Many Factors Affect Learning

Chapter 10 contrasted the “automatic” processes that our brain uses to carry out well-learned activities with the conscious, highly monitored, controlled processes that we use to solve novel problems and perform calculations. Automatic processes consume little or no short-term memory (attention) resources and can operate in parallel with each other, while controlled processes place high demands on short-term memory and operate one at a time (Schneider & Shiffrin, 1977).

The first time or even the first several times we perform an activity, we do it in a highly controlled and conscious manner, but with practice it becomes more and more automatic. Examples include peeling an apple, driving a car, juggling balls, riding a bicycle, reading, playing a musical instrument. Even an activity that might seem to require our attention, such as sorting good cherries from bad ones, can become automated to the point that we can do it as a background task, with plenty of cognitive resources left over for having a conversation, watching the news on television, etc.

This progression from controlled to automatic raises an obvious question for designers of interactive applications, online services, and electronic appliances: How can we design them so that using them becomes automatic within a reasonable amount of time?

This chapter explains and demonstrates factors that affect how quickly people can learn to use interactive systems. To preview the factors, We learn faster under the following conditions:

- Operation is task-focused, simple, and consistent
- Vocabulary is task-focused, familiar, and consistent
- Risk is low

WE LEARN FASTER WHEN OPERATION IS TASK-FOCUSED, SIMPLE, AND CONSISTENT

When we use a tool—whether it is computer-based or not—to do a task, we have to translate what we want to do into the operations provided by the tool. Some examples:
Imagine that you are an astronomer. You want to point your telescope at the star Alpha Centauri. Most telescopes don’t let you just specify what star you want to observe. Instead, you have to translate that goal into how the telescope’s positioning controls operate: in terms of a vertical angle (azimuth) and a horizontal angle, or perhaps even the difference between where the telescope is pointing now and where you want it to point.

Assume you have a telephone that doesn’t have speed dial. To call a person, you have to translate the person into a telephone number and give that to the phone.

You want to create an organization chart for your company, using a generic drawing program. To indicate organizations, suborganizations, and their managers, you have to draw boxes, label them with organization and manager names, and connect them with lines.

Cognitive psychologists call the gap between what a tool user wants and the operations the tool provides “the gulf of execution” (Norman & Draper, 1986). A person using a tool must expend cognitive effort to translate what she wants into the tool’s available operations and vice versa. That cognitive effort pulls the person’s attention away from her task and refocuses it on the requirements of the tool. The smaller the gulf between the operations that a tool provides and what its users want to do, the less the users need to think about the tool and the more they can concentrate on their task. As a result, the tool becomes automatic more quickly.

The way to reduce the gulf is to design the tool to provide operations that match what users are trying to do. To build on the examples above:

- A telescope’s control system could have a database of celestial objects, so users could simply indicate which object they want to observe, perhaps by pointing to it on a display.
- Telephones with speed dial allow users to simply specify the person or organization they want to call, rather than having to translate that to a number first.
- A special-purpose organization chart editing application would let users simply enter the names of organizations and managers, freeing users from having to create boxes and connect them.

To design software, services, and appliances to provide operations matching users’ goals and tasks, designers must thoroughly understand the user goals and tasks the tool is intended to support. Gaining that understanding requires three steps:

1. Perform a task analysis
2. Design a task-focused conceptual model, consisting mainly of an objects/actions analysis
3. Design a user interface based strictly on the task analysis and conceptual model
We learn faster when operation is task-focused, simple, and consistent

Task analysis

Describing in detail how to analyze users’ goals and tasks is beyond the scope of this book. Entire chapters—even whole books—have been written about it (Beyer & Holtzblatt, 1997; Hackos & Redish, 1998; Johnson, 2007). For now, it is enough to say that a good task analysis answers these questions:

- What goals do users want to achieve by using the application?
- What set of human tasks is the application intended to support?
- Which tasks are common, and which ones are rare?
- Which tasks are most important, and which ones are least important?
- What are the steps of each task?
- What are the result and output of each task?
- Where does the information for each task come from?
- How is the information that results from each task used?
- Which people do which tasks?
- What tools are used to do each task?
- What problems do people have performing each task? What sorts of mistakes are common? What causes them? How damaging are mistakes?
- What terminology do people who do these tasks use?
- What communication with other people is required to do the tasks?
- How are different tasks related?

Once these questions are answered (by observing and/or interviewing people who do the tasks that the tool will support), the next step is not to start sketching possible user interfaces. The next step is to design a conceptual model for the tool that focuses on the users’ tasks and goals (Johnson & Henderson, 2002).

A conceptual model explains the function of the software and what concepts people need to be aware of in order to use it. Ideally, the concepts should be those that came out of the task analysis. The more direct the mapping between the tool’s concepts and those of the tasks it is intended to support, the less translating users will have to do, and the easier the tool will be to learn.

After you have designed a conceptual model that is task-focused, as simple as possible, and as consistent as possible, you can design a user interface for it that minimizes the time and experience required for using the application to become an automatic process.

Objects/actions analysis

The most important component of a conceptual model is an objects/actions analysis. This specifies all of the conceptual objects that an application will expose to users, the actions that users can perform on each object, the attributes (user-visible settings) of each type of object, and the relationships between objects (Card, 1996; Johnson & Henderson, 2002).
The software’s implementation may include objects other than those listed in the conceptual model, but, if so, those extra objects should be invisible to users. Objects and actions that are related purely to implementation—such as a text buffer, a hash table, or a database record—do not belong in a conceptual model.

The objects/actions analysis, then, is a declaration of the concepts that are exposed to users. Follow this rule: “If it isn’t in the objects/actions analysis, users shouldn’t know about it.”

If we were designing software for managing checking accounts, a task-based objects/actions analysis would include objects like transaction, check, and account. It would exclude non-task-related objects like buffer, dialog box, mode, database, table, and text string.

A task-based conceptual model would include actions like writing and voiding checks, depositing and withdrawing funds, and balancing accounts, while excluding non-task-related actions like clicking buttons, loading databases, editing table rows, flushing buffers, and switching modes.

In a task-focused conceptual model, the attributes might be as follows:

- **Checks** have a payee, a number, an amount, memo text, and a date
- **Accounts** have an owner and a balance
- **Transactions** (deposits and withdrawals) have an amount and a date

If the model included attributes from computer technology, such as transaction record format, it would not be task-focused. Users wouldn’t care what internal format the application used for storing transaction records. Forcing them to care would detract from the learnability and usability of the software, no matter how much effort went into designing the user interface.

The objects, actions, and attributes for checkbook management may seem obvious, so let’s consider a task for which the objects/actions analysis may seem less clear-cut: customers posting comments about products at an online store.

Suitable objects for a conceptual model might include customers, products, customer comments, and responses to comments. Unsuitable objects would include databases, tables, and persistent cookies.

Actions on products would include viewing and adding comments. Actions on comments would include viewing and responding, and, for a user’s own comments, editing. The attributes of a comment might include the title, the customer’s name, and the posting date.

Notice that for both the checkbook management application and the customer commenting system, important conceptual design issues can be decided before the user interface is designed, or even before we know whether the user interface is presented on a personal computer screen or via voice menus on a telephone.

**As simple as possible**

In addition to being focused on users’ tasks, a conceptual model should be as simple as possible. Simpler means fewer concepts. The fewer concepts a model has for
We learn faster when operation is task-focused, simple, and consistent. The better, as long as it provides the required functionality. Less is more, provided that what is there fits well with users’ goals and tasks.

For example:

- In a To-Do List application, do users need to be able to assign priorities of 1–10 to items, or are two priority levels — low and high — enough?

- Does a Search function need to allow users to enter full Boolean expressions? If it allowed that, would a significant number of people use it? If not, leave it out.

- Does a ticket machine in a train station need to be able to offer tickets for train routes other than the routes that this station is on?

In most development efforts, there is pressure to add extra functionality “in case a user might want it.” Resist such pressure unless there is considerable evidence that a significant number of potential customers and users really need the extra functionality. Why? Because every extra concept increases the complexity of the software. It is one more thing users have to learn. But actually it is not just one more thing. Each concept in an application interacts with most of the other concepts, and those interactions result in more complexity. Therefore, as concepts are added to an application, the application’s complexity grows not just linearly, but multiplicatively (see Fig. 11.1).

**Consistency**

The *consistency* of an interactive system strongly affects how quickly its users progress from controlled, consciously monitored, slow operation to automatic, unmonitored, faster operation (Schneider & Shiffrin, 1977). The more predictable the operation of

![Figure 11.1](image)

**FIGURE 11.1**

The complexity of an application increases non-linearly as concepts are added.
CHAPTER 11 Many Factors Affect Learning

EXCESS COMPLEXITY DUE TO SEPARATE CONCEPTS BEING TOO SIMILAR

Some software applications are too complex because they have concepts that overlap in meaning or functionality. For example, one company’s customer-support Web site presented four concepts that the developers considered quite different:

- **Membership**: whether a company had paid for the customer-support service.
- **Subscription**: whether a company had subscribed to a customer-support newsletter.
- **Access**: which areas of the customer-support Web site users in a company could access.
- **Entitlements**: services provided for each membership level.

Users confused these four concepts. The four concepts should have been collapsed into *one*, or at least fewer than four.

Another company developed a Web site for people seeking to buy a home. There were two ways to start looking for a home: (a) name the state, county, or town; and (b) point to a location on a map. The site called these two methods “by location” or “by map,” respectively, and required users to choose one. A usability test found that many users did not think of those as different ways of finding a home. To them, both methods were *by location*; they just differed in how the location was specified.

a system’s different functions, the more consistent it is. In a highly consistent system, the operation of a particular function is predictable from its *type*, so people quickly learn how everything in the system works and its use quickly becomes habitual. In an inconsistent system, users cannot predict how its different functions work, so they must learn each one anew, which slows their learning of the overall system and keeps their use of it a controlled, attention-consuming process.

Interactive systems can be consistent or inconsistent on at least two different levels: the conceptual level and the keystroke level. Consistency at the conceptual level is determined by the mapping between the objects, actions, and attributes of the conceptual model (see above). Do most objects in the system have the same actions and attributes, or not? Consistency at the keystroke level is determined by the mapping between the conceptual actions and the physical movements required
to execute them. Are all conceptual actions of a certain type initiated and controlled by the same physical movements, or not?

The objects/actions matrix

An optional but sometimes useful step in designing an interactive system is to illustrate its conceptual model as a matrix of objects and actions. Objects are listed down the left edge; actions are listed across the top (see Fig. 11.2). For now, we are ignoring the object type hierarchy and simply listing the objects. The more objects, the taller the matrix; the more actions, the wider.

Constructing an objects/actions matrix lets you visualize the simplicity or complexity of your interactive system’s conceptual model. The larger the matrix, the more concepts there are to learn. A tall matrix indicates many objects to master. A wide one indicates many actions to master. The matrix also illustrates how consistent or inconsistent the conceptual model is—how easy it is for users to transfer what they have learned about one part of the system to another.

A small, dense matrix indicates a design that will be easy to learn: few objects, few actions, and the operations on every type of object are the same (see Fig. 11.3A). For example, the conceptual objects in a simple drawing program would be graphical elements: lines, ellipses, arcs, rectangles, triangles, text labels, etc. The applicable actions on graphical objects would presumably be create, delete, view/edit attributes, move, copy, resize, rotate, flip, etc. The objects/actions matrix for such a simple drawing application would have a row for each object type and a column for each action. All the actions would apply to every object type, so the matrix would be densely packed, like that in Figure 11.3A.

A large, sparse matrix reflects an inconsistent design that will be hard to learn and remember because every conceptual object has different actions (see Fig. 11.3B). Such a design will be hard to learn and remember, no matter what user interface is plastered on it.

A good rule of thumb is to simplify the conceptual model so that the matrix representing it is as small and dense as possible. However, a small matrix reflects limited functionality. Achieving a small matrix is difficult when the application is anything
More functional than, say, a simple drawing program, a personal phone directory, or a Web site for looking up postage rates. Consider, for example, the objects and actions an intensive-care patient-monitoring system would have. Even a typical word processing application—e.g., Microsoft Word or Apple Pages—embodies a nontrivial array of conceptual objects and actions.

However, for any desired functionality, a designer can develop conceptual models of varying complexity. For example, most personal bank account tracking applications have similar functionality, but Intuit's Quicken has an ultra-simple conceptual model, which may be one reason that it is so popular. Software designers should just aim for the simplest conceptual model (with the most compact objects/actions matrix) for the required functionality.

Although easy-to-learn, easy-to-use systems often have small, dense object/action matrices, they can have other matrix configurations as well. Consider an application in which all functionality is accessed through five or six generic actions that apply to all objects. Such a system could have a large number of objects without much negative impact on learnability, because all objects operate in a totally consistent way. The object/action matrix for such a system, although tall, would be narrow and dense. This approach has been used to design some highly functional systems, such as the Xerox Star office workstation (Johnson et al., 1989). In Star, the same six commands—move, copy, open, delete, show properties, and copy properties—applied to all objects: characters, words, paragraphs, table rows, tables, charts, email messages, documents, folders, printers, etc.

If we include the object type hierarchy in the matrix, we can see another sort of conceptual model that is easy to learn: one in which objects fall into clear categories, each one having its own actions, perhaps with a few actions that apply to all objects (see Fig. 11.4). The matrix for such a system isn’t small or dense, but it also isn’t scattered. It has a regularity and consistency that aid learning and retention.

An example would be a real estate service offering both commercial and residential properties, with different actions for each as well as actions that apply to both types of properties.
We learn faster when operation is task-focused, simple, and consistent.

I regard creating an objects/actions matrix as an optional design step for two reasons:

- Experienced interaction designers rarely need to actually draw the matrix to know whether the conceptual model underlying their design is simple or complex, consistent or inconsistent.

- Usability testing can reveal aspects of an application’s conceptual model that designers did not know needed to be simplified.

It may be enough for designers, as they design the conceptual model for an application, to imagine what its objects/actions matrix would look like if they drew it.

The goal is to devise a conceptual model that is task-focused, as simple as possible, and as consistent as possible. From such a model, one can design a user interface for it that minimizes the time and experience required for using the application to become automatic.

Keystroke consistency

When a designer moves from conceptual design to actual user interface design, keystroke-level consistency becomes important.

Keystroke-level consistency is harder to illustrate and measure, but it is at least as important as conceptual consistency in determining how quickly the operation of an interactive system becomes automatic. The goal is to foster the growth of what is often called “muscle memory,” meaning motor habits.

Achieving keystroke-level consistency requires standardizing the physical actions for all activities of the same type. An example of a type of activity is editing text. Keystroke-level consistency for text editing requires the keystrokes (and pointer movements) to be the same regardless of the context in which text is being edited—documents, form fields, filenames, etc. Other types of activities for which keystroke-level consistency is desirable are opening documents, following links, choosing from a menu, choosing from a displayed set of options, clicking buttons, scrolling a display, etc.

A system that is inconsistent at the keystroke level does not let people quickly fall into “muscle memory” motor habits but, rather, keeps them guessing about what keystrokes to use in each context, even when contexts differ only slightly.
A common way that developers promote keystroke-level consistency is to follow look-and-feel standards. Such standards can be presented in style guides or they can be built into common user interface construction tools and component sets. Style guides exist for the entire industry and they exist separately for desktop software (Apple Computer, 2009; Microsoft Corporation, 2009) and Web design (Koyani, Bailey, & Nall, 2006). Ideally, companies also have internal style guides that augment the industry style guides to define a look and feel for their own products.

However, conventions are encapsulated, the goal is to stick to conventions at the keystroke level while perhaps innovating at the conceptual and task levels. We as designers really don’t want our software’s users to have to keep thinking about their keystroke-level actions as they work, and users don’t want to think about them either.

**WE LEARN FASTER WHEN VOCABULARY IS TASK-FOCUSED, FAMILIAR, AND CONSISTENT**

Ensuring that an application, Web service, or appliance exposes a small, consistent, and task-appropriate set of concepts to its users is a big first step, but it is not enough to minimize the time it takes for people to learn an interactive system. You also have to make sure that the vocabulary—what concepts are called—fits the task, is familiar, and is consistent.

**Terminology should be task-focused**

Just as the user-visible concepts in an interactive system should be task-focused, so should the names for the concepts. Usually, task-focused terms for concepts emerge from the interviews and observations of users that designers conduct as part of the task analysis. Occasionally, software needs to expose a concept that is new to users; the challenge for a designer is keeping such concepts and their names focused on the task, not on the technology.

Some examples of interactive software systems using terminology that is not task focused:

- A company developed a desktop software application for performing investment transactions. The application let users create and save templates for common transactions. It gave users the option of saving templates either on their own PC or on a network server. Templates stored on the PC were private. Templates stored on the server were accessible to other people. The developers used the term “database” for templates on the server because they were kept in a database. They used “local” for templates on the users’ own PC because that’s what “local” meant to them. Terms that would be more task focused are “shared” or “public” instead of “database”, and “private” instead of “local.”

- iCasualties.org provides up-to-date tallies of the number of Coalition military personnel killed or injured in the Iraq and Afghanistan wars. It starts by asking...
We learn faster when vocabulary is task-focused, familiar, and consistent.

Site visitors to select a “database.” However, visitors to this site don’t care or need to know that the Web site’s data is stored in multiple databases. Task-focused instructions would ask them to select a country in which there is an ongoing conflict, not a database (see Fig. 11.5).

**Terminology should be familiar**

To reduce the time it takes for people to master your application, Web site, or appliance, so that using it becomes automatic or nearly so, don’t force them to learn a whole new vocabulary. Chapter 4 explained that familiar words are easier to read and understand because they can be recognized automatically. Unfamiliar words cause people to use more conscious decoding methods, which consumes scarce short-term memory resources and thereby lowers comprehension.

Unfortunately, many computer-based products and services present users with unfamiliar terms from computer engineering—often called “geek speak”—and require them to master those terms (see Fig. 11.6). Why? Operating a stove doesn’t require us to master terminology about the pressure and chemical composition of natural gas, or terminology about the production and delivery of electricity. Why should shopping on the Web, sharing photographs, or checking email require us to learn geek speak such as USB, TIFF, or broadband? But in many cases, it does.
Some examples of interactive software systems using unfamiliar terminology:

- A development team was designing a video-on-demand system for schoolteachers to use in classrooms. The purpose of the system was to allow teachers to find videos offered by their school district, download them, and show them in their classrooms. The developers’ initial plan was to organize the videos into a hierarchy of “categories” and “subcategories.” Interviews with teachers showed, however, that they use the terms “subject” and “unit” to organize instructional content, including videos. If the system had used the developers’ terminology, teachers who used it would have to learn that “category” meant “subject” and “subcategory” meant “unit,” making the system harder to master.

- Continental Airlines’ Web site displays several error messages that speak “geek” (see Fig. 11.7). Most are attempts to tell the Web site user about a problem, but because they use an unfamiliar jargon, few users understand what the site is saying and so are unsure what to do. Such error messages are more appropriate for reporting the problem to system engineers. Error messages like these should either be rewritten in terms users understand, or they should be displayed to the Web site administrators who monitor the operation of the site rather than to the users.

- Windows Media Player sometimes displays error messages that use familiar terms in unfamiliar, “geeky” ways (see Fig. 11.8). The error message in the figure is referring to the state of the software, but the average Media Player user is likely to interpret it as referring to the state in which he or she lives.
We learn faster when vocabulary is task-focused, familiar, and consistent

In contrast to these examples, Southwest Airlines’ Web site tries to prevent errors from occurring, but when they do occur, it explains the problem using task-focused, familiar language (see Fig. 11.9).

Terminology should be consistent

People want to focus their cognitive resources on their own goals and tasks, not on the software they are using. They just want to accomplish their goal, whatever it is. They are not interested in the software. They interpret what the system presents only superficially and very literally. Their limited attentional resources are so focused on their goal that if they are looking for a Search function but it is labeled “Query” on the current screen or page, they may miss it. Therefore, the terminology in an interactive system should be designed for maximum consistency.

The terminology used in an interactive system is consistent when each concept has one and only one name. Caroline Jarrett, an authority on user interface and forms design, provides this rule:

Same name, same thing; different name, different thing. (FormsThatWork.com)

This means that terms and concepts should map strictly 1:1. Never use different terms for the same concept, or the same term for different concepts. Even terms that are ambiguous in the real world should mean only one thing in the system. Otherwise, the system will be harder to learn and remember.

An example of different terms for the same concepts is provided by Earthlink’s frequently asked questions (FAQ) page in the Web-hosting section of its site (see Fig. 11.10). In the question, the two available Web-hosting platforms are called “Windows-based” and “UNIX-based,” but in the table they are referred to as “Standard” and “ASP.” Customers have to stop and try to figure out which one is which. Do you know?
CHAPTER 11 Many Factors Affect Learning

FIGURE 11.10
Earthlink’s Web-hosting FAQ uses different terms for the same options in the question and in the table.

FIGURE 11.11
Photoshop uses different names for the tolerance parameter in two color-replacement functions: (A) “Fuzziness” in Replace Color; (B) “Tolerance” in Paint Bucket.
An example from Adobe Photoshop shows that inconsistent terminology can impede learning. Photoshop has two functions for replacing a target color in an image: *Replace Color*, which replaces the target color throughout an image with a new color, and *PaintBucket*, which replaces the target color in an enclosed area with a new color. Both functions have a parameter that specifies how similar a color in the image must be to the target color before it will be replaced. The inconsistency is that the Replace Color function calls this parameter “Fuzziness,” but the Paint Bucket function calls it “Tolerance” (see Fig. 11.11). Photoshop’s online Help documentation for Replace Color even says “Adjust the tolerance of the mask by dragging the Fuzziness slider or entering a value” [emphasis added]. If the parameter were simply called “Tolerance” in both color replacement functions, people who learned one function could quickly transfer that learning to the other. But it isn’t, so people have to learn the two functions separately.

Finally, WordPress.com provides an example of the same term for different concepts—also called *overloading* a term. For administering a blog, WordPress provides each blogger with a Dashboard consisting of monitoring and administrative functions organized into several pages. The problem is that one of the administrative function pages in the Dashboard is also called the “Dashboard,” so the same name refers to both the whole Dashboard and one page of it (see Fig. 11.12). Therefore, when new bloggers are learning to use WordPress, they have to discover and remember that sometimes “Dashboard” means the entire administrative area and sometimes it means the Dashboard *page* of the administrative area.

**Developing task-focused, familiar, consistent terminology is easier with a good conceptual model**

The good news is that when you perform a task analysis and develop a task-focused conceptual model, you also get the vocabulary your target user population uses to talk about the tasks. You don’t have to make up new terms for the user-visible concepts in your application—you can use the terms that people who do the task already use. In fact, you *shouldn’t* assign new names for those concepts, because any names you assign will likely be computer technology concepts, foreign to the task domain.1

From the conceptual model, a software development team should create a product *lexicon*. The lexicon gives a name and definition for each object, action, and attribute that the product—including its documentation—exposes to users. The lexicon should map terms onto concepts 1:1. It should not assign multiple terms to a single concept, or a single term to multiple concepts.

Terms in the lexicon should come from the software’s supported *tasks*, not its implementation. Terms should fit well into the users’ normal task vocabulary, even if

---

1 Unless you are designing software development tools.
they are new. Typically, technical writers, user interface designers, developers, managers, and users all help create the lexicon.

Certain concepts in GUIs have industry-standard names. These are the GUI equivalents of “reserved words” in programming languages. If you rename such concepts or assign new meanings to the standard names, you will confuse users.

Follow the product lexicon consistently throughout the software, user manuals, and marketing literature. Treat it as a living document: As the product evolves, the lexicon changes based on the basis of new design insights, changes in functionality, usability test results, and market feedback.
WE LEARN FASTER WHEN RISK IS LOW

Imagine you are visiting a foreign city on business for a week or two. You have spare time after your work duties are finished in the evenings and on weekends. Compare two possible cities:

- You have been told that this city is easy to get around in: it is laid out in a consistent grid of streets and avenues with clear street and subway signs written in a language you understand, and the residents and police speak your language and are friendly and eager to help tourists.

- You have been warned that this city has a convoluted, confusing layout, with winding, poorly marked streets; the few street and subway signs are in a language you cannot read, and residents don’t speak your language and are generally contemptuous of tourists.

In which city are you more likely to go out exploring?

Most interactive systems—desktop software, Web services, electronic appliances—have far more functionality than most of their users ever try. Often people don’t even know about most of the functionality provided by software or gadgets they use every day. One reason for this is fear of being “burned.”

People make mistakes. Many interactive systems make it too easy for users to make mistakes, do not allow users to correct mistakes, or make it costly or time-consuming to correct mistakes. People won’t be very productive in using such systems: they will waste too much time correcting or recovering from mistakes.

Even more important than the impact on time is the impact on learning. A high-risk system, in which mistakes are easy to make and costly, discourages exploration: people who are anxious and afraid of making mistakes will tend to stick to familiar, safe paths and functions. When exploration is discouraged and anxiety is high, learning is severely hampered.

In contrast, a low-risk system, in which mistakes are hard to make, low in cost, and easy to correct, reduces stress and encourages exploration, and therefore greatly fosters learning. With such systems, users are more willing to try new paths: “Hmmm, I wonder what that does.”

To foster learning, interactive systems should be low-risk environments, so users are not afraid to explore and try new things. Designing software this way means doing the following:

- Prevent errors where possible
- Deactivate invalid commands
- Make errors easy to detect by showing users clearly what they have done (e.g., deleting a paragraph by mistake)
- Allow users to undo, reverse, or correct errors easily
SUMMARY

The goal of this chapter is to explain and demonstrate the factors that affect how quickly people can learn to use interactive systems so proficiently that operating the system is handled largely by automatic cognitive processes. We learn to use interactive systems faster under the following conditions:

- Their operation is based on users’ goals and tasks (not on the implementation of the system), conceptually simple, and consistent.

- The vocabulary they employ is familiar to users, is based on that of the task domain, and is used consistently in the sense that it maps terms onto concepts 1:1.

- They provide a low-risk environment, in which errors are difficult to make and, when users do make errors, they are low in cost and easy to correct.