

The Designer's Guide to the Buttons

A History of Interfaces from Early Computers to Virtual Reality.
In Search of a Universal Design System



Andrew Marcus

Andrew Marcus

The Designer's Guide to the Buttons

A History of Interfaces from Early Computers to Virtual Reality.
In Search of a Universal Design System

New York, 2026

Andrew Marcus

The Designer's Guide to the Buttons:
A History of Interfaces from Early Computers to Virtual Reality.
In Search of a Universal Design System

This book is a comprehensive guide to the creation and evolution of computer interfaces. Unlike conventional collections of rules and standards, it is built as a historical narrative that reveals the causal relationships behind modern interface solutions. By tracing how interfaces evolved, it helps readers understand the principles that govern them, not merely memorize ready-made patterns.

The book is divided into three parts. The first part covers desktop interfaces from Xerox Alto and Apple Lisa to modern macOS and Windows. The second part focuses on mobile devices—from Apple Newton to the iPhone and Android. The third part explores the future of interfaces: interaction in virtual and augmented reality, illustrated through Meta Quest and Apple Vision Pro.

The edition features hundreds of illustrations, diagrams, and screenshots, as well as examples from 180 popular applications. Thanks to its clear, sequential structure, the book will be useful to both new and experienced designers.

About the Book

THIS BOOK IS INTENDED FOR A WIDE AUDIENCE interested in UX/UI, product or interface design, digital technology, and the history of computing. It is valuable to both newcomers and experienced professionals.

Beginning designers will find it especially engaging to trace how familiar elements of interface—buttons, text fields, checkboxes, and dropdown menus—originated from their physical-world counterparts: electric bells, radio dials, paper ballots, and various mechanical devices. It's not merely a historical excursion but an invitation to see everyday things from a new perspective.

For professionals, the book offers a reason to reflect deeply on the nature of interface decisions. It helps answer questions that seem simple yet often prove challenging: Why is a component built the way it is? Should a button brighten or darken on hover? What is the best way to highlight a selected item in a list: by a check mark or with a color? What's the difference between a dropdown list and a dropdown menu? How to align form fields for the best clarity?

The book analyzes 180 interfaces from popular applications, both desktop and mobile. Hundreds of illustrations showcase classic and contemporary products alike: Google Docs, Notion, Airbnb, Uber, Microsoft Word, WhatsApp, Telegram, and many others. Designers often struggle to find solid, real-world examples of interface components; this book solves that problem by building visual literacy through direct comparison.

This book is not just a history of interface design—it is also a practical manual for creating a modern design system. Each chapter focuses on a single component and examines it in depth: from its very archaic origins to its modern anatomy and a universal matrix of states, styles, and variations.

Several features make the book unique.

First, it includes 35 QR codes linking to popular articles, video tutorials, and supplementary materials, turning it into a truly interactive encyclopedia. You no longer need to google for a new term or search for a live example of the interface in action. Just scan the code with your phone!

Second, all the design principles described in here are consolidated into a real, working design system, published on the Figma Community and used in large-scale international projects the author has worked on.

Third, the structure of the book allows complex sections—such as component matrices—to be skipped without losing comprehension, letting you simply enjoy the story and visuals.

Finally, it is the only book in the world with an intermission.

Contents

Part one. Desktop Interfaces	8		
<i>Chapter 1. Invention of the Button</i>	<i>10</i>	<i>Chapter 6. List Box</i>	<i>72</i>
Skeuomorph Attacks!	12	Multi-Select List Box	74
Ancient Buttons	14	Searching a List	75
Button States	16	Anatomy	76
Primary and Secondary Buttons	18	Component Matrix	78
Enter Photoshop	20		
UI Kits and Design Systems	22	<i>Chapter 7. Dropdown List</i>	<i>79</i>
Photoshop Exits. Enter Figma	23	Dropdowns Debunked	80
Neomorphism	24	Dropdown Lists Best Practices	82
Liquid Glass	25	Searching a List	84
The Anatomy of a Button	27	Combo Box	85
Container	27	List Scrolling	86
Left Icon	29	Multi-Select Lists	87
Right Icon	30	Component Matrix	89
Counter	31		
Button States	31	<i>Chapter 8. Dropdown Menu</i>	<i>90</i>
Focus Mode	32	Ways to Open a Menu	91
Sizes	33	Paper and Computer Menus	93
Component Matrix:	34	Anatomy	94
Bringing it All Together		Classification of Menu Sections	98
		3D Menus	99
<i>Chapter 2. Toggle Button</i>	<i>37</i>	Left and Right	100
Component Matrix	40	Searching a Menu	101
		Component Matrix	103
<i>Chapter 3. Checkbox</i>	<i>43</i>	<i>Chapter 9. Text Field</i>	<i>107</i>
Statuses	45	Material Design's Flying Circus	109
Styles	48	Anatomy	112
Anatomy	49	Placeholder	112
Component Matrix	50	Icons	113
		Password Fields	114
<i>Chapter 4. Segmented Button</i>	<i>53</i>	Input Mask	115
Multi-Select Segment	56	Segmented Field	116
Anatomy	57	Suggestion List and Combo Box	117
Component Matrix	60	Text Area	119
		Component Matrix	120
<i>Chapter 5. Radio Button</i>	<i>61</i>		
Anatomy	65	<i>Chapter 10. Tabs</i>	<i>124</i>
Directions for Use	67	First-Level Tabs	127
Radio Checkbox	69	Second-Level Tabs	128
Component Matrix	70	Tabs of the Third and Fourth Levels	130

Segmented Button as Tabs	132	Multi-Select Lists	195
Vertical Tabs	133	The Components	195
Tab Overflow	135		
Unusual Tab Designs	136	<i>Chapter 17. Text Field</i>	197
Anatomy	138	Suggestion Lists and Combo Boxes	199
Component Matrix	139	Text Area	200
		The Component	201
<i>Chapter 11. Other Components</i>	141		
Toggle	141	<i>Chapter 18. Menus and Sheets</i>	202
Split Button	143	Ways to Open a Menu	204
Tag Input	144	Edit Menu	205
Slider and Range	145	Sheets	206
Calendar	145	The Components	206
Intermission	148	<i>Chapter 19. Tabs</i>	208
Form Design	151	Tab Bar	208
Components Equivalence	153	Minefield Interface	210
The Adaptation Trap	155	First-Level Tabs	211
		Second-Level Tabs	212
Part two. Mobile Interfaces	158	Tabs of the Third and Fourth Levels	213
<i>Chapter 12. History of Mobile Interfaces</i>	160	Side Menu	214
Newton	162	The Components	216
The Rest	166		
iPod	168	<i>Chapter 20. Cells and the Bento Interface</i>	217
		Tiles	217
<i>Chapter 13. The Thumb Zone</i>	170	Horizontal Lists	218
		Back to the Lists	219
<i>Chapter 14. Mobile Lists</i>	173	Bento	220
Anatomy	178		
Component Matrix	179	Part three. Mixed Reality	224
		<i>Chapter 21. Virtual Reality</i>	226
<i>Chapter 15. Mobile Button</i>	181	Augmented Reality	228
Button as a List Item	184	Mixed Reality	232
Button Groups	184		
Toggle Buttons	185	<i>Chapter 22. Virtual Interface</i>	233
Floating Buttons	185	Space and Environment	233
Anatomy	186	Virtual Windows	234
Component Matrix	187	Interacting with Reality	238
<i>Chapter 16. Checkbox, Radio Button, Toggle, Segmented Button, and List</i>	189	<i>Chapter 23. Button in Mixed Reality</i>	245
Dropdown List	192	Component Matrix	247

Before the Common Era

Common Era



1911
Founding of IBM



1976
Birth of Apple



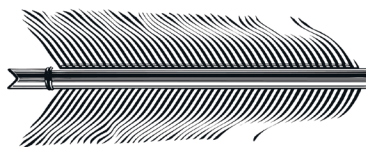
1968
First Mouse

1973
Xerox Alto Release

1985
Fall of the Empire

Release of Windows 1.0

1993
Launch of Newton



Prehistoric Period

Ancient Era

Middle



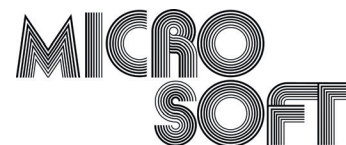
Charles Babbage
1791...1871



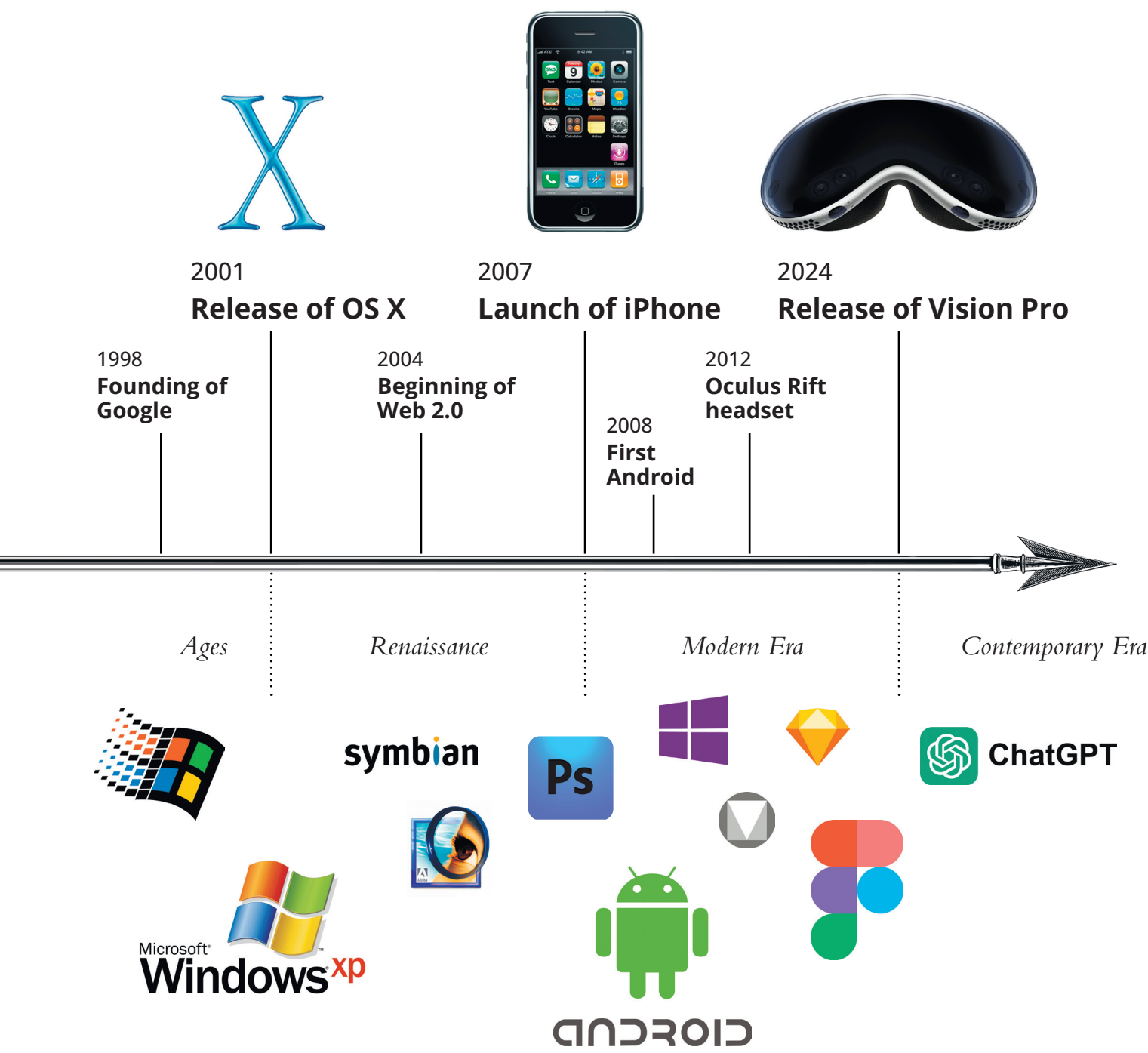
Alan Turing
1912...1954



Jeff Raskin
1943...2005



Common Era



Part one

Desktop Interfaces



The Invention of the Button



Doorbell, 19th century

WE KNOW ABOUT THE APPEARANCE of the first electric button roughly as little as we know about the invention of the wheel—even though the button was created in 19th-century Europe, and the wheel appeared around year -3500 in Mesopotamia.

Rachel Plotnick, in her book *Power Button*, writes that buttons emerged “no later than 1880” and were used to close an electrical circuit. And although people had already been using Morse code for almost fifty years by then, the device’s little gizmo was technically a lever, not a button—so Plotnick is probably right.

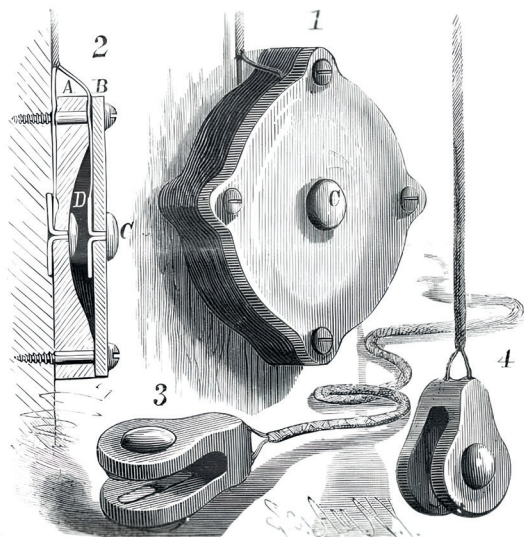
The earliest buttons were made mostly of bronze and framed with wide panels packed with decorative details, as was the case with just about any object from the Victorian era. They were primarily used for doorbells and calling hotel staffs. You can still find such bells in some European cities today, or buy a modern reproduction on eBay.

Curiously, at first, buttons—like electricity itself—provoked genuine fear. People in the 19th century were far more terrified of electricity than we are today of “artificial intelligence.” It’s easy to understand why. After all, I have never heard of a ChatGPT chair, but the electric chair was already in use back then for executions.

As for the buttons themselves, even though they quickly became popular, many educated people in the late 19th century worried that if everything was simply “left to run its course,” buttons would destroy the younger generation’s ability to navigate the world. People would get so used to buttons, they warned, that they’d forget how to do anything with their hands. And although few doubted the usefulness of the new invention, it was nonetheless believed that a person should first study the principles of electrical circuits in detail, then the inner workings of the buttons themselves, and only then start pressing them.

SIMPLE PUSH BUTTON FOR ELECTRIC BELLS.

Mr. Gosonko, of Kozloff, Russia, sends us the following simple device for a key or push button for electric



GOSONKO'S PUSH BUTTON FOR ELECTRIC BELLS.

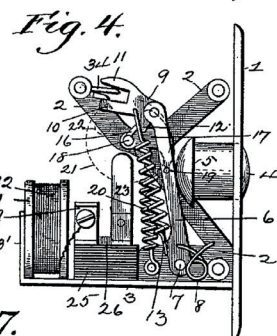
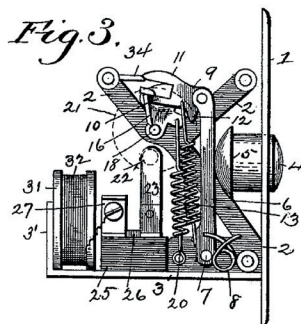
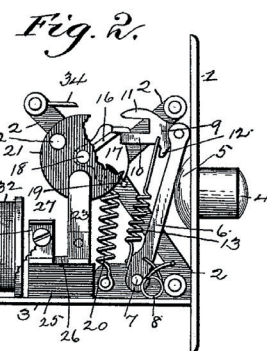
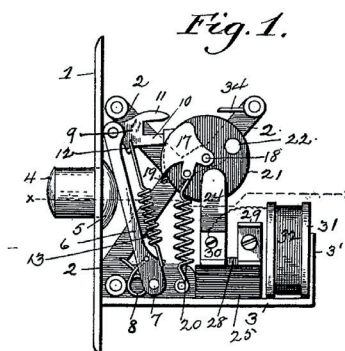


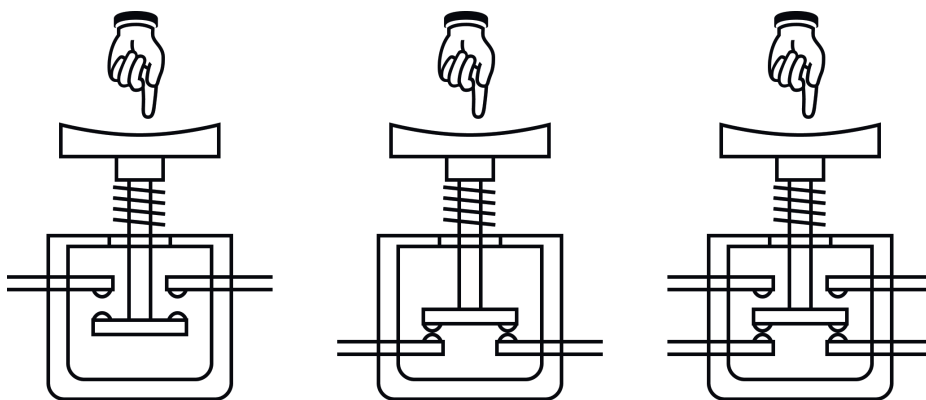
Fig. 7.

Let us, too, take a careful look at how a button actually works.

Its most complicated part lies not in the press itself, but in the mechanism that returns it to its original position. Over the years, inventors proposed all kinds of ways to “unpress” a button. The simplest was to use a spring or mount the button on a flexible rubber membrane, as suggested in 1885 by a certain Mr. Gosonko from the town of Kozlov, who sent his invention to *Scientific American*. Other designs were far more intricate—like the doorbell button patented in 1894 by the American engineer McLaughlin, which consisted of thirty-three individual parts.

Left: a drawing of Mr. Gosonko's button from Kozlov, *Scientific American*, № 26, 1885

Right: a drawing of American engineer McLaughlin's button, U.S. Patent № 521.808, 1894



Different buttons use different mechanisms, but the underlying principle is the same

However different their mechanisms might be, all buttons—whether made of three parts or thirty-three—worked in essentially the same way: they simply closed an electrical circuit by connecting two wires. Once the circuit was closed, current flowed through it, activating the bell. Simple as that.

Once the passion for doorbells had been satisfied, by the end of the 19th century, buttons began to be used for all sorts of new purposes. They started appearing in all kinds of devices of the era. In 1888, Kodak launched a famous

advertising campaign for its new button-operated cameras, whose slogan declared: “You press the button, we do the rest.” Another company, Eveready, received a patent in 1899 and released the first battery-powered flashlights with a button switch, using batteries of its own manufacture, today known under the Energizer brand.

Kodak advertisement for its button-operated cameras, 1888

New Kodak Cameras.

*“You press the button,
we do the rest.”*
(OR YOU CAN DO IT YOURSELF.)

Seven New Styles and Sizes

ALL LOADED WITH
Transparent Films.

For sale by all Photo. Stock Dealers. *Send for Catalogue.*

THE EASTMAN COMPANY, ROCHESTER, N. Y.

Buttons very soon conquered more serious uses. By the late 19th century, American homes were rapidly replacing bulky wall-mounted levers with modern and compact electric light switches. Around the same time, buttons began appearing in elevators, which had previously been controlled by levers and powered by hydraulic or steam systems. A little later, in the 1910s, cars began to be started with buttons. Twenty years after that, buttons found their way into radios. By the 1950s, cars were experiencing a boom in button-filled dashboards, and by the 1960s, buttons had taken over virtually all consumer electronics: televisions, telephones, washing machines, tape players, arcade machines, and even airplanes.

The triumphant march of electric buttons around the world met almost no resistance—right up until the mid-1970s. Only one last bastion stood in the way of their global domination, a small dark cloud in an otherwise perfectly clear sky: a recent invention, the latest sensation in electrical fashion—the personal computer. Yet just short of the podium, something unexpected happened: buttons collided with a force they could not overcome, and from that moment began their slow but steady decline.

On March 1, 1973, the Xerox PARC research center introduced the first graphical user interface.

Skeuomorph Attacks!

DECADES HAVE PASSED SINCE THEN, and one thing is clear today: the number of real, physical buttons around us has dropped dramatically. Sure, computers still use more than a hundred buttons on a keyboard, but that hardly compares with the number of virtual buttons we press on a screen every day. Buttons never managed to conquer the personal computer. Instead, the computer swallowed the buttons, sending their souls to a virtual heaven known as the graphical user interface.



Calculator in early iOS imitated the Braun ET66 as part of skeuomorphism

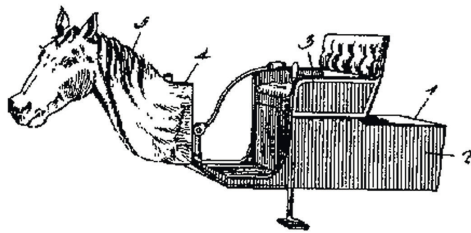
What's interesting is that today we barely notice the substitution. Yet on-screen buttons are nothing like the real ones. A virtual button offers no physical feedback whatsoever. While a real button can be felt and pressed with your eyes closed—you can sense its springy travel or hear a click—a virtual button can only be pressed while looking at the screen, and you can't feel it at all.

Still, on-screen buttons try their best to behave like the real ones. They have shadows, highlights, depth, and sometimes even sound—especially when the designer is dealing with a stubborn client. And while designers often overdo the decoration, all these visual tricks didn't appear by chance.

As early as the late 19th century, inventors realized that the brave new world of buttons and electricity was too strange for people accustomed to riding horses and dining by candlelight. Lamps, electric doorbells—all of this frightened people because it was unfamiliar. To make new devices less intimidating, inventors tried to give them the shapes of familiar objects. Electric chandeliers, for example, were designed to look like ornate candle candelabras, and light bulbs were sometimes shaped like flames.

Often, this kind of imitation really did help people adapt to the new world, but sometimes it went too far. The most famous example is the legendary Horsey Horseless. In an effort to make automobiles less frightening for horses and villagers, one American inventor came up with the idea of attaching to the car a full-size horse's head. Luckily, the invention never went beyond the patent stage; otherwise, instead of gleaming hoods, Henry Ford's cars might have been outfitted with massive bronze horse heads.

Fig. 1.



Left: drawing from
Patent № 30551 for a
horse-headed automobile

Right: illustration from
Historical Motor Scrapbook

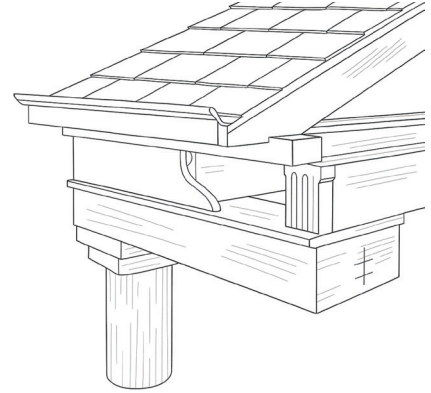
You have probably already guessed what this is about. This kind of imitation—when the design of a new invention mimics that of an older one—is what we call *skeuomorphism*. And while the word itself is unfamiliar to most people, the phenomenon is known to anyone who has ever held a phone.

When people talk about skeuomorphism, they usually mean the design of early versions of OS X and iOS, where every interface element was styled after a real-world object. In the Notes app, for example, the notepad was drawn as sheets of paper stitched together with a decorative coil; the camera simulated the narrowing of a mechanical aperture when taking a photo; and every button carried a stack of shadows, a gradient, and a texture.

These are indeed good examples of skeuomorphism, but in reality, it isn't limited to the interface of an early iPhone. In fact, it is *any* imitation whatsoever in which a new object mimics an older one in order to preserve familiarity.

And although the word appeared in the late 19th century, the phenomenon itself has been with us for thousands of years. The triglyph, for instance—the pattern of three vertical bars running along the entablature of ancient Greek temples—has no architectural function at all. It just imitates much older wooden temples, where the pattern appeared simply because the ends of wooden beams stuck out that way.

Triglyph in ancient Greek temples—a skeuomorphic element imitating the beams of older wooden temples



The graphical user interface introduced by Xerox PARC in 1973 became a monumental turning point in the history of technology. This leap was far more astonishing than the transition from wooden temples to stone ones, or from flaming candles to electric lightbulbs. Perhaps that is why almost every element of the graphical interface is—or at least once was—a skeuomorph, mimicking some object from the real world.

The engineers at Xerox brought to life something that had existed only in imagination and a couple of Hollywood movies. The graphical interface was so unlike anything people had seen before that it *had* to imitate the real world for an ordinary person to understand it.

The book you are holding in your hands tells the story of the inner logic of the graphical interface, uncovering the deepest mechanisms that make it work. This book is a kind of reverse engineering for the world of design. Step by step, as we reconstruct the history of each element—button, checkbox, dropdown list and so on—we will come to understand why they look and behave exactly the way they do, and not otherwise. Like the first users of electric buttons, we are simply obliged to understand the inner workings of virtual ones.

I am convinced that a truly complex, coherent, and extensible interface can be designed only by someone who thoroughly understands its underlying logic. And that logic can be grasped only by bringing up from the bottom of history the original problems and their solutions that made modern interfaces what they are today.

Ancient Buttons

WE BEGIN WITH BUTTONS, the simplest element of the interface.

Their Ancient Era began in 1973 with the release of the Xerox Alto, the first personal computer with a graphical user interface. Along with it came the first

interface button, marking the end of the prehistoric age of physical buttons. It looked as simple as possible: a short English label enclosed in a checkered frame.

Obviously, the designers modeled the virtual button after real, physical ones, which were often rectangular. But the idea of placing the label inside the button—that was something new. In the real world, text is rarely put directly on a button, since simple paint wears off quickly under a finger. That’s why physical buttons are usually labeled below or beside the control. Just imagine what an interface would look like if the buttons were round, with labels sitting off to the side. Fortunately, such design appears only in video games and simulators. Then again, checkboxes and radio buttons actually look exactly like that—but more on that later.

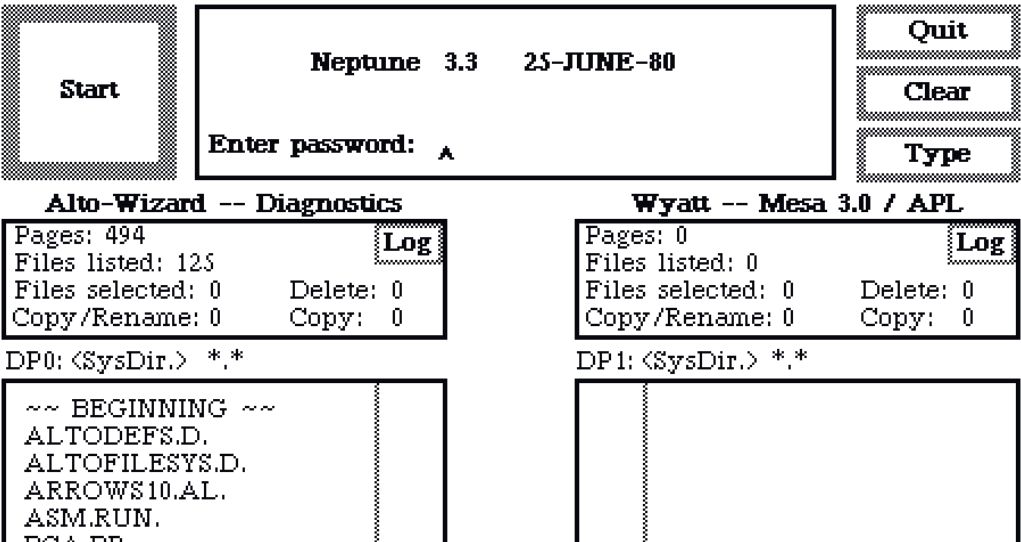
Despite their primitiveness, ancient buttons followed one of the core rules of interface design from the very beginning: since a button performs *an action*, its label should be *a verb*.

Let’s look at one of the first file managers in history, the Neptune app for the Xerox Alto. It was the ancient analogue of Windows Explorer, though its two-column layout makes it look more like Norton Commander. We not concerned with file operations right now; what is interesting is the top control bar, which contains several buttons: *Start*, *Quit*, *Clear*, and *Type*. Their meaning is easy to guess from the name alone, and even if you remove the outline, the verb label still makes it clear that this spot can be pressed and an action will occur.



Buttons in game interfaces and stylized UIs can be round, with labels beside

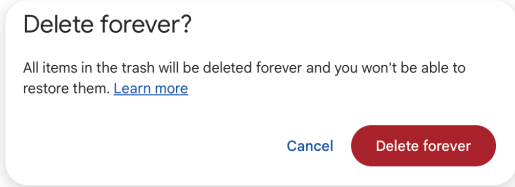
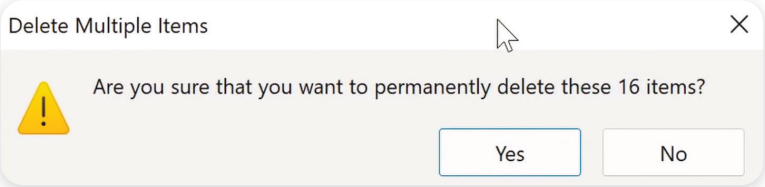
Top: elevator in France
Bottom: Fallout menu



Neptune 3.3 file manager for Xerox Alto computers

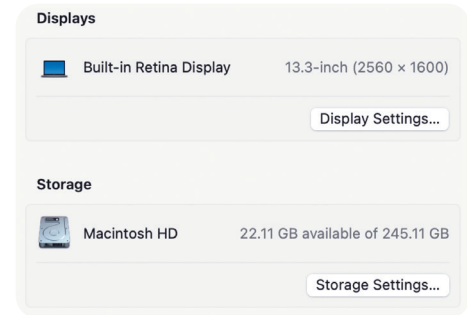
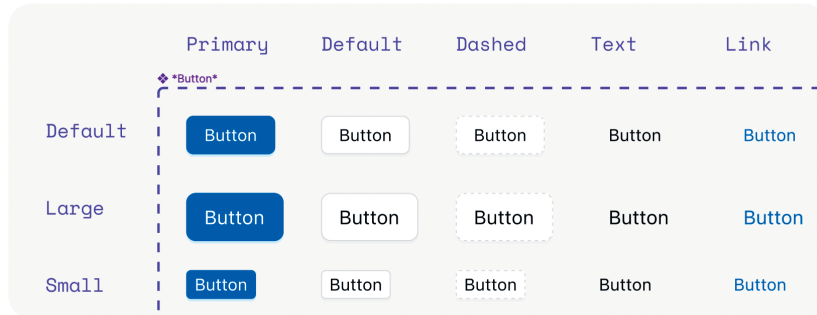
Although this rule was established back in 1973, designers still often forget about it today, which can seriously degrade the interface. For example, when choosing between *Yes* and *No* in a file-deletion dialog, the user is forced to read a long and often useless block of text. Meanwhile, choosing between *Delete* and *Cancel* would be obvious without any explanation.

Windows: the trash cleanup dialog forces to read the message; Google Drive makes the choice obvious



Left: Ant Design system, an example of mixing a button and a hyperlink

Right: when a button leads to another screen, a noun with an ellipsis is used instead of a verb



This reflects a systemic misunderstanding of subtle logical distinctions that, once ignored, can snowball into serious mistakes in complex interfaces. For example, links and buttons cannot be merged into a single component, if only because links have a different set of states: in addition to *Hover* and *Pressed*, they can also be *Visited*. But details like these are routinely ignored today.

Old vs. new Amazon AWS sign-in dialog. Designers can't decide whether *Sign in using root user email* should be a link or button

Sign in as IAM user

Account ID (12 digits) or account alias

IAM user name

Password

☐ Remember this account

Sign in

[Sign in using root user email](#)

[Forgot password?](#)

IAM user sign in

Account ID (12 digits) or account alias

IAM username

Password

☐ Show Password [Having trouble?](#)

Sign in

Sign in using root user email

Button States

IT SEEMS THIS IS THE FIRST TIME I've used the term *state*. Physical buttons have two positions: the default one and the pressed one, in which the button stays

for a fraction of a second, sinking into the panel and then springing back to its original place. Virtual buttons cannot be pressed with a finger; they have nowhere to sink and no pressed position. And, in fact, we don't really press on-screen buttons. We press the mouse button, and the "pressing" of the on-screen button is nothing more than an illusion.

When we press the mouse button, we get tactile feedback in the form of a click. If, at that moment, we focus on the on-screen button and forget the mouse exists, our perception creates the illusion of pressing the button on the screen. The same effect allows a person to operate a bicycle or a car without thinking: we don't consciously feel the handlebars or the pedals, as if we merge with the mechanism into a single whole.

This idyll is disturbed by only one flaw: the on-screen button still doesn't get actually pressed, and while a person does receive tactile and auditory feedback from a mouse click, that feedback is not reinforced visually. Therefore, to make the illusion fully work, the button must visibly react to being pressed.

Incredibly, those very ancient designers who introduced the world to the first graphical interface took care of this subtle detail right from the start: when an interface button was pressed in the Xerox Alto, its background turned black for a split second. This technique was nothing more than an imitation of the shadow cast by a physical button when you press it with your finger.

It would be incorrect to call such an imitation a position, which is why we use another term today: *the state* of a button. Thus, we can say that buttons in the Xerox Alto interface had two states that later entered designers' vocabulary under familiar names: *Default* and *Pressed*.

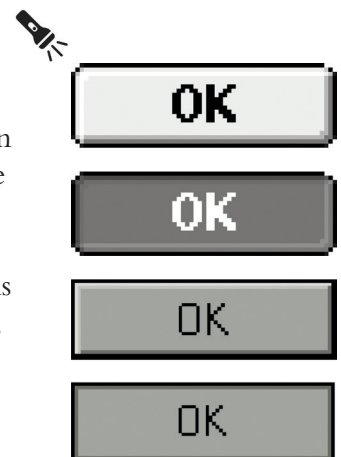


Xerox Alto button: Default and Pressed states

The black color was chosen to imitate a pressed state simply because there were no other options: at the time, all displays were monochrome. Later, as displays began to support shades of gray, the idea was developed further, and designers started to imitate lighting in greater detail.

The Ancient Era was in full bloom when Apple was born on April 1, 1976, marking the beginning of a new epoch. Contrary to the common stereotype, the company's early computers couldn't work with graphics at all and relied on a text-based interface. Apple's first computer with a graphical interface was the Lisa, introduced only in 1983—just a year before the release of the Macintosh, the subsequent fall of the Empire, and the decline of the Ancient Era.

However, the Middle Ages that followed were not as dark as some historians claim. Even though the interfaces of the Medieval period look outdated today, they were well-refined for their time. For example, designers of Mac OS and Windows learned to give buttons volume using two simple tricks: they added a dark border to the bottom and right edges, and a light border to the top and left edges. This was meant to show that the button protruded slightly toward the viewer from the surface, with light falling on it from the upper-left corner. When pressed, the light border turned dark, and in Mac OS, the background darkened as well.



Early Mac OS and Windows simulated top-left lighting by adding light and dark shading to button edges

These primitive button shadows were the first step toward fully developed skeuomorphism, which early designers struggled to achieve due to technical limitations. In the late 1990s, rapid advances in computer graphics began—advances that continued throughout the 2000s and pulled the graphical interface along with them.

Primary and Secondary Buttons

THE MIDDLE AGES SAW PLENTY OF INVENTIONS, though none could truly be called revolutionary. From 1984 to 2001, the computer industry seemed to be gathering strength, preparing to unleash its full potential in the Renaissance, which began with the release of Mac OS X.

Apple's new operating system introduced the glossy Aqua interface, built on OpenGL graphics. Its striking look became the system's main attraction. During the presentation, Steve Jobs said that one of the goals was to design something so appealing that users would want to lick the screen. And the new buttons really were beautiful. They resembled droplets of water glistening in the sun, hence the name Aqua.

Steve Jobs at the Mac OS X presentation, Jan 5, 2000

“One of the design goals was that when you saw it, you wanted to lick it”

There was another detail almost no one noticed. In the same presentation, Jobs drew attention to a new kind of button. On one of the screens, to the right of the familiar gray *Cancel* button, there was a blue *Apply* button that gently shifted its color, as if imitating a neon glow. Steve explained that the blue button was the primary button—the one you could activate with the Return key.

This was the first time interface elements were divided into multiple types according to their importance. Until then, all buttons were identical dull gray rectangles, but now one of them—the primary one—could be blue. This made it immediately clear which action was the main or recommended one.

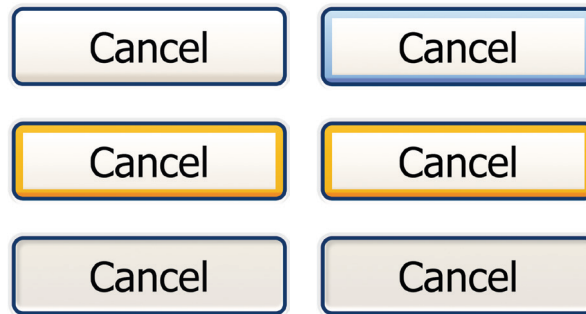
This separation of buttons into two types quickly became a staple of interface design. Today, it's hard to find an interface without these two types. Design systems typically call them *Primary* and *Secondary*, and often include additional types as well, which we'll examine later.

Secondary and Primary buttons in OS X 10.0



At the same time, Microsoft also decided to rethink its interface and, in that same year of 2001, released Windows XP. This system, too, introduced a distinction between primary and secondary buttons. Both types had a white background, but the primary button received a supplementary blue outline. In addition, the system brought its own innovation to interface design by adding

a third state for buttons. Now, a button not only darkened when pressed but also displayed an orange highlight when the mouse hovered over it.



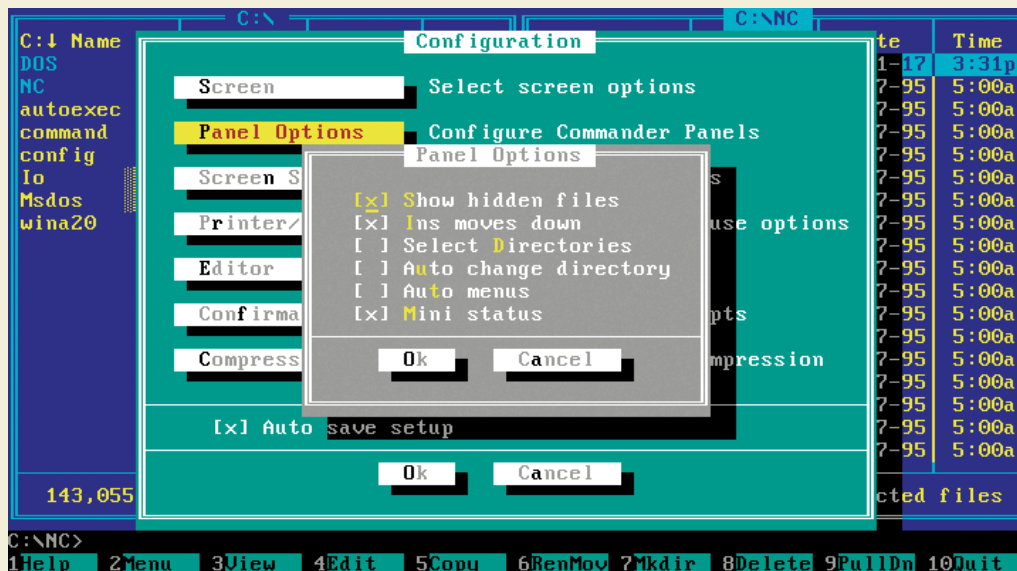
Windows XP buttons

Left: Secondary buttons
Right: Primary buttons

Top to bottom: Default,
Hover, Pressed states

Interestingly, the third state—what we now call *Hover*—never made its way into Mac OS. The system still uses only two states: default and pressed. Perhaps Apple’s designers have a point: after all, there is no such thing as “hovering” in the physical world, so why add it to virtual buttons?

Nevertheless, most modern interfaces do use a hover state, and I see no reason not to. A slight color change when the cursor hovers over a button adds a bit of charm to the interface, and the brief darkening at the moment of pressing creates a sense of depth, polish, and responsiveness.



Text Strikes Back

In parallel with the progress of the graphical interface, a strange mutant was evolving—the text interface. It combined elements of the command line with those of a full-fledged interface, drawing flat buttons, checkboxes, and even dropdown menus using special characters and solid-color fills. The most famous example of such an interface was Norton Commander, a file manager that gained worldwide popularity in the 1990s. It was often used by professionals accustomed to typing commands on the keyboard who didn’t find graphical interfaces particularly convenient.



The aesthetic-usability effect

This effect is so important that it can even compensate for logical flaws. For example, Facebook's interface—despite being quite inconvenient—still appears friendly and visually appealing. This is known as *the aesthetic-usability effect*. Some experiments have shown that, between two otherwise equivalent interfaces, the one that looks more attractive tends to feel more usable.

So what exactly is a hover state? In searching for an analogy in the physical world, I arrived at two interesting metaphors.

Imagine you're looking for the right button on a control panel in a poorly lit room. You're using a small flashlight with a narrow beam, pointing it at them one by one, and the button illuminated by the flashlight becomes brighter. This might well be the effect the designers of Windows XP were trying to imitate.

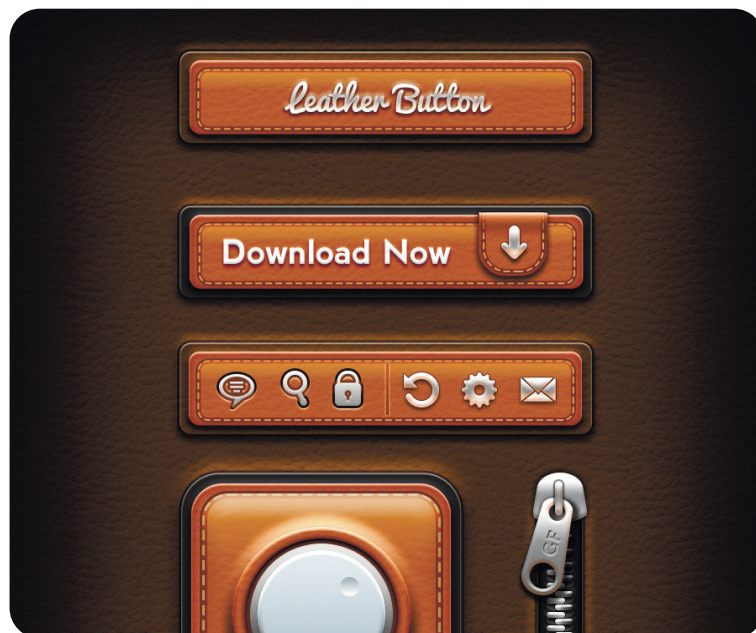
However, some design systems do the opposite, slightly darkening the button on hover. And that approach is justified, too. Imagine bringing your finger close to the button you're about to press. In that case, the finger casts a shadow on the button, making it a bit darker.

Which approach to choose is up to the designer. I have always liked the odd flashlight analogy and used to make buttons slightly brighter on hover, but while writing this book, I had to reconsider that approach. It turned out to be logically incorrect: buttons without fill simply can't be made brighter—they're already completely white.

Enter Photoshop

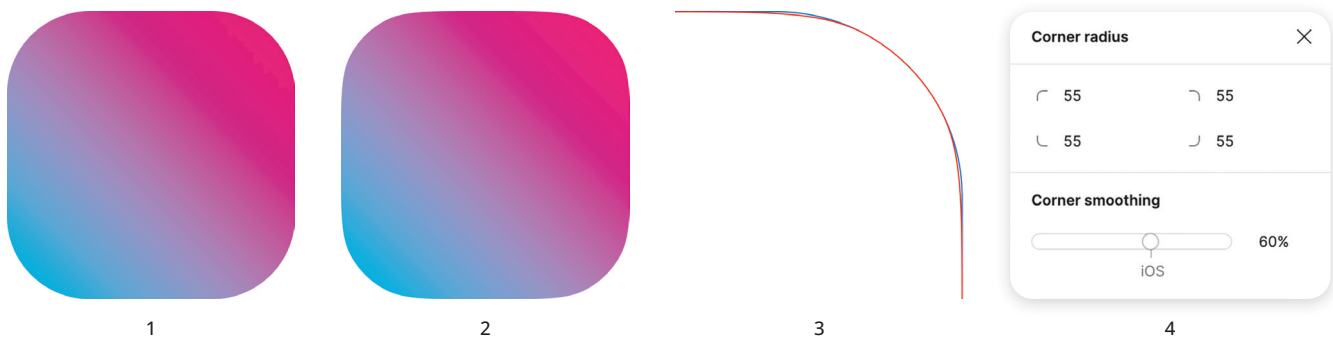
SKEUOMORPHISM EVOLVED RAPIDLY throughout the 2000s. Designers of that era seemed to compete to see who could stack the most shadows, gradients, textures, and animations onto their buttons without triggering an epileptic seizure in the user. A typical button of the time usually included: a drop shadow at the bottom, a glossy highlight at the top and a glow underneath it, a gradient fill, an outline, a texture or noise layer, and some additional effects applied to the label.

A leather-styled button from graphicsfuel.com—a typical example of peak skeuomorphism, 2012



The height of refinement was the contour shadow, a secret feature hidden in Photoshop. Instead of a standard linear shadow, you could create one that faded along a Bézier curve. A contour shadow was seen as the same kind of meticulous detail that a superelliptical border radius represents in iOS today. And, just like superellipses, it wasn't supported by any browser. However, this wasn't a problem back then, buttons were made using images.

Squircles use superelliptical corners (2) and hard to distinguish from rounded squares (1) yet a key part of iOS style. Figure 3 shows comparison; Figure 4 is the setting in Figma



The peak of skeuomorphism came in the late 2000s, beginning in 2007 with the release of the iPhone, the first phone operated with fingers. Many designers still remember the interfaces of the early iOS versions with a certain warmth: every app had its own unique design, polished down to the smallest detail. Notes looked like a real notepad; the camera showed a closing-aperture animation when taking a photo; and the books in the reading app sat on a wooden bookshelf.



Superellipse and squircle



Calendar app on iPad running iOS 6—the last version of the system to use skeuomorphism

Curiously, throughout all those years, designers continued drawing interfaces in Photoshop—a photo-editing tool completely unsuited for the job.

The first proper tool for interface design was Sketch, released in 2010, which gained some popularity by the end of the decade but never truly won designers'

hearts. Most likely, Sketch arrived too early. At the time, skeuomorphism was still popular, yet Sketch worked only with vector graphics. On top of that, the app was available only for Mac OS.

So designers, while fully aware of Sketch’s advantages, moved to it slowly and hesitantly, secretly launching Photoshop to add a texture or a dotted outline here and there. It would be several more years before the blessed Figma appeared, and almost a decade before it gained all the functionality designers needed.

Salvation came unexpectedly: skeuomorphism was killed by Apple itself.

UI Kits and Design Systems

THE ERA OF SKEUOMORPHISM ALSO SAW the rise of UI kits—collections of interface elements created in a consistent visual style.

Here’s how they worked. Suppose a designer needed to create the interface of an app, say a book reader. After preparing the initial concept, the next step was choosing an appropriate visual style and drawing the full interface, with all the details, fonts, colors, textures, and decorative elements.

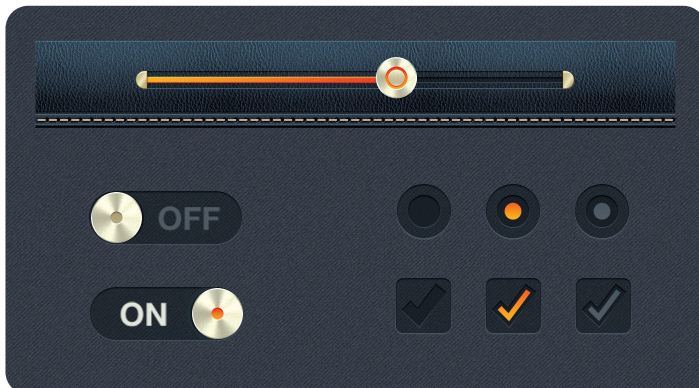
This part was always the hardest. There’s a reason creative people talk about the “fear of the blank page.” It happened like this: you opened Photoshop, stared at the white canvas, and had no idea where to begin. Even to draw the simplest button, you had to know at least what color it should be and what font the label should use. But to understand that, you first had to imagine the entire interface as a whole and only then zoom back down to the level of a single button.

UI kits were invented to solve exactly this problem. A designer would go to Dribbble or Envato, download a ready-made set of buttons—for example, in a wooden style (since books sit on wooden shelves)—then find a matching texture and font, draw the missing elements, and gradually assemble all the app’s screens.

Although UI kits were a useful source of inspiration, they were not very helpful for designing complex interfaces, where, in addition to the visual layer (UI), you also needed a deep understanding of the underlying logic (UX).

UI kits were far too primitive for large-scale tasks. Typically, they included a couple of simple buttons, a radio button, a checkbox, a toggle, and a range or a slider. That was enough to design a simple reading app, but nowhere near enough for designing a large application like Microsoft Excel.

Most UI kits were visually appealing but contained very few components and weren’t suitable for complex interfaces



There was another problem: skeuomorphism. Although many buttons really did look so delicious you almost wanted to lick the screen, this kind of styling seriously held design back.

A large project's interface consists of dozens of components, each with multiple styles and states. Take a simple button, for example. It can be filled or outlined, or rendered as plain text. You can add an icon to the left or right of the label, and you often need buttons in several colors: white, blue, or red (for destructive actions). And each of these variations may have at least four states: default, hovered, pressed, and disabled. Even the simplest button is extremely difficult to draw in all these styles, variants, and states when working in a skeuomorphic style. That's not to mention the dozens of other components! This is precisely why, in 2013, Apple killed skeuomorphism. By sacrificing visual beauty, the company paved the way for design systems—the next step in the evolution of interface design.

A design system can be viewed as an advanced form of a UI kit, containing dozens of components with numerous states, styles, and types. Components can be elementary—such as a button, checkbox, or input field—or composite, such as a calendar, form, or control panel. Composite components are built from elementary ones: a control panel, for instance, is nothing more than a row of buttons and other components arranged horizontally. This principle, known as atomic design, was introduced in 2013 by designer Brad Frost. A proper design system also includes a color library (for both light and dark themes), as well as typographic styles for headings and labels.

This book is dedicated to the search for a universal design system. After studying the history of user interface in detail and analyzing dozens of existing systems, I noticed that all of them are built on a set of universal principles. At the same time, every design system violates these principles in some way. Many chapters of this book are devoted to examining these principles and dissecting the mistakes found in today's design systems.



Atomic design

Photoshop Exits. Enter Figma

THE ERA OF SKEUOMORPHISM CAME TO AN END in 2013, when Jony Ive, Apple's design chief, introduced the flat iOS 7, followed year later by the flat Mac OS Yosemite. The design of these systems came to be called "flat" because of the complete absence of shadows, gradients, and textures: Ive removed even the faintest hints of skeuomorphism.

Users met the new interface with hostility. The early versions of flat design looked dull compared to the bright, lively interface of the old iOS. But over time, the design was polished, and it became quite appealing. Ive himself explained the demise of the old style with two reasons. First, he argued that skeuomorphism had been useful only in the early years of smartphones, when users were still getting used to the new technology. Second, he believed that unnecessary details distracted the eye, putting the style of apps above their function.

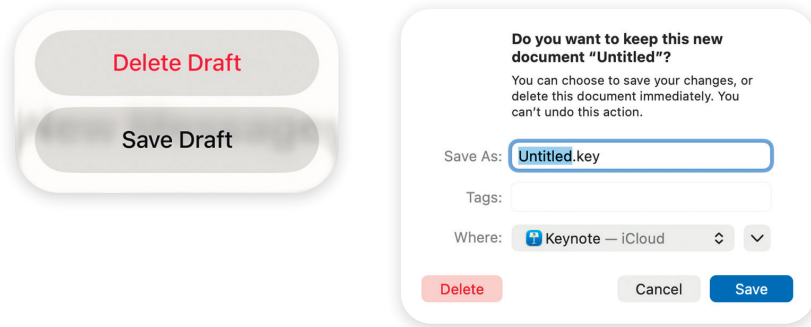
Despite all the missteps and the wave of criticism, Mr. Ive turned out to be largely right—and remarkably farsighted. Importantly, the end of skeuomorphism opened the way for the brilliant design tool Figma, which replaced Sketch and has since become beloved by all designers. But even more importantly, flat design paved the way for design systems, which by then had been nearly impractical

outside large corporations. As a result, the mid-2010s saw a kind of Cambrian explosion in interface design, producing an entire swarm of new buttons, radios, checkboxes, and dropdowns.

One of the first new buttons to appear in the flat Mac OS was the danger button. In fact, it had already existed in the early versions of iOS, but it didn't make its way to desktop until a decade later. The danger button is highlighted in red and typically performs a destructive action, such as deleting a file.

Following the danger button, appeared many variations of this component: outlined, borderless with hover feedback, borderless without hover feedback, and even plain text versions. One by one, we will examine all these styles and include them in our design system.

Danger buttons in modern iOS and macOS



Beyond purely stylistic changes, one of the major advances after the fall of skeuomorphism was the emergence of auxiliary elements, typically placed along the edges of components. A modern button consists not only of a base and a label; it may also include an icon to the left or right of the text, as well as a small supplementary label, often used as a counter.

It's not that none of this existed during the skeuomorphic era. Of course, there were danger buttons and buttons with icons. But creating an entire set of buttons with dozens of combinations was simply impossible. First, Photoshop couldn't work with component libraries: if you needed to update the design of a button across ten different pages, you had to open each file separately and replace the layers manually. Second, even if you solved that problem, maintaining a design system in the skeuomorphic style—with all its shadows, textures, and highlights—would have been extremely time-consuming.

Sketch was a significant step forward: it allowed designers to work more closely with libraries and component properties. But the true revolution came with Figma, which expanded this toolkit to an unprecedented level.

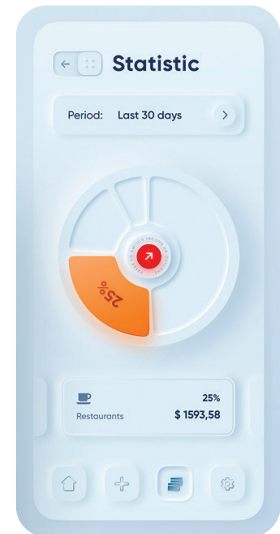
Neomorphism

THE NOSTALGIA FOR SKEUOMORPHISM lingered among designers for a long time. Many of us, even after recognizing the advantages of flat interfaces, could not fully accept the disappearance of shadows and gradients and therefore sought a compromise between the two incompatible styles. The most successful attempt was made by designer Alexander Plyuto, who introduced a visual language later called *neomorphism*.

The style emerged in 2019, when Plyuto presented a concept for a banking app on Dribbble. The unusual interface employed a soft color palette, elegant milky shadows, and a gentle glow in the upper-left corner of components. Its aesthetics resonated so strongly with the community that the concept went instantly viral. Over the following years, neomorphic interfaces proliferated at an incredible rate, winning the hearts—and portfolios—of designers, and many began predicting a bright future for the style.

At first, it seemed that neomorphism might take over the world, yet it soon became clear that the style was too demanding for large-scale adoption. Alas, it required a meticulous approach to color and an exceptionally delicate treatment of lighting. Beyond purely technical difficulty, neomorphism also proved to be incompatible with most corporate brand systems. Its aesthetics depended on a very specific palette and fell apart the moment one introduced even a slightly different shade.

Although many applications were eventually released in a neomorphic style, most of them were created by enthusiasts. Large companies tried to adopt the trend, but their efforts rarely went beyond tentative experiments in advertisements and design concepts. Unfortunately, neomorphism turned out to be not only too complex but also too beautiful: using an app turned into admiring the interface. Within a couple of years, the hype faded and soon disappeared entirely. Attempts to revive skeuomorphism, however, did not end.



The first neomorphic design, 2019

Liquid Glass

THE SOLUTION CAME, QUITE LITERALLY, from another dimension. A new wave of realism in interfaces was sparked by the Vision Pro—an AR headset—and, at the time of writing this book, that wave is only beginning to grow into a tsunami.

Augmented reality opened the door to spatial interfaces, in which virtual elements integrate seamlessly into the physical world. Such an interface does not lie on a two-dimensional screen but exists directly in front of the user's eyes, occupying actual space. A browser window can now hang on a wall, and a 3D model can stand on a table among ordinary objects. For virtual elements to truly settle into the real world, they must behave accordingly: refract light, exhibit highlights, cast shadows, respond to viewing angles, and so on.



Liquid glass was inspired by Vision Pro, yet it still uses a simpler effect

Apple's designers concluded that the best material for these tasks would be something resembling glass. Indeed, a glassy button appears natural, is easily noticeable thanks to light refraction and edge highlights, and yet doesn't block what's behind. This idea led to a new design language inspired by liquid glass.

Interestingly, although liquid glass originated in augmented reality, the Vision Pro headset itself uses a much simpler version of the effect than the one found in iOS 26. Perhaps we will see it in future iterations of the headset, and the puzzle will finally come together.

Immediately after the June 9, 2025 presentation, the public noted that glassy interfaces had existed before. In fact, the first operating system to use a glass effect was Windows Vista, released in 2007. Its style was called *Aero Glass*. Window titles, the Start button menu, and some applications, such as the media player, were made semi-transparent with a slight background blur. macOS also featured a glass effect long before it was replaced by liquid glass. It appeared in the Launchpad menus, the Dock, dropdown menus, and sidebar panels. The macOS version used much stronger background blur, referred to as *frosted glass*, or simply *blur*.

Glass effects: Aero Glass in Windows Vista, frosted glass in macOS 15, and liquid glass in iOS 26



Of course, liquid glass resembles all other glasses only at first glance. In reality, it is far more refined. Technologically, it is superb. Engineers even achieved dispersion along the edges of glass components, causing them to shimmer with color, and implemented a perfectly calibrated geometry of light distortion. This is not just a simple filter but genuine graphics card work and use of shaders, almost as in video games.

This complexity, however, is also the core problem of liquid glass. Unlike ordinary blur, such an effect cannot be produced without GPU rendering. On the web, it relies on WebGL, which is resource-intensive and poorly compatible with HTML. It will likely take quite some time before developers' favorite libraries—React and Vue—learn to simulate it. This means that iOS design will enter a state of real chaos: only native apps will be able to use liquid glass, while thousands of popular web apps will be stuck with outdated interfaces. The gap between platforms may become outright insurmountable, especially if Android introduces its own fancy-schmancy effect.

Liquid glass effects include magnification, dispersion, distortion, and surface tension



The third part of this book is devoted to augmented reality, examining the interfaces of AR headsets and glasses. As for liquid glass, this effect is primarily visual, and while the book does touch on visuals, its main focus lies on the internal logic and evolution of interfaces. Fortunately, a button remains a button even if we teach it to refract light and to wobble in response to a finger's glide.

The Anatomy of a Button

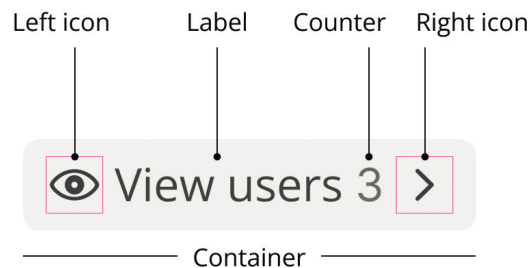
IT IS TIME TO MOVE FROM THEORY TO PRACTICE. Let's take a scalpel in our right hand and a pair of tweezers in our left, and dissect a button to understand its anatomical structure.

Through the entire body of the button—the part we'll call *container*—runs its backbone: *the label*. Typically, the button's width adjusts to the length of its label; the label itself never wraps to a second line and is kept as short as possible.

To the left of the label is an icon—a pictogram chosen to illustrate the action the button performs. There may also be an icon on the right of the label, such as a > right-pointing chevron, which is often used for a *Next* button. Finally, a small secondary label sometimes wedges itself in between the main label and the right-side icon—a small bit of text I've come to call *a counter*.



How to create a button in Figma: component properties



Naturally, you don't have to use all of these elements. The point is to create a universal button component in which these elements exist as optional parts. In design systems, such options are implemented as *component properties* that the designer can turn on or off as needed.

Container

ALTHOUGH SOME BUTTONS consist of text only, most have a colored background that visually separates them from their surroundings. This background—often called the body, base, or container—is usually a rounded rectangle. However, in Figma, the container is never created with the Rectangle tool. Instead, it's built as a *Frame* using *Auto Layout*, with all button elements—its label, icons, and the counter—placed inside it.

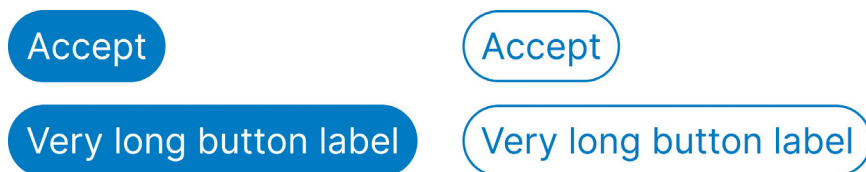
Because the width of a button must automatically adapt to the length of its label, you don't set a fixed pixel width. Instead, the container is given the *Hug contents* property. As for visual styling, the two most common container types are *Fill* (a solid background) and *Outline* (a border with a stroke).

Examples of simple Fill and Outline buttons. The button's width automatically adjusts to the length of its label



Perhaps the most debated parameter of the container shape is the degree of corner rounding. Until Snow Leopard, Mac OS used heavily rounded buttons shaped almost like pills or capsules. Apple then abandoned this approach and switched to rectangular buttons with rounded corners—only to come back to circular buttons in macOS Tahoe. If we wish, we can easily create a pill-shaped button in Figma by setting the corner radius to its maximum. Such a form works well for stylistic purposes, but in general it should be avoided: today, pill shapes are associated primarily with tags and badges.

Buttons with full-round corners aka *pill buttons*; this shape is better for tags than for buttons



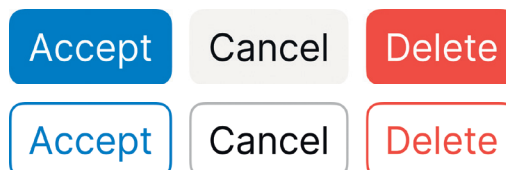
YouTube's pill buttons (top) resemble Medium's tags (bottom) and are distinguished mainly by their icons and size



The background color of a button can be anything. Most interfaces use three colors: blue for primary buttons, gray for secondary buttons, and red for danger buttons. In design systems, the button's color is rarely set directly. Instead, it's tied to the button's type and controlled by a *Type* property, which usually has three values: *Primary*, *Secondary*, and *Danger*.

Note that it's not just the background color that changes, but the label color as well. Obviously, black text is hard to read on a blue background, so primary and danger buttons with a filled background use white labels instead.

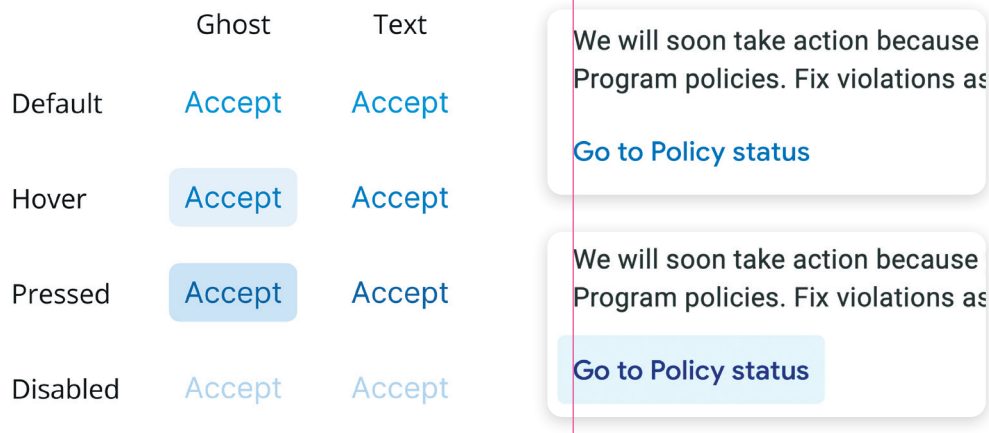
Button color is defined by its type: Primary is usually blue, Secondary is gray, and Danger is red



Two styles—Fill and Outline—aren't enough for complex interfaces, so most design systems include two more.

One of them is called *Ghost*. Sometimes this style is described as an empty, transparent, or invisible button. Of course, such buttons are perfectly visible; they just have no fill or outline, leaving only the label and icons. Still, a Ghost button lights up with a light background both on hover and on press. At that moment,

it becomes clear that the button does have a container and internal padding, but its color is by default transparent.

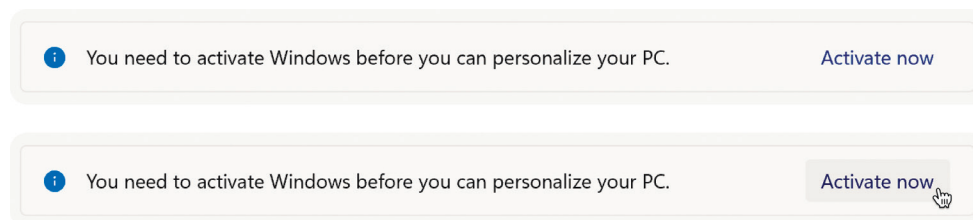


Left: Ghost buttons have a container and highlight on hover and press; text buttons have no container or padding

Right: An example of proper alignment for a Ghost button in Google Play Console. The button shifts left so that the label and text edges align

The second additional style is called *Text*. Unlike a Ghost button, a Text button does not light up with a background on hover or press. Only the label color changes, which makes it similar to a hyperlink. This is the style designers often—and incorrectly—label as *Link*. A Text button has no container and no internal padding.

Ghost buttons are common in Google products and other large platforms. They are often used in navigation and toolbars, as well as for secondary actions on sign-in screens.



Ghost buttons have a container and internal padding, which become visible through the highlight on hover

Text buttons are used for the same purposes. Often, a secondary action is so unimportant that you want to draw as little attention to it as possible. In such cases, a Text button is the perfect choice. It also has no internal padding, which helps save valuable space.



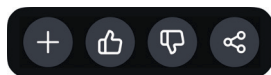
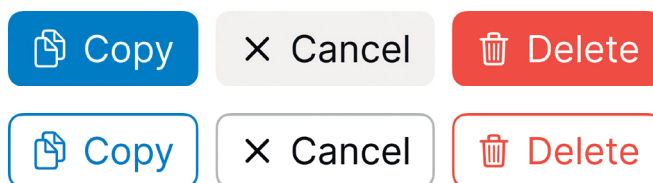
Text buttons have no container, are compact and unobtrusive

Left Icon

AN ICON IS OFTEN PLACED to the left of a button's label. Its purpose is mostly decorative. As it supports the label and draws attention, it shouldn't appear on every button. Typically, an icon is added to the button you want to emphasize.

If several buttons appear in a row, icons are often added to each to prevent the row from looking like a line of plain text.

An icon to the left of the label helps the button stand out from the others

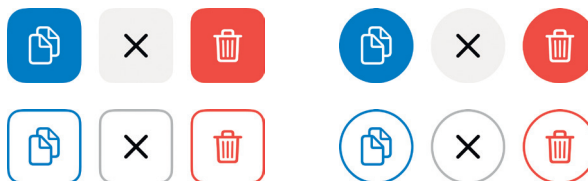


Round icon-only buttons in Amazon Prime Video and Facebook

Now that we've added an icon to the button, nothing stops us from removing the label entirely and ending up with an icon-only button. Such a button should be used with caution. The icon must be understandable to the user without a label or tooltip—something that's not easy to achieve.

The list of truly unambiguous icons is very short, containing no more than a dozen pictograms. For instance, an icon of a trash bin is clearly understood as *Delete*, and a gear will most likely open settings. However, a simple symbol such as a cross can mean anything: *Close*, *Delete*, *Hide*, or *Cancel*. The same issue applies to an icon of two stacked files: without context, it's impossible to tell whether it copies the selected file to the clipboard or creates a duplicate.

Use icon-only buttons with caution, as even the simplest pictograms can be interpreted differently



If we now fully round the corners of a square icon button—just as we did earlier with a rectangular one—we get a neat circular button. Unlike pill-shaped buttons, which are easy to confuse with tags or badges, circular icon buttons are a solid design choice and are used quite often, for example, by Google.

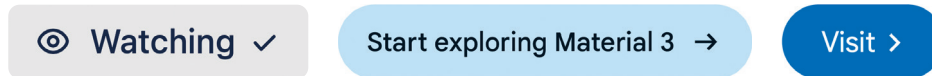
Right Icon

AN ICON DOESN'T HAVE to be on the left side of the label—it can appear on the right as well. Most often, this is a right-facing chevron, which pairs nicely with a gray button labeled *Next*. For variety, let's experiment with a different icon here, such as the arrow for opening a link in a new window. I haven't seen similar examples for a red button, but we can easily imagine a *Continue* button with an exclamation mark icon.

An icon on the right helps clarify the button's label



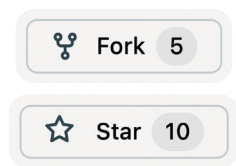
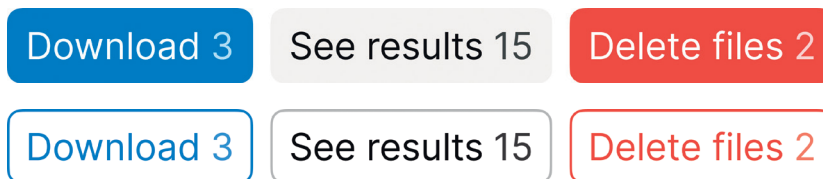
The difference between a left and right icon is subtle. An icon on the left reinforces the label. For example, if a button says *Download*, the ☁ cloud icon goes on the left because it depicts the download action itself, acting as an illustration. An icon on the right, however, *clarifies* the label. This is easy to see with a button labeled *Open*: the ↗ arrow-in-a-square icon doesn't illustrate the very act of opening—it clarifies that the link will open in a new window.



Buttons with right icons: Trello, Google Material Design, Google Images

Counter

THE FINAL ELEMENT of a button is the counter. This small secondary label sits between the label and the right icon. I call it a counter because many buttons genuinely count things. For example, a *Download files* button may show how many files will be downloaded; a *Delete* button with a counter helps indicate how many items will be removed; and so on.



GitHub's *Fork* and *Star* buttons show how many users have already performed the action

Of course, the counter can be used for other purposes as well. In the following chapters, we'll see that this small secondary label can serve more than just counting.

Button States

FROM OUR HISTORICAL OVERVIEW, we learned that buttons react to user actions: they highlight on hover and darken on press, mimicking the physical world. To account for this behavior, every component is created in several states. In Figma, this is done using *Variants*—different versions of the same component.

A component's states are implemented as a property, usually named *State*, that can be switched just like types and styles. The default state is called *Default*, and the hovered state is called *Hover*. Designers often call the pressed state *Active*, but that's inaccurate: a simple button doesn't activate; it gets pressed for a fraction of a second, which is why I prefer the term *Pressed*.

The fourth state is *Disabled*, used for inactive buttons. Strictly speaking, it isn't a state in the same sense. We call a state the button's reaction to user actions, but a disabled button is disabled not by user interaction, but by the developer.

Debates over the usefulness of the disabled state have been ongoing since its emergence in the 1980s. At first glance, disabling a button seems pointless: if it shouldn't be used, why not just hide it? The problem is that a button may be

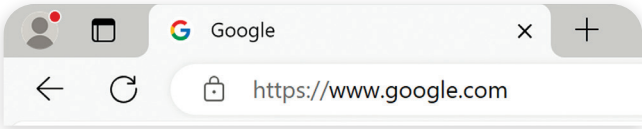
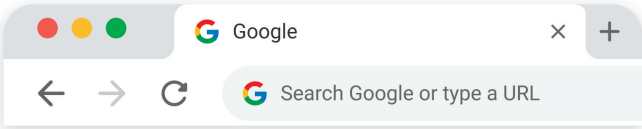


Component variants in Figma



UX questions:
hide or disable?

Even the → *Forward*
button in browsers can be
either disabled (Chrome)
or simply hidden (Edge)



needed in general but undesirable in certain cases. For example, a *Delete* button may be available only for user files, not for system ones. If the button is always hidden, a new user who clicks through a few files and never sees a delete option may conclude that files simply cannot be deleted at all. To avoid this confusion, buttons are usually shown but made semi-transparent and unpressable.

There is also a third approach. Its supporters argue that a button should always be visible and clickable, and the system should show an error when the action isn't allowed. A typical example is a form where the *Submit* button becomes available only after all fields are filled in. In this case, it's indeed better to allow the user to click the button even when the form is incomplete—and then highlight the empty fields in red. But this approach clearly wouldn't work for something like a browser's *Forward* button, so it's not universal.

The choice between the two main options—disabling or hiding a button—comes down to this: if the action is unavailable temporarily or in a specific situation, the button should be disabled. If the action is permanently unavailable due to a constant restriction, the button shouldn't be shown at all. For example, if a user has turned on *Do not disturb* mode, that's a temporary condition, so the call button beside their name should appear disabled. If, however, calls are forbidden by the system administrator, that's a permanent restriction, and the button shouldn't be shown at all.

So we have four button states: *Default*, *Hover*, *Pressed*, and *Disabled*. As agreed, the button will become slightly darker on hover and significantly darker on press. For the disabled state, we'll use a semi-transparent background and text.

Button states, from left
to right: Default, Hover,
Pressed, Disabled



Focus Mode

DESIGNERS ALMOST ALWAYS mistakenly include a fifth state in a design system called *Focus*. It's used for a button that currently has keyboard focus. Such a button gets an additional outline: in macOS, it appears as a thick blue border, and in Windows as a thin dotted border. This state indicates that the button can be pressed from the keyboard.

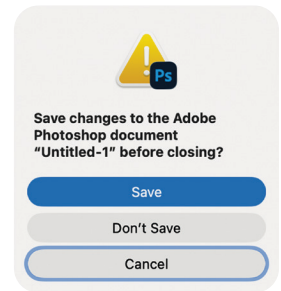
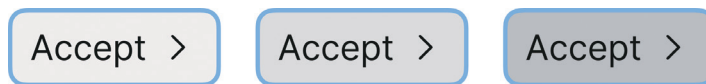
Let's be honest: this isn't a particularly important feature of modern design. Triggering buttons from the keyboard was popular in the 1990s, when something was always breaking in Windows 98, and a wireless mouse could stop working

after a driver update. In such cases, you had to roll back the driver while navigating through system settings purely with the keyboard.

Today, such situations are extremely rare. For example, in macOS, keyboard navigation across buttons is disabled by default and only works for text fields. And even if you enable it, the behavior is odd: a focused button is triggered only with the *Space bar*, while *Return* always triggers the blue Primary button—even when that button isn't in focus.

It seems that designers long ago gave up on this state, and not without reason. First, almost no one actually uses keyboard navigation anymore. Unless you're designing a system-level, fault-tolerant interface, you can safely omit this state for components other than text fields. As for users with limited hand mobility, most of them use touchscreens or accessibility plugins. Second, if keyboard navigation is enabled, the operating system will add a focus outline automatically, assuming, of course, that the element is properly marked up as a button.

There's another problem: focus isn't actually a state—it's *a mode*. The difference is that states are mutually exclusive: a button can only be in one state at any given time. But a button in any of these state—Default, Hover, or Pressed—can still be in focus at the same time.



Operating systems today don't pay much attention to keyboard navigation. Here, pressing *Return* triggers the *Save* button, even though the *Don't Save* button is in focus

Focused buttons can be in any state, which means *Focus* is a *mode*

This is one example of the logical confusion that has taken root in many designers' minds, including professionals. In every design system I'm aware of, *Focus* is implemented as a state alongside *Hover* and *Pressed*, even though it would be more accurate to treat it as a separate property.

For all the reasons mentioned above, we'll include the focus mode in our design system only for compatibility and won't spend much time on it. It will be implemented as a layer with a blue outline and controlled by a separate binary property with a toggle.

Sizes

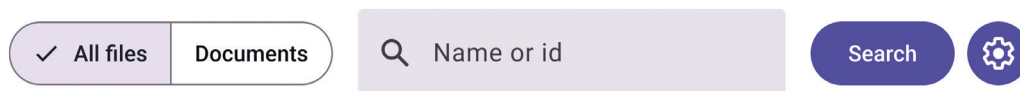
MANY DESIGN SYSTEMS, INCLUDING WINDOWS and macOS, use only a single component size. Others, such as Bootstrap and Ant Design, use three: *Small*, *Medium*, and *Large*. Since a component's width usually adapts to its contents, when we talk about sizes here, we mean only height. Unlike width, height is set as a specific value in pixels using the *Fixed* property in Figma.

I haven't found any consistent logic for choosing button sizes that could serve as a general rule. Every design system uses its own values, and all of them are arbitrary. Based on my own taste and experience, the most reasonable sizes are 28, 36, and 44 pixels.



You may choose different sizes if you like. The important thing is that all components share the same height, since they may appear in a single row. This rule seems obvious, yet it's often broken. For example, in Material Design, the text field is 56 pixels tall, while the button and other components are only 40 pixels tall. As a result, the text field sticks out of an otherwise neat row of buttons in a toolbar.

The Material Design text field has greater height than other components



When building our design system, we'll avoid making such obvious mistakes. We'll also apply visual compensation: larger buttons require larger internal padding and a stronger corner radius in order to look balanced.

Component Matrix: Bringing It All Together

SO, WE HAVE DISCUSSED ALL the parameters of a button. Now our task is to bring them together and define the button as a component with a set of properties.

It's important to understand the following. This entire set of types, styles, sizes, and states produces a huge number of combinations—144 variants of the same button, to be exact. Such a large number of variants can't be represented as a simple list, which is why design systems store components as tables where styles and states are labeled vertically, and sizes and types are labeled horizontally.

To start, we'll try this approach with a single size and a single style. If we list the button states vertically and the types horizontally, we get a 3×4 table with 12 buttons.

The combinations of component properties form a table in which the columns correspond to types, styles, and variants; and the rows correspond to states

	Primary	Secondary	Danger
Default	Accept >	Cancel >	Delete >
Hover	Accept >	Cancel >	Delete >
Pressed	Accept >	Cancel >	Delete >
Disabled	Accept >	Cancel >	Delete >

I like to refer to such a table as a matrix of variants and states, or simply as a *component matrix*.

The example above shows the simplest case: a matrix for a single button size. To construct the full matrix of all combinations, we replicate this table, placing three copies horizontally and four vertically. For clarity, we group the buttons by color. Horizontally, we label the groups by size; vertically, by style. The result is a large component matrix consisting of 144 button variants.

Primary

Secondary

Danger

Small

Medium

Large

Small

Medium

Large

Small

Medium

Large

Fill

Default

Hover

Pressed

Disabled

Outline

Default

Hover

Pressed

Disabled

Ghost

Default

Hover

Pressed

Disabled

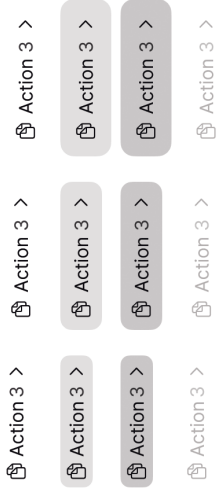
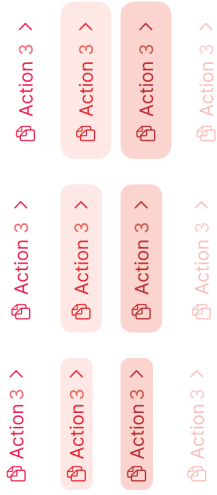
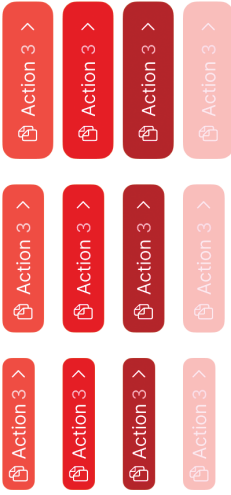
Text

Default

Hover

Pressed

Disabled



This is exactly what a component looks like in a professional design system. Buttons, text fields, dropdowns, and other components can have as many styles and types as needed; they may even treat focus mode as a state or have other minor mistakes. But the key distinction between a professional design system and an amateur one is the *completeness of combinations*, which can be achieved only by representing the components as a matrix.

I have seen many design systems that ignore this rule and present components linearly. Even though large corporate products are built on top of them, working with such systems is often extremely difficult. You could never find a button in the exact style you needed, so you had to detach the component and make edits manually. An incomplete design system is like a cheap screwdriver set: it works fine for standard screws, but the moment you need to take apart a laptop, the Torx bit is always missing.

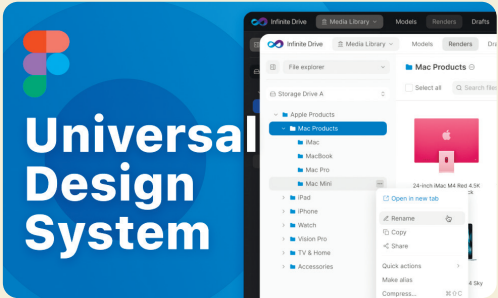
I firmly believe that a good designer, like a good engineer, should always have a complete set of tools and never remove a screw with manicure scissors.



Universal Design System

CONGRATULATIONS, DEAR READER. We’ve just created our first component—the universal button (*Latin: Buttonis universalis*).

Take a look at the spread where all possible button variations barely fit on a single page. I’ve shown the full version without any shortcuts, even though modern Figma lets you use binary properties to show and hide layers inside a component without creating separate variants. The feature is extremely useful, but it obscures the component’s underlying logic and beauty. What designer would give up such elegance?



Universal Design System

A design system inspired by this book is available on Figma Community. You can use it not only as a supplemental material, but also as a foundation for your own projects.