they use it, and what they'd like in an improved setup.

The US government has written up some best practices in contextual inquiry: http://www.usability.gov/methods/analyze_current/learn/contextual.html.

More structured usability testing is discussed in 4.1.11.

Sections 4.1.8 through 4.1.10 discuss best practices in interviews with stakeholders and users.

Developing a spreadsheet application, on the other hand, probably doesn't require wandering down city streets. Instead, go into an office and observe people working with spreadsheets.

INTERVIEWS.

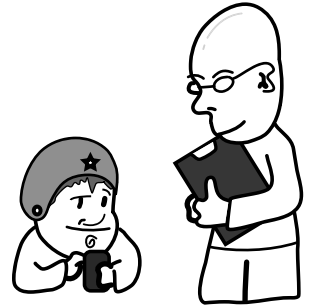
Interview people about what they use, what value they gain from it, and why they chose the product in the first place. How did they choose that product initially? What are their most common tasks? How have they customized it? How do they deal with errors? If the product is hardware, in what ways has it shown wear after repeated use?

Explore edge cases as well as common ones. How have lifelong experts customized their products? How have people with significant investment in your kind of product adapted it for their use? How do people of an intermediate skill level use your product?

IN SHORT.

Great products are often made when someone decides to fill a need of their own that current tools don't yet satisfy. In that way, she's developing for herself, and releasing it in the hopes that it will serve others similarly. But products can also be designed for others right from the start.

Before you design, you need to know who you're working for—and that's why research is such an important step.



Interaction design is an art and a craft. Art changes by the day; craft is stable. People find ways to design for new interaction models as soon as those models are created, but many design concepts remain fresh after many years. The focus of this book—on theory and technique over style and tactics—has been intentional.

It would be dangerous to over-think the problems that this book discusses; after all, if you follow the rules too closely, you might end up with something that's boring, confining, and over-engineered. It would be just as dangerous to ignore the products that target everyday life; these have to address a more skeptical market, and vigorous competition makes a product more interesting and useful. In *all* fields, though, it has become vital for developers and designers to understand the way that people experience things: the product is the interface, and the interface is the product.

Cadence & Slang's principles are useful only towards defining a framework, a starting point. Frameworks are helpful, but they don't make anything *whole*. The soul of the product isn't there yet. And while soul is the hardest part of a product to conceive and build, and nearly impossible to describe in text, any product without it is only partially complete. That's what you have to do next. I hope this book helps you along the way.

Nick Disabato
Chicago, IL
August 2013

A:

Conclusion

B:

Reference

Affordance: Anything that allows and encourages an action to take place. Affordances can be physical objects, properties of an object, elements of an interface, or a standalone product itself.

Baseline: A consistent vertical height to align every element so that text gains a visual rhythm and people learn where to expect subsequent elements. Baselines can apply to both print design and interactive layouts.

Behavior: A product's response to input.

Completion Time: The amount of time that it takes to finish a task.

Control: Any element that triggers behavior through clicking, typing, or touching.

Data-Ink Ratio: The amount of ink (or pixels) devoted to expressing data, divided by the total amount of ink (or pixels) in the layout. The data-ink ratio should usually remain as close to 1 as possible.

Element: Anything that modifies an interface, from text to graphics to buttons to form controls.

Error Rate: The number of errors that someone commits per path. (Hopefully zero.)

Experience: The sum of paths over the entire course of someone's individual relationship with a product.

Flow: The mental state of immersion in an activity. In interaction design, flow frequently occurs while involved in repeating a task or conducting a long task.

Grid System: A consistent division of a layout into columns and/or rows, providing a framework in which content is arranged. In an interface layout, the most common division is by columns.

Interaction: The sum of paths in a given relationship with a product, from when it's opened (or turned on, etc) until when it's closed (or turned off, etc).

Interface: The appearance and immediate context of a product, including layout, platform, system, and hardware.

Language: How the product communicates with people. This encompasses appearance, layout, copy, and behavior of elements.

Layout: The way that elements are arranged in relation to one another.

Mapping: A connection between each of a series of related controls and each of a series of corresponding behaviors.

Mode: Any kind of setting that someone can trigger to activate a different set of behaviors in an otherwise identical interface.

Natural Mapping: A mapping where controls follow the same order as their behaviors.

Normative Context: The subservience of a product's language and interface to its platform and system.

Path: A chain of steps that completes a given task.

Pattern: A repeatable solution to a common design problem, encompassing a definition of the problem; discussion of its essence and implications; and an abstracted, reusable, visually expressed solution.

Pattern Language: A series of interdependent patterns that express the common solutions of a product's design.

Pliant: The property of a control such that, when triggered, it prompts some sort of behavior. Elements are sometimes disabled, or rendered **unpliant**, to indicate that they shouldn't be triggered. Unpliant elements are often referred to as *ghosted*, because they are commonly tinted a light gray.

Platform: The product to which your product is subservient. For example, the operating system your product runs in is a frequently encountered platform.

Product: Any software or hardware that affords use. (Infrastructure, like cables or network switches, doesn't qualify: it's a means to an end.)

Quasimode: A mode that requires conscious input to activate and deactivate.

Repetition: A step, or set of steps, performed multiple times in a row to complete a collection of similar tasks.

Rhythm: A repetition performed periodically and consistently for a long period of time.

State: The set of behaviors dictated by a mode. Modes contain at least two states.

Step: A discrete action that triggers behavior.

System: The integrated set of products which comprise an interface—usually a computer, console, or smartphone.

Task: The desired outcome of one's interaction.

Transition: The behavior between two subsequent steps.

Usability Testing: Any method that quantitatively determines the usability of a product. Measured in terms of error rate, number of steps required, or completion time.

Individual key presses
are not steps, but
keyboard shortcuts are.

TEN BOOKS.

Christopher Alexander, *Notes on the Synthesis of Form* (Harvard Paperbacks), *The Timeless Way of Building*, and *A Pattern Language* (Oxford University Press): The original source of design patterns, to which section 2.6 of *Cadence & Slang* owes its debt. Christopher Alexander is an architect who invented pattern languages as a means of designing and planning humane cities and buildings. Since then, the idea of modular, repurposable patterns has found its way into many other walks of life, including education, object-oriented programming, and interface design.

Robert Bringhurst, *The Elements of Typographic Style* (Hartley & Marks): The inspiration for the format of *Cadence & Slang*, this book—part reference guide, part manifesto of the printed word—remains vital for anyone concerned with making readable, beautiful text both online and offline.

Alan Cooper, *About Face* (Wiley): First published in 1995, this book provided the scaffolding for the craft of interaction design. It focuses considerably on many topics covered in *Cadence & Slang* including the way elements are arranged and the way they behave from step to step. Many of the terms defined in *Cadence & Slang* were first coined here. It's currently in its third edition; make sure you acquire a new copy.

Kim Goodwin, *Designing for the Digital Age* (Wiley): Goodwin is top brass at Alan Cooper's company (cf. *About Face*), and her reference tome reflects their design process. Where Cooper more closely covers the rigor of behavior and layout, Goodwin provides deep insight into the research side of interaction design: how to recruit and interview people, how to synthesize findings in a team, and how to observe real-world use. This book is tremendously useful in organizations where contextual inquiry takes a front seat.

Steve Krug, *Don't Make Me Think!* (New Riders): This book does a great job of introducing readers to the way that people generally read websites, and it also provides a solid overview of how to conduct usability testing. This is one of the most accessible books on testing for the web in particular, and it remains relevant more than a decade after its original publication.

Per Mollerup, *Marks of Excellence* (Phaidon): A reference on the meaning and history behind various company logos. Useful for determining the meaning behind many symbols, and can provide inspiration for iconography.

Don Norman, *The Design of Everyday Things* (Doubleday Business): A concise overview of the way that the built environment can better fit our needs. Discusses affordances, common errors, and natural mappings.

Jef Raskin, *The Humane Interface* (Addison-Wesley Professional): One of the first books to discuss interaction design. Raskin was the first interface professional to work at Apple, during the dawn of the personal computer. Among many other topics, Raskin discusses the implications of modes in personal computing.

Jennifer Tidwell, *Designing Interfaces* (O'Reilly): The original book of patterns for interface design, this book defines common layouts and behavior.

Edward Tufte, *The Visual Display of Quantitative Information*, *Envisioning Information*, *Visual Explanations*, and *Beautiful Evidence* (Graphics Press): This quartet of books—a fifth is to be added in the coming years—remains the undisputed source for quality in information display. Edward Tufte's work elevates the presentation of data to a craft, honed after considerable practice. Printed in staggeringly high quality by Tufte himself, they're quite economical to own. Essential for when you're presented with a body of data and need some way to express it beautifully.

This edition's body and sidebar text is set in Documenta by Frank E. Blokland. The page numbers, running heads, and example interface messages are set in Skopex Gothic by Andrea Tinnes. Greek text is set in Didot, designed by Ambroise Firmin-Didot and digitally released by Takis Katsoulidis and George Matthiopoulos of the Greek Font Society. Hebrew text is set in Adobe Hebrew by John Hudson. The title and chapter heads are set in Zine Slab, designed by Ole Schäfer and issued by FontFont. Some of the original diagrams were left unaltered from the first edition, with their text set in National by Kris Sowersby as per usual.

CREDITS FOR THE SECOND EDITION.

While several dozen people were involved in the first edition of this book, the second edition's team was much smaller. This edition was edited by Kaitlyn Tierney. The drawing of Nick is by Jana Kinsman. Other illustrations are by Daniel Bogan. Erin Watson helped out with some of the writing as well, solving some pronoun troubles and sundry kerfuffles.

Thanks to all 812 of our Kickstarter backers for contributing to the independent publishing world in a small way. Your love is known and felt.

ABOUT THE AUTHOR.

Nick Disabato is an interaction designer and writer who loves making great things and having great conversations. He has very curly hair, and he runs a small design consultancy called Draft, which you can read about at draft.nu. He thinks you're probably great. For more information, and to get in touch, you can head over to nickd.org.

C:

Colophon

