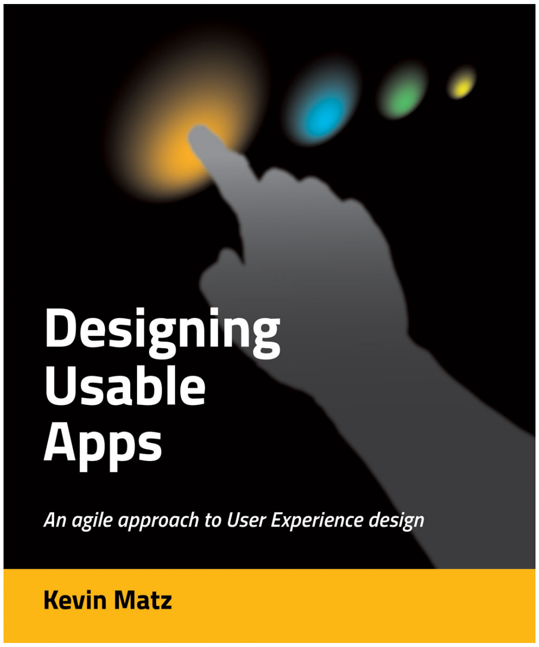
This is a free sample excerpt from the book:

Designing Usable Apps

An agile approach to User Experience design



Author: **Kevin Matz**
264 pages (softcover edition)
Print edition ISBN: 978-0-9869109-0-6
E-book edition ISBN: 978-0-9869109-1-3

www.designingusableapps.com

Available in print and e-book formats at Amazon.com and other booksellers

WinchelseaPress

9

The fundamentals of human-computer interaction

To design a software application that is easy to learn and use, it is helpful to understand the basic psychology of human-computer interaction. In this chapter, we'll explore some of the key concepts and learn the practical implications for design.

How users get things done with a software application

Users interact with software by performing physical actions with input devices such as keyboards, mice, touchscreens, and microphones. Graphical user interfaces present controls like buttons, sliders, and drop-down boxes, and the user performs actions on these controls, either directly by gesturing on a touchscreen, or indirectly via mouse clicks or keyboard keystrokes.

Non-graphical interfaces typically rely on the user issuing commands to perform actions, whether by typing them in at a command line, or via spoken commands in a voice-activated system.

But how do users know which actions to perform to get their work done?

The usual model for thinking about this involves a hierarchical breakdown of work into *goals*, *tasks*, and *actions*:

- A user usually has a high-level **goal** in mind of what he or she wishes to accomplish with the application. This might be something like writing a letter, retouching a photograph, conducting a video chat with a coworker, paying a credit card bill, or comparing prices for flights. Goals are statements about *what* the user wants to achieve, rather than *how* it will be achieved.
- To accomplish a goal, the user usually has to perform some number of general steps or structured activities that we could call **tasks**.
- To perform a task, the user will perform *actions* in the interface. **Actions** are specific operations involving the user interface, such as pressing or clicking on a button, entering text, selecting an item from a menu, dragging-and-dropping an icon, and so on.

Let's imagine that a user of a word processor has the *goal* of writing and printing out a letter. This *goal* might be achieved with some combination of the following general tasks:

- Creating a new document
- Entering text
- Editing and proofreading text
- Spell-checking
- Adjusting page formatting
- Previewing
- Printing

To accomplish the task of creating a new document, the user might perform the following series of *actions* in the interface:

- Click on the "File" pull-down menu
- Click on the "New" menu option
- Enter a document title in a dialog box
- Click on the "OK" button to close the dialog box

It's important to understand that goals can often be achieved by means of various different sets of tasks, and tasks can often be achieved by means of various different sets of actions. So for the task of creating a new document, alternatively, the user might have used a shortcut keystroke such as Alt-N, or perhaps the user might have opened an existing document and re-saved it with a different filename.

And while there may be some cases where tasks can be achieved by following a strict step-by-step sequence of actions, in many cases, such as entering and editing text in a word processor, tasks are more of an ongoing or iterative process, and multiple tasks might become intermixed with each other as work is done towards reaching the goal.

The action cycle

An experienced user will usually know what tasks are needed to accomplish a goal, and can figure out what actions are needed to accomplish each task. New users learning how to use an application, on the other hand, are usually uncertain about what actions can be done to accomplish a task, and may even be uncertain about what tasks are necessary to achieve their goal.

Some users will seek out documentation or online help resources to find instructions on how to use the product. However, most users will begin by taking an exploratory approach.

When first trying to accomplish a task, a user will typically explore and inspect the interface for clues. Once the user has identified a potential action that may help move the user along the path to accomplishing the task and achieving the goal, the user will execute that action, and then observe what happens. If the results of the action — the feedback — matched what was expected, then the user will continue on with the next step in completing the task. If not, then the user may try an alternative action, or the decision might be made to modify the task or the goal.

A user will generally continue this cycle of searching for suitable actions, choosing actions, performing actions, and evaluating the results, until the goal has been satisfactorily achieved, or until the user gets stuck and needs assistance to continue.

Donald Norman elaborated on this process more formally in his book *The Design of Everyday Things* (Norman, 1990), describing it as the **seven-stage action cycle model**, which consists of the following steps:

1. Identifying an immediate goal
2. Forming an intention to act

3. Determining a plan of specific actions
4. Carrying out the actions
5. Observing the results by perceiving the state of the system and the world
6. Interpreting the results
7. Evaluating whether the actions had the desired results

These steps are repeated in an ongoing cycle — the evaluation of the effects of the actions informs the selection of the next goal — and so this model describes human-computer interaction as an continuous feedback loop between the user and the machine.

Mental models

As your users learn how to perform tasks with your application, they gradually form a *mental model* of how it works and how to operate it. A **mental model** is a conceptual representation in a user's mind of how a system works, and how to operate its interface. A user's mental model reflects the user's *current* understanding, and that understanding is subject to change as the user gains experience with the product, or forgets details over time.

When faced with a new situation, users rely on their mental models to reason about the situation and the system, and to make decisions and formulate strategies on how to proceed. Users will also form expectations for the application's behavior based on their mental models.

But mental models are not always *correct* representations of how a system works and behaves, and the mismatch between a user's incorrect mental model and the system's actual **implementation model** can explain many usability problems. It's important, therefore, for a system to be designed in such a way as to help users form a correct mental model of the system's operation.

What makes up a mental model?

Mental models are cognitive structures in peoples' minds. It's hard to say that mental models have any particular form or structure. A mental model is not inherently visual, although visual images do form an important part of a mental model.

A mental model for a software-based system consists of the following elements.

General appearance

A user will mentally form visual images of the "places" (screens, pages, tabs, windows, etc.) of the system that the user has encountered and is familiar with. But these mental images are typically very vague and imperfect; most users will not have photographic recall.

For a typical complex software application, users will become familiar with the general layout of the places they encounter frequently. The level of detail of mental images will vary depending on each user and the frequency of use.

For example, as a frequent user of Microsoft Word, I have a vague image of the layout of the main window in my mind, though without looking, I wouldn't be able to recall the exact sequence of icons in the toolbar or even what precise sequence of pull-down menus exists after *File* and *Edit*. I know some of the dialogs like *Font* and *Find/Replace* well enough that, even if the text of the labels and buttons were blurred, I could still recognize the dialogs by the "shapes" of their layouts. But my recall is not good enough to be able to sketch them out accurately.

Concepts, vocabulary, and rules

As we saw in previous chapters, every software-based system or product solves some sort of problem (though it may be a trivial problem, such as keeping the user entertained, in the case of a game).

The concepts, vocabulary, and rules involved in the context of that problem are referred to as the **problem domain**, the **business domain**, or the **application domain**.

For some systems, the problem domain is relatively small. The operator of an e-mail client only needs to understand a handful of concepts, like e-mail addresses and attachments. Other systems will demand much more in-depth knowledge and understanding of a domain. Imagine what a master operator in the control room of a nuclear power plant needs to know!

Some applications, like the nuclear power control system, must be designed with the assumption that the users already have the prerequisite knowledge of the domain, whether through education, training, experience, or some combination of these.

Other applications have the responsibility of communicating their unique concepts to the user. Games, being imaginary worlds rather than real-world problem-solving tools, are an extreme example of this. The first time you play, say, *Angry Birds*, you need to learn what objects are in the game and how they interact (in other words, the basic rules of the game).

Or, take Twitter as another example. To use Twitter, you need to understand what a "tweet" is, and you need to learn that you can follow other users and that other users can follow you. If you had never heard of and never used Twitter before, you'd most likely explore the Twitter website or app and figure out how it works via experimental self-discovery. In the somewhat unique case of something as popular as Twitter, the chances are that you might learn the concepts second-hand by watching a friend use it or by hearing about it in the media.

In many cases, users can grasp concepts without explicitly knowing the associated vocabulary and terminology. For example, web browser users can enter website addresses without knowing that the addresses are technically called URLs (*uniform resource locators*).

Additionally, users often don't need a full understanding of many concepts if the software handles the appropriate details for them. Users of a shipping postage calculator may only need of know *of* a customs duty fee; they do not need to know its precise rules and regulations, as they will trust the software to calculate the fee for them.

In many cases, users who aren't aware of all of the application's concepts, or don't understand them completely, are still often able to use the application effectively, if not optimally. Virtually all beginning users of Microsoft Word are unaware of the concept of styles, for instance, but they are still quite capable of producing documents.

Upcoming chapters on design principles will give us insights on how to structure applications to help users discover and learn key concepts.

Navigation map

Many applications consist of places (screens, pages, tabs, windows, etc.), which the user can "visit". When it is necessary for the user to differentiate between these places and to be able to get to them quickly, the user will gradually form a mental **navigation map** indicating how to get to the different destinations.

Navigation is often one of the actions needed to carry out a task in an application. For example, to purchase goods on an e-commerce site, you may need to navigate to the shopping cart page and then click on a *Checkout* button, which leads to a sequence of pages for finalizing the purchase.

Sometimes there may be more than one way to get to a location. For example, to get to the *Print* dialog box in most Windows applications, you can navigate to the *File* menu and choose *Print…*, or you can use the shortcut Ctrl-P, or you can click on the printer icon in the toolbar. The user may not be aware of all ways to navigate to a destination, and users aware of multiple options will tend to use only one of them frequently.

Action plans or strategies for accomplishing tasks or for reacting to situations or problems

Users may memorize *plans of actions* needed for carrying out certain tasks.

An **action plan** might take the form of a simple sequence of steps to follow. Or, with sufficient experience with the product, users may internalize a conceptual structure similar to a flowchart diagram that has various decision points and branches with steps to follow under different circumstances. (But note that most users will not actually have a literal visual depiction of a flowchart in their mind, and keep in mind again that the structure may not necessarily be complete or correct.)

Sometimes a user may not necessarily understand why a certain sequence of actions performs a particular task, but they've still memorized the sequence and are able to reproduce it. This can happen when the user has been taught how to perform a task in a training session, but some of the fundamental concepts (the "why" behind the actions) haven't been explained. It can also happen when the user has discovered by accident how to perform a task.

General heuristics and conventions

The user's mental model may include general **heuristics** — rule-of-thumb guidelines learned from experience — and conventions from a broader context that can be applied to the system at hand. For example, based on the user's experience with the operating system, one such heuristic might be, "to dismiss a dialog box, click on the *OK* button or click on the 'X' in the title bar".

Perceived implementation model

In some cases, a user may begin forming a general conception of how the product works internally, at some basic level, though this is never guaranteed.

For simple mechanical devices and machines, you are often able to see all of the moving parts, and you can mentally envision how the parts interact when the device is in operation. If you examine and manipulate a manual can opener, for instance, you can see how the edge of the can is pinched between the wheel and the blade, and you can imagine how turning the handle slices open the can.

For simple mechanical devices, seeing and understanding how the parts work can be helpful and may even be necessary for operating the device correctly. But for more complex mechanical devices, like the engine of an automobile, the inner workings are often too complicated for non-engineers to understand — and so such machinery is tucked out of sight. Automobile operators are offered simplified, abstract controls — like the gas pedal and the automatic gearshift — which eliminate the need to know how the engine works. In other words, the user's mental model of the underlying implementation can be extraordinarily simple (basically: "the engine consumes gas to run, so I need to make sure there's still enough gas in the tank"). The user's mental model can instead focus on the actions needed to make the car move: put the gearshift into *Drive* and depress the gas pedal.

For software products, designers need to hide the internal workings to the maximum extent possible. While some technically-sophisticated power users might try be able to reverse-engineer how the underlying algorithms and data storage schemes and communications protocols work, users should never have to know about such technical implementation details.

But even if users are perfectly shielded from unnecessary technical implementation details, they will still often be able to observe patterns in how the system operates and responds to inputs. From these observations and patterns, users will form simple implementation models, and implementation models on this level of abstraction are a good thing.

Let's say your application has an on-screen table containing a list of contacts, and there is an *Add* button to let the user add a contact to the list. The user observes that every time a new contact is added, it appears at the bottom of the list, below the other entries. Based on this observation, the user will tend to presume that the contacts are maintained in a sequential list, and new contacts are always simply added to the end of the list (rather than being added at the top of the list or being inserted at appropriate places in order to maintain alphabetical ordering or some other sort order).

This is a very simple and abstract form of implementation model, but it helps the user predict what will happen when the action of adding a new contact is performed, and when users encounter another similar-looking table, they will typically assume that the same behavior applies there as well.

Communicating an intended mental model to users

As a user interface designer, you'll have your own conceptual mental model in your mind of how the application will function. In order for users to be able to operate the application effectively, they will eventually have to have similar mental models in their minds.

One way of building up a mental model in a user's head is to provide a structured instructional curriculum that explains the product's concepts and operation. This may take the form of documentation or training. For most products, however, the vast majority of users will not read the documentation, and training, when available, is not always pedagogically effective.

Some users have the benefit of being able to watch other users use the application, and this can be a very effective way of learning the basic concepts and developing an understanding of how to perform tasks. Having an expert nearby whom the user can ask for assistance is also very helpful.

But without any training, documentation, or opportunities to watch and ask other users, the only way a user can figure out how to use the application is to simply start using it, and learn via trial-and-error.

The visual presentation of the application's user interface provides cues as to how to accomplish actions and tasks, and the behavior of the application provides feedback on whether the actions and tasks are having the intended effect. And so by continually exploring and experimenting with the application, the user will gradually build up a mental model of the application's functionality. With time and experience, it is hoped that the user's mental model will increasingly approximate the designer's conceptual model.

To use the terminology popularized by Donald Norman in *The Design of Everyday Things*, the conceptual model in the designer's mind is called the **design model**. The user's mental model is simply referred to as the **user's model**. And the visual presentation and the behavior that the product's user interface exhibits is what is called the **system image**.

And so to design a usable and learnable product, then, the designer's challenge can be viewed as *aligning the design model and the system image* in such a way that the system image accurately portrays the design model and enables users to develop their own users' models that approximate the design model as closely as possible.

As the completeness and correctness of a user's mental model increases, that user's skill at operating the application should very gradually approach that of the application's designer.

Structuring the system image to make an application learnable and understandable is tricky, and the remainder of this book concentrates on exploring how to do this by means of understanding psychological principles, design principles, design techniques, and usability testing and evaluation techniques.

Human memory

Learning to use a product involves learning and memorization, and operating a product often relies on the user keeping the context of the situation in short-term memory.

Human memory is complex and a little mysterious, and unlike electronic data storage, it is not perfectly reliable and predictable. In this section, we'll take a whirlwind tour of human memory, and then apply this knowledge to user interface design.

A model of memory

There are a number of psychological models of human memory. Most distinguish between short-term and long-term memory as separate but interrelated structures or systems in the brain. While there is no consensus on the "correct" model, one model useful for us is as follows:

- **Short-term memory** or **working memory** is a temporary store that can hold a small amount of information, such as a handful of words, numbers, or symbols, related to your current train of thought. Working memory decays very rapidly; the information can be lost when your attention is drawn to something else, and so you often have to rehearse or repeat the information to yourself to avoid having it disappear. The capacity of working memory is said to be about "seven, plus or minus two" items (Miller, 1956), and it's for this reason that North American phone numbers were chosen to be seven digits long — it's difficult to hold more than about seven digits in your mind when you hear a phone number and you're trying to write it down.
- **Middle-term memory** or **contextual memory** holds the information you need to be able to work on your current activity, but you won't permanently remember most of this information. For example, during a conversation, you'll have in mind the details of what has been discussed so far. Or if you're working on your tax return,

you'll know where on your desk you've put your different papers and receipts and you'll remember some of the key numbers and details.

- **Long-term memory** is a more persistent store of knowledge and memories of experiences — facts, concepts, ideas, names, images, sounds, voices, places, emotional feeling states, and so on. Long-term memory also stores procedures and skills, for both cognitive and sensory-motor tasks. Long-term memory might alternatively be called **permanent memory**, but this is misleading as information is often subject to forgetting or "false recall".

How does memorization happen?

Memorization, the act of intentionally committing something from short-term memory to long-term memory, usually happens through repetition. Generally, the more often you encounter (see or hear) something, the more likely you are to remember it later. Studying involves actively and intentionally re-reading, rehearsing, and practicing.

But we also tend to remember information and experiences that are surprising, novel, important, or unusual without any repetition.

The exact nature of how the brain forms memories remains unknown, but it is likely that information and memories stored in long-term memory are somehow stored symbolically. That is, if you hear a professor telling you information in a lecture, you may memorize some of the information in the lecture, but you will probably not store a perfect audio recording of the professor's lecture. You may of course remember the professor's voice, especially if it is particularly unique, but this is separate from the information content of the lecture, which you can make use of in practical contexts without "playing back" the "audio recording" of the professor's voice.

There are some rare people who do have a perfect photographic memory, though, and most people can remember music precisely enough that they can distinguish if a later performance differs by only one note.

We tend to store information in logical groupings which psychologists call **chunks**. Memorization is most effective when a chunk is associated with other existing chunks of information in memory. **Associations** are logical connections or relationships between pieces of information. If you meet and get to know a new person, for instance, you'll associate the image of their face with their name and their other personal details you might learn, like their occupation and family members.

If you are trying to learn a complex concept or process, and you feel that your understanding is incomplete or insufficient because of unanswered questions, memorization and later recall will tend not to be as reliable as when you feel that you have a complete and logical understanding of the matter.

Recall and recognition

Recall of information from long-term memory is usually triggered by some **cue** or **prompt**. Seeing someone's face, for instance, can trigger you to recall that person's name.

Successful recall is never guaranteed. The more recently the information was memorized or accessed, though, the more likely you are to be able to recall it (the **recency effect**).

Successful recall of some piece of information is also more likely to occur when you've already recalled related information. It's as if related information is stored in adjacent locations in the brain, and by accessing information in a particular region, you "light up" that region, and then recalling other related information from that region becomes easier.

Sometimes you will struggle to recall something, and the information may or may not come to you at a later time. Sometimes recall is inaccurate; you recall incorrect information. You might misremember a formula when you're taking a math test, for instance. Sometimes you may have doubt about whether the recalled information is correct, but just as frequently, you may not recognize the error.

Often you may not be able to recall something, but you can **recognize** it when you see it. The information was in your memory, but for some reason it was "shrouded" and didn't lend itself to being accessed at that moment.

Forgetting

The less frequently a chunk of information or a skill is accessed from long-term memory, the more likely it is to be forgotten. This is natural — things that are relevant to your daily routines will be remembered, and additionally, they will be continually reinforced due to the recency effect. On the other hand, facts that you studied years ago but haven't needed, or the names of people whom you met years ago but haven't kept in contact with since, will tend to fade away.

But there are also many cases where letting information or skills languish for long periods of time won't necessarily guarantee that they will be forgotten. Highly-developed motor and cognitive skills that can be done unconsciously after much practice — like riding a bicycle or speaking a foreign language — can often still be performed with surprising levels of competence even after years of neglect.

Applying knowledge of human memory to user interface design

On the basis of this understanding of memory, memorization, recall, and forgetting, here are some guidelines to keep in mind when designing software:

- Structure your interface to reduce or eliminate the need to memorize and recall things. Donald Norman discusses the notion of "knowledge in the world" versus "knowledge in the head". For example, presenting a list of options in a menu is an example of "knowledge in the world": the user can view the menu, read and recognize the options, and make a selection without needing to memorize or recall anything. If you were to require the user to enter commands at a command-line interface, on the other hand, this would require the user to memorize and recall the commands, thereby requiring the user to store that knowledge "in the head", and making subsequent recall potentially subject to errors and forgetting.
- If a task has a defined sequence of steps, guide the user through the task flow step-by-step by presenting forms and controls in a logical, sequential order. If appropriate, consider offering a **wizard**-style interface, with multiple pages that can be traversed with *Previous* and *Next* buttons. Avoid forcing the user to remember a series of commands or how to navigate to various seemingly unrelated places to finish the task.
- While shortcut keystrokes and command-line interfaces are appreciated as time-savers by advanced users, you shouldn't make these the sole means of interaction, as they require memorization and recall. If you must rely on shortcuts or commands, make it easy to refer to a quick-reference chart or other help material.
- Make icons and names easily recognizable so that they can be found easily when scanning a list or menu. Icons can be clarifying if the images represent things that are concrete and recognizable. The icons also need to be easily differentiable from each other. If the user has to memorize and recall what an peculiar or abstract icon really means, or if the user must squint and try to puzzle out the difference between several nearly identical icons, then it defeats the purpose of using a graphical representation. An icon's image and a textual label should be shown together if the image is abstract or its meaning is unclear.
- If the user will have to work with ID numbers such as product or customer numbers, it can be advantageous to limit these to about seven or fewer digits or characters in length, if possible, so that it's easier to temporarily store the numbers in working

memory.

- Arbitrary names are harder to remember and recall than names that accurately describe what they represent. When names don't match what they actually represent, not only do they become problematic to learn, but the additional memorization and recall add to the user's cognitive burden.

  Shell commands in Unix-based operating systems are particularly bad at violating this principle. For example, most Unix systems offer a command called "less" for showing the contents of a text file. The name "less" is a play on words; "less" is an enhanced version of another command called "more". ("more" is a filter command that lets you view a file or other stream of data in a page-by-page fashion; its name stems from the fact that it makes the console pause until you press the space bar to show "more" of the file or stream contents.) The name "less" doesn't in any way communicate what that command does; it's a banal pun by somebody trying to be clever. For the use case of showing the contents of a file, "list" would be one example of a more self-explanatory, more memorable, and equally concise name for this command.

- Offer a good online help system with search and index capabilities, or offer other forms of reference documentation, so that users can quickly look up instructions and information that they may have forgotten.

- In search and index systems, allow users to use synonyms and variations in case they can't recall the exact word or phrase (or the correct spelling) needed to identify something.

- Try to use commonly accepted, well-known, standard names for things rather than inventing your own terminology. Avoid using abbreviations or acronyms if they are not immediately obvious.

- Be consistent; don't make the user remember different ways of performing the same action in different contexts. I'm familiar with one enterprise system where some drop-down lists had to be opened with a Ctrl-L keystroke combination, while certain other drop-down lists had to be opened with Alt-F11. Technical limitations of the platform were given as the reason for that situation: fixed lists of values could be presented with the Ctrl-L drop-down list, where as dynamic lists of values required the alternative. However, I suspect a little more effort could have yielded a more user-friendly solution.

The impact of hardware devices on software ergonomics

A product that is **ergonomic** is designed in a way that helps reduce physical discomfort, stress, strain, fatigue, and potential injury during operation. While ergonomics is usually associated with physical products, the design of the a software application's interface also influences the way the user physically interacts with the hardware device on which the application runs. And ergonomics also extends to the cognitive realm, as we seek to design software that helps people work more productively, comfortably, and with a minimum of mental strain. We can do this by reducing the dependence on memorization, for example.

To create an ergonomically sound software application, it is important to first think about the properties and the context of use of the hardware device on which the application will run. For the majority of consumer and business applications, there are currently three main forms of general-purpose personal computing devices:

- **Desktop** and **laptop computers** with a screen, keyboard, and a pointing device such as a mouse or trackpad. These devices are comfortable for users sitting at a desk for a long period of time.
- **Tablet devices** with touchscreens. These devices have a form factor that is comfortable for sitting and consuming content (reading webpages, watching movies, etc.), but entering information and creating content via touch-screen control is generally not as comfortable and convenient as with a desktop machine.
- **Mobile phones** and similar devices such as portable music players. These devices are usually used for relatively short bursts of activity throughout the day and while on the go.

For more specialized applications, you might have a combination of software and custom-designed, special-purpose hardware. Examples include a vending machine that sells subway tickets, an automated teller machine, or an industrial thermostat control. If you are a designer for such a product, you may have responsibility for designing the form of the physical interface in addition to the software.

To give you an idea of some of the practical ergonomic aspects that you should keep in mind when designing for different devices, let's compare desktop computers with touchscreen tablets:

- Tablet devices with multi-touch touchscreens are pleasant and fun to use from an interaction standpoint because you can interact directly with on-screen elements by touching them with your finger. Desktop machines generally don't offer touchscreens (although at the time of publication, this is beginning to change). Touchscreens on desktop monitors can be uncomfortable for extended use, because reaching your arm out to the monitor places strain on the arm and shoulder muscles, and this quickly becomes physically tiring. Desktop setups thus rely on pointing devices such as mice and trackpads, which can be used with the hand and arm in a resting position on the desk. These pointing devices introduce a level of indirection, however: moving the pointing device moves a cursor on the screen.

- On desktop systems, there is a pointing device cursor (mouse arrow), whereas touchscreen devices have no such cursor. Some mouse gestures, such as hovering the cursor over a control, thus have no counterpart in touchscreen systems (and so, for example, pop-up "tooltip" messages that appear when you hover the mouse cursor over a control do not exist on touchscreen systems). On both desktop and touchscreen systems, however, a text cursor, called a **caret**, appears when a text field receives the focus.
- While a mouse may have multiple buttons, and clicks can be combined with holding down modifier keys (Control, Alt, Shift, Command, etc.), touchscreens don't offer as many activational options. When you drag your finger across the screen, is it to be interpreted as a scrolling gesture, or an attempt to drag and drop an object on the screen? Cut-and-paste and right-clicking to get a context menu are easy on a desktop machine, but on a tablet, such operations require double-touch or touch-and-hold gestures that are not always immediately evident.
- Fingers range in size substantially; young children have small, narrow fingertips, whereas some men have very thick, fat fingers. Touchscreen buttons and icons thus must be large enough to accommodate "blunt" presses without triggering other nearby controls. In contrast, the mouse arrow allows pixel-precise pointing, and so buttons and icons can be substantially smaller on desktop applications than on touchscreen devices.
- When the user is touching something on the screen, the user's finger and hand will obscure part of the screen, so you have to be careful about what you display and where, so that important information is not hidden. When pressing an on-screen button, the user's fingertip will obscure the button being pressed. Because button presses don't always "register", users seek visual feedback to see that the button press worked, and so you either need to make the buttons large enough so that the animation of the button being depressed is visible, or you should give some other clue when the user retracts the finger to show that the button was pressed (maybe pressing a *Next* button makes the application navigate to the next screen, which is very clear feedback that the button press was successful).

Auditory feedback, like a clicking sound, can also be useful as a cue that the button was pressed successfully. Some mobile and tablet devices can also vibrate slightly, to provide tactile feedback when a button is pressed.

- Mobile devices and tablet devices are often held by the user in one hand while standing, and so the user has only the other hand free to operate the touchscreen.

When designing a product, understanding the constraints and limitations, as well as the opportunities, of the hardware devices the software will run on will help you design appropriate and comfortable interactions.

Cognitive load and mental effort

Users interact with a software application by means of physical actions. These actions can include pressing keys and key combinations, typing for a sustained amount of time, precisely aiming a pointing device (homing the mouse pointer onto a target), and clicking the mouse or gesturing on the screen.

Performing each such action incurs a **cost** of time, physical effort, and some mental effort. In other words, performing an action requires the user to expend some energy.

In addition to the above physical actions, there are other actions that cost time, physical and mental effort, and energy, such as:

- Reading labels, titles, and instructions
- Choosing an option from a list or menu
- Scrolling
- Navigating
- Seeking (trying to locate something specific)
- Switching contexts (for example, switching between two windows, pages, or tabs)
- Switching visual attention (for example, reading text, then referring to an illustration, and then returning to the text)
- Recalling from memory a specific piece of information, such as a command name or an ID number
- Recalling from memory how to carry out a task sequence

- Waiting for a response from the system
- Recovering from some kind of distraction (like an unexpected pop-up dialog that is not directly relevant to the task at hand)

In general, users don't mind performing actions when the actions clearly help to make progress towards achieving a desirable goal, and when there seems to be some underlying rationale for why the actions are necessary.

However, being forced to undertake actions that are perceived as unnecessary quickly produces feelings of annoyance; the application is forcing the user to waste time and energy. With enough repeated annoyances, it is only natural that resentment will form towards the product (and its designers, who evidently have little respect for the user).

Therefore, we should obviously aim to design applications in such a way that any unnecessary actions, thinking, or waiting are eliminated.

But consider this situation: If we can take a design that requires ten mouse clicks to accomplish a task, and revise it so that it only requires seven mouse clicks, then we'd probably say that the revised design is superior, because three evidently superfluous mouse clicks were eliminated. But what if the first design didn't require much conscious thought; the user might have had to repetitively click *Next* ten times in a row in a wizard where all the default settings were acceptable. And perhaps the second design required a lot of thought as to how to set up various options in a very large, complex control panel. In this case, it sounds like the first design is the easier one to use, even though it involves a few more clicks.

So while we should generally aim to reduce the average number of physical and low-level actions a user must perform, we should really take a broader view. We must consider the *cognitive load* imposed by the task and the user interface.

Cognitive load refers how mentally taxing it is to do a task. It is essentially a way of referring to how much sustained attention and brainpower is required to do something.

The more complex a task is — that is, the larger the number of contextual details of the task that the user has to keep in working memory, and the more the task demands a high level of focused attention — the higher the cognitive load is for that task.

And so a good general design strategy is to reduce the user's cognitive load as much as possible. The title of Steve Krug's popular usability book, *Don't make me think!* (Krug, 2000), is a useful slogan to remember — reducing the amount of thinking the user has to do is arguably the single most important goal to consider when designing usable software products.

Thinking might just be the hardest kind of work there is. At least, it is the most avoided. For instance, most users of spreadsheets and word processors use a lot of repetitive manual keystrokes for, say, formatting content. While these users may suspect that there must be a more efficient way of doing the reformatting, they continue to use their trusted but labor-intensive methods, because thinking through the problem, investigating alternatives, and learning how to configure and use features like styles, macros, or scripting would involve more intense mental effort than just plowing through using manual techniques that require less thinking. While an alternative method would probably save time over the long run, most users doesn't want to spend the time and effort in the short term to figure out that alternative.

Types of thinking

Let's consider some of the different kinds of thinking that users of software might have to engage in. If your application requires some of the following kinds of thinking, it might be worth investigating whether you can restructure the design to reduce to some degree the need to engage in them:

- Determining what the next step in a procedure should be
- Holding things in working memory for the duration of the task
- Having to recall facts, commands, or procedures from long-term memory
- Having to memorize things in long-term memory
- Having to look up information from a reference source
- Making decisions or judgements
- Mentally integrating information from multiple sources

For many intellectual tasks, a lot of thinking goes on in the user's head, and the software only incidentally serves as a way to facilitate the activity and record the results of the thinking. For example:

- Creative output: coming up with ideas and generating the content of writing, audio, or visual art projects
- Problem-solving

- Reading and understanding long passages of text
- Actively conducting research with the goal of discovering, synthesizing, or creating new knowledge

For these higher-level activities, there's often little you can do to reduce the amount of thinking required, because the thinking involved has very little to do with the hands-on operation of the software. The best you can do is ensure that the software works reliably, generates good-quality output, and supports the task as best as possible.

So if your application provides access to academic journal articles, the search function should provide relevant results, which will reduce the amount of time the user spends searching, navigating, and reading. Likewise, if your application is a word processor and your user is writing a novel, there's little you can do to relieve the user of the very tricky mental work involved in creative writing, including developing a plotline, creating and fleshing out characters, and crafting the narrative and dialogue. But your application can aid the user in secondary ways, perhaps by offering support for organizing notes and materials, or by providing smarter tools and workflows for editing and proofreading.

Reducing the amount of all types of work and effort — both thinking and physical actions — will result in a product that is easier and more enjoyable to use.

Quantifying cognitive load and task efficiency

If we wanted to attempt to quantify the cognitive load — i.e., the thinking and effort involved — for performing a particular task, we could write out a list of the actions or operations that a user would have to do to carry out that task under normal circumstances. We could then estimate or assign a score, representing our idea of the effort involved, to each individual action, and then sum up all of the scores to get a total effort score for the task. We could then evaluate different design alternatives by comparing their scores.

The KLM-GOMS model, the *Keystroke-Level Model* for the *Goals, Operators, Methods, and Selection Rules* analysis approach (Card *et al.*, 1983), is one analysis technique based on this idea, but instead of assigning scores representing effort, an estimate of the time required to do each action is estimated instead. The amount of time it takes to complete a task is a good proxy for physical effort, although it does not accurately measure the intensity of mental effort.

Let's take a very condensed tour of the KLM-GOMS approach.

To accomplish a goal, the user will break the work into tasks, and for each task unit, the user will take a moment to construct a mental representation and choose a strategy or method for carrying out the task. This preparation time is called the **task acquisition time**, and can be very short — perhaps 1 to 3 seconds — for routine tasks, or much longer, perhaps even extending into several minutes, for creative design and composition tasks.

After the task acquisition, the user carries out the task by means of a sequence of actions or operations. The total time taken to carry out the actions is called the **task execution time**. Thus the total time required to complete a task is the sum of the task acquisition and task execution times.

To estimate the task execution time, KLM-GOMS defines basic operations (we assume here that we are dealing with a keyboard-and-mouse system):

Operation		**Description**	**Suggested average values**
K	Keystroking	Pressing a key or mouse button, including the Shift key and other modifier keys	Best typist: 0.08 s Good typist: 0.12 s Average typist: 0.20 s Worst typist: 1.20 s
P	Pointing	Moving the mouse pointer to a target on the screen	1.10 s
H	Homing	Moving a hand from the keyboard to the mouse or vice-versa	0.40 s
M	Mental operation	Mental preparation or thinking	1.35 s
R	System response operation	Time taken for the system to respond	varies

Figure 9-1

So to use the mouse to click on a button, we would have a sequence of operations encoded as "HPK": homing, to move the hand to the mouse; pointing, to move the mouse to target the mouse cursor over the button; and a keystroke, representing the pressing of the mouse button.

In addition to these operators, the KLM-GOMS model also includes a set of heuristic rules governing how the "M" operation, the mental operation, is to be inserted into an encoded sequence. For instance, "M" operations should be placed before any "K" and

"P" operations, except for various special cases. So the "HPK" sequence discussed above would become "HMPK". The suggested heuristic rules are quite complex and arcane, so please refer to the original article by Card *et al.* if you need to know all of the details. In actual practice, it will usually suffice if you simply insert "M" operations in places where you feel there is some thinking or decision-making effort involved.

As an example, let's consider the task of finding instances of a search term in a document in a text editor. One possible sequence of actions to accomplish this might be:

- Click on the "Search" menu
- Click on the "Find Text" item in the menu
- Enter "kittens" as the search term in dialog
- Click on the "OK" button

This can be encoded using KLM-GOMS and used to formulate an estimate of the average time required as follows:

Action/Operation	Encoding	Time (s)
Task acquisition	-	1.5
Click on the "Search" menu	H[mouse]	0.40
	MP["Search" menu]	1.35 + 1.10
	K["Search" menu]	0.20
Click on the "Find Text" item	MP["Find Text" item]	1.35 + 1.10
	K["Find Text" item]	0.20
	H[keyboard]	0.40
Enter "puppy" as the search term	7K[k i t t e n s]	7(0.20)
Click on the "OK" button	H[mouse]	0.40
	MP[OK button]	1.35 + 1.10
	K[OK button]	0.20
Total		12.05 s

FIGURE 9-2

Of course, we would expect a more skilled user to be able to accomplish the same task in substantially less time by using shortcut keystrokes rather than the mouse. You could do a separate analysis of each possible task sequence to compare the relative efficiency of each alternative.

There are obviously limitations to this kind of analysis; it provides a general rough estimate only, and it assumes that users know the right sequences of actions to complete a task. It also does not account for errors and mistakes. But when you are designing an interface and considering how to design an interaction, methods such as the KLM-GOMS model give you a way to compare the efficiency of different alternatives, and all other things being equal, the alternative that can be done in the least amount of time is the most convenient to the user, and often involves the least cognitive load.

Recreational and creative uses of software

Our discussion of cognitive load might make it sound like operating software is an arduous ordeal, and while this might be true for some enterprise systems, it's not the case for all software. Games require interaction with an interface, but this is not perceived as being work. Having to click 100 times to delete 100 spam comments on a blog would be considered intolerably poor design. Yet people will happily click hundreds of times when playing a game such as *Mah Jongg*. There are also games like flight simulators where players gain enjoyment from doing what others, such as aircraft pilots, do in their everyday jobs.

As well, when people get deeply involved in producing a creative work, whether writing a novel or drawing art, what might appear to be work to others may not be perceived as work by the artist. And so work that is voluntary and creative is simply more pleasurable than work that is involuntary and mundane. Motivating factors, like competition in games, can also change the way work is perceived.

Design techniques for reducing cognitive load

We've argued minimizing cognitive load is essential for making software more pleasant to use. Here are some tips and techniques to employ for reducing the cognitive load imposed by your software product:

- Use consistent naming, labelling, icons, and visual presentation to reduce any confusion.
- Avoid redundancy so that the same information doesn't need to be read and processed repeatedly.
- Put related things close together, and avoid forcing the user to switch between different tabs or windows or to scroll back and forth to find or enter information.
- Avoid distractions like pop-up dialogs that break the user's concentration and flow.

- Identify and eliminate any unnecessary steps. You might allow expert users to hide instructions and turn off warning messages.
- If your application has multiple tasks or screens that share similarities, be consistent in designing the visual appearance and workflow of these tasks and screens, so that once the user has learned how to use one, the same patterns can be applied to the others.
- Automate as much manual work as is reasonably possible. In some cases, though, you may want to allow experts to have the option of doing things manually if they need an extra level of control or precision.
- Where there is a list of steps to be followed, always make it clear how to do the next step. When possible, guide users through tasks with wizard-style interfaces rather than force users to memorize a complex procedure.
- Use visual cues and clues to avoid the need for memorization and recall. Allow options to be selected from menus instead of requiring users to memorize commands.
- Reduce delays and latency as much as possible. Give feedback quickly. If an operation will take a long time, use a progress bar or other indicator to show that the system is busy, and when possible, give an estimate of how much time the remainder of the processing will require.
- In productivity applications, opening the application with a blank document can be confusing for new users, as they may not know where to begin. When possible, offer to take the user to a tutorial in the online help system, or provide templates or sample documents so that users can modify an existing document and learn by following a pattern.
- Avoid forcing the user to memorize data in the short term. For example, in one enterprise system (the same one as mentioned previously, with the inconsistent drop-down lists), in order to accomplish virtually any use case, the user was required to visit a series of screens, and the same nine-digit customer number had to be re-entered in each window. This was absurd when the system could easily remember the context of which customer is being operated on and fill in this information automatically.

Flow states, focus, concentration, and productivity

Many kinds of software, including productivity applications and enterprise information systems, are intended to be used for sustained periods of time. Such applications should encourage the user to focus and work productively. Similarly, entertainment products aim to immerse the user in an enjoyable experience.

Psychologist Mihaly Csikszentmihalyi described and popularized the concept of **flow**, which is the mental state of being completely focused on an activity. For a user who is in a flow state:

- Performance of the activity occurs naturally and unconsciously. Creativity and productivity are high.
- The user experiences deep concentration and immersion in the activity. The user is simultaneously alert and relatively relaxed.
- The user often becomes so engrossed in the activity that he or she is unaware of the passage of time (often described as "living in the moment").
- The difficulty of the activity is a good match for the user's skill; there is sufficient challenge to keep the user's interest, but not so much that the task seems impossible, and the activity is not so mundane that it causes boredom.
- The user is confident and has a sense of control over the situation.
- Usually, the user is working towards achieving a specific goal. (For some applications, the goal may not always be particularly productive; for games, the goal may be simply to finish one more level.)

Here are some things you should know about flow states with regard to software:

- Beginning users generally cannot be expected to be able to enter a flow state; it requires some level of comfort and competence with operating the application.
- It is often difficult to get into a flow state, and simply wishing to concentrate does not make it happen. Typically, it takes 15 minutes or more of struggling and working unproductively before one can "get into the groove".
- Interruptions such as phone calls and incoming e-mail notifications, and distractions such as chattering coworkers or a television in the same room, can pull a user out of a flow state. When returning to the activity after a distraction, it usually takes another period of time to get back into a flow state.

There is little you can do as a designer to *explicitly* help a user enter a flow state, but you can encourage and sustain concentration and flow by making the experience work smoothly and by minimizing or eliminating any repeated frustrations that might hinder the user from concentrating. Here are some design suggestions for doing that:

- Try to eliminate interruptions like modal pop-up dialogs that present notification and warning messages. Offer expert users the option of turning off any repetitive warnings.

- Keep the visual presentation simple. Brightly-colored images, and especially anything animated or blinking, can distract the user from reading text or concentrating on a work activity.

- When helping guide the user through task flows, make it obvious what the next step is, so that the user doesn't have to start exploring the interface, which easily leads to distraction.

- Avoid making the user switch repeatedly between different pages, screens, or tabs to find related information; each context switch can be disorienting and can cause users to forget what they were just doing.

- Make it easy for users to save any work in progress and then later pick up where they last left off.

- Show completion progress for lengthy tasks. When possible, reward the user for completing tasks; even a simple chime sound effect when some lengthy process is completed can be satisfying.

- Ensure that the system gives feedback promptly; especially in web-based systems, strive to reduce latency. Having to wait several seconds for confirmation that a button was pressed can become very annoying very quickly and can break the flow of work.

- It can be hard for humans to concentrate on multiple things at one, so when possible, don't make users manage multiple tasks at the same time. On the other hand, when the system is busy with a long-running process, you might give the user the option to have the process run in the background so that he or she can work on something else in the meantime. When users have to wait for an unknown length of time, they will frequently switch to something else while waiting (like checking e-mail or surfing the web).

If you are designing a typical software application, preventing distractions in the user's environment is out of your control. However, in some cases, you and your team may have the opportunity to influence the design of users' physical workspaces. For example, for an air traffic control operations center, in addition to designing the software itself, you may be able to influence the layout and design of the workstations and the office facility to prevent distractions.

Motivation, rewards, and gamification

Can software products be designed to motivate users and increase productivity?

If you're running an organization and your staff gets their work done using an enterprise application, you want to increase their productivity. Or, if you're running a community-driven website that relies on user-generated content, you want to encourage participation and repeat visits. Especially in a business setting, some of the tasks that have to be done are often tedious or unpleasant.

But there's one class of applications that tends to have little difficulty keeping users intensely focused and always coming back for more: Games. Some people think that some of the things that make gameplay addictive can also be applied to other kinds of applications. This is called **gamification**, and it's currently a hot fad.

What makes games addictive?

- First, there's the *voluntary* nature of game-playing. People are more likely to enjoy something when they're choosing to do it, rather than being required to do it.

- Second, games have *goals* and *rewards:* You want to get to the next level, and it's satisfying when you finally achieve it. Some games have elaborate systems of rankings, and as your skill improves, you get promoted; other games revolve around hunting for various desirable "items". And winning the game is ultimately the most satisfying reward.

- Third, as players achieve these goals and rewards, there is a sense of *progression* and an awareness that the player's *skill* is improving.

- Fourth, most online games are multiplayer games, and so there is an element of *competition*. Many people are driven to win and want to be the best. There is pride and social recognition in being at the top of the "high scores" leaderboard.

- Finally, multiplayer games can be a *social* experience, whether you're competing head-to-head with other players, or forming cooperative teams. For some people, games are a casual way to spend time and share social experiences with friends and family.

If these ideas make games fun and addictive, then can some of those ideas be brought to other products like enterprise applications and websites, and will this make those products more fun and addictive? It depends on the product and its users, but often the answer is yes — as long as it's not done in an overly gimmicky way.

Stack Overflow, a programming question-and-answer community website, is enormously popular. Much of that popularity today is due to the vast amount of content that often shows up at the top of the search results for programming-related queries. But how did they get all that content? It was created by the users, and a clever incentive system had a lot to do with it. Users accumulate points for successfully answering questions and earn "badges" as recognition of achieving certain milestones. Some badges unlock special privileges, like the ability to moderate discussions. And users with lots of points and badges enjoy respect and status for their contributions to the community.

Rewards systems can be effective for work that is easily measured. But you can breed resentment if the rewards system is not seen as reliable or fair. Creative work is particularly difficult to reward because objective metrics for measuring quality and even productivity are often impossible to define. For example, how would you create an algorithm to judge the quality of a graphic artist's logos? Or if one programmer took two hours and wrote 100 lines of code to solve a problem, and another took one hour but needed 200 lines, who is more productive?

One proxy for quality is popularity; on a community-driven website, you can let users "upvote" or give points to other contributors to reward them for good contributions. When the community is large and active, this system can be quite effective. This sort of peer voting is more problematic in a workplace setting, though. Asking employees in small teams to judge each others' work and hand out rewards rarely results in objective evaluations and can exacerbate office politics.

Reward systems are always well-intentioned, and yet they often lead to unexpected and unintended consequences. In a business environment, management will inevitably use these systems as a metric for judging and comparing workers' performance, even if that was not the original intention, and this can be problematic if the rewards system is not an accurate measurement of actual job performance. And metrics-based incentives encourage workers to game the system, to the detriment of the organization and its customers. I'm aware of a technical support call center that measured the time spent per call and disciplined workers whose average time per call exceeded a certain target. While the scheme was intended to reduce costs, it only had the effect of forcing workers to do anything possible to reduce call durations. So rather than try to actually resolve callers' issues, workers would unnecessarily forward calls to someone else or even give faulty but short answers so that they could hang up as soon as possible. This only led to an increased volume of calls from angry customers!

Competition can be a powerful motivator for some people; sales teams have used competition (such as salespeoples' results and rankings being posted in the hallway) as a motivator for years. But competition can be a turn-off for many others. If you structure the system so that there is only one winner, then you'll have one happy winner and the rest of your users will be unhappy losers. And on community websites, competition can discourage newcomers: How can a new user possibly compete against the obsessive-compulsive contributors who have been participating for years and have 50,000 points?

So if you're considering applying some of the ideas of gamification to your product, be sure that you understand your users, and be sure to think through all of the consequences.

Gamification can be very appealing to some audiences, and gimmicky to others. Established professionals, for instance, tend to be highly self-disciplined, take a lot of pride in their skills and accomplishments, and gain intrinsic satisfaction out of doing their job well. These people will be personally insulted by the notion that their work can be turned into a "game" with phony competition and incentives.

For professional users, the simple indication of progress on long tasks is probably the best reward. There is satisfaction in finishing a task, and for longer tasks, it's reassuring to know that you're making progress towards completion. So in a data-entry-centric application such as income tax software, it can make sense to break down data-entry forms into sections or pages that be checked off when complete. A graphical progress meter showing the percentage of work completed and work remaining can also be very useful. Figure 9-3 shows an example of a progress indicator on LinkedIn's profile editing page:

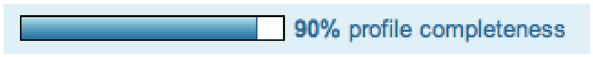


FIGURE 9-3

Because there is satisfaction derived from completing tasks, it is a good idea to break lengthy work sessions into smaller task units whenever possible. For example, reading a 500-page book with 50 small chapters tends to be more satisfying than reading a 500-page book with only 5 big chapters, because there's a feeling of completion and accomplishment when you reach the end of a chapter.