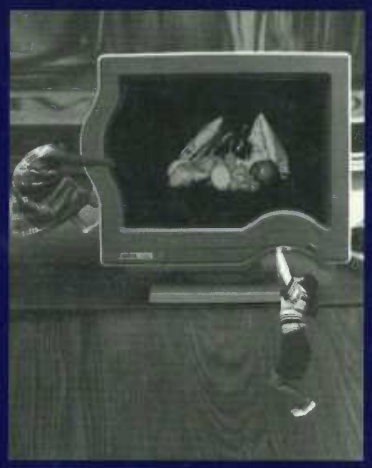
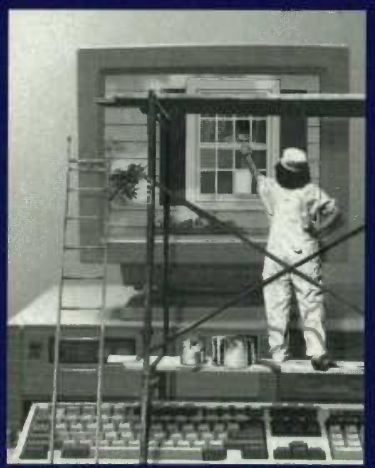
Designing the User Interface



Strategies for Effective Human–Computer Interaction



Ben Shneiderman

Second Edition

Shneiderman

Designing the User Interfa

Strategies for Effective Human–Computer Interaction Second Edition

Wesley

57286

Computer Science/User Interface

Second Edition

Designing the User Interface

Strategies for Effective Human–Computer Interaction

by

Ben Shneiderman, *University of Maryland*

Changes in computer systems have made vast information and communication resources available to a growing number of users. The user interface—the part of every system that determines how people control and operate their computers—is the primary gateway to these remarkable creative tools. Insofar as the user interface will affect the performance of a system, the productivity of users, and the satisfaction and clarity that users experience, its design has become a critical consideration in software and hardware development.

In this thorough revision of a best-selling book, Ben Shneiderman again provides the most complete and the most current introduction to user interface design. With characteristic enthusiasm and vision, the author carefully discusses the underlying issues, principles, and empirical results, and with an authority based on years of well-known work in the field, describes practical guidelines and techniques necessary to realize an effective design. System designers, programmers, software developers, and product managers will all find here invaluable guidance for making their systems easier to learn, easier to use, more productive for users and ultimately more satisfying to experience.

Coverage includes the human factors of interactive software (with added discussion of internationalization and users with disabilities), interaction styles (menu selection systems, command languages, direct manipulation), design considerations (including response time, system messages, screen design, and color), and methods to develop and assess the user interface. The book concludes with a provocative discussion of the social and individual impact of user interface design.

New or Expanded Topics Include:

- Virtual/Artificial Reality
- Graphical User Interfaces
- User Interface Management Systems
- Computer Supported Cooperative Work
- Natural Language
- Window Management
- Multimedia and Hypertext
- Dynamic Data Visualization

Designing the User Interface, Second Edition is written for anyone who needs to understand the complex interaction between people and machines. It will benefit people who build interactive systems and people who market them, people interested in systems design and people interested in human performance.

Addison-Wesley Publishing Company

90000
9 780201 572865
ISBN 0-201-57286-9

Designing the User Interface

Strategies for Effective Human–Computer Interaction

Second Edition

Ben Shneiderman
The University of Maryland

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid
San Juan • Milan • Paris

Library of Congress Cataloging-in-Publication Data

Shneiderman, Ben.
Designing the user interface : strategies for effective human
-computer interaction / Ben Shneiderman. -- 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-57286-9
1. Human-computer interaction. 2. User interfaces (Computer
systems) 3. System design. I. Title.
QA76.9.I58S47 1992
004.6--dc20 91-35589
CIP

Chapter opener illustrations © Paul S. Hoffman; chapter opener outlines © Teresa Casey.

Reprinted with corrections October, 1993

5 6 7 8 9 10 DO 959493

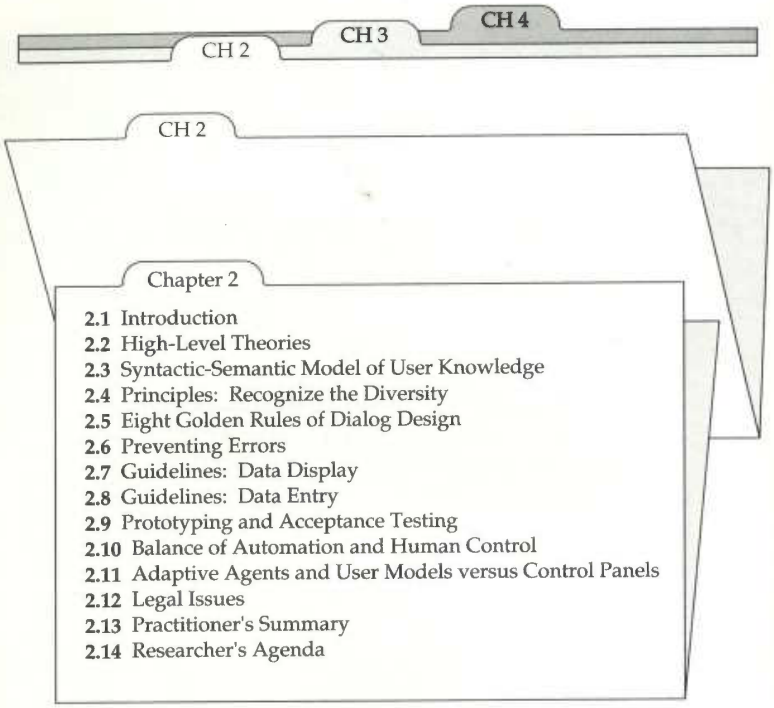

INTERFACE ISLAND
Your guide to avoiding the pitfalls et cetera of this dangerous, tricky and treacherous terrain.
MURKY MTS.
CONSISTENCY FOREST
CLARITY CLEARING
LOGIC TREE

CHAPTER 2

Theories, Principles, and Guidelines

We want principles, not only developed—the work of the closet,—but applied, which is the work of life.

Horace Mann, *Thoughts*, 1867



2.1 Introduction

Successful designers of interactive systems know that they can and must go beyond intuitive judgments made hastily when a design problem emerges. Fortunately, guidance for designers is beginning to emerge in the form of (1) high-level theories or models, (2) middle-level principles, (3) specific and practical guidelines, and (4) strategies for testing. The theories or models offer a framework or language to discuss issues that are application independent, whereas the middle-level principles are useful in weighing more specific design alternatives. The practical guidelines provide helpful reminders of rules uncovered by previous designers. Early prototype evaluation encourages exploration and enables iterative testing and redesign to correct inappropriate decisions. Acceptance testing is the trial-by-fire to determine whether a system is ready for distribution; its presence may be

seen as a challenge, but it is also a gift to designers since it establishes clear measures of success.

In many contemporary systems, there is a grand opportunity to improve the human interface. The cluttered displays, complex and tedious procedures, inadequate command languages, inconsistent sequences of actions, and insufficient informative feedback can generate debilitating stress and anxiety that lead to poor performance, frequent minor and occasional serious errors, and job dissatisfaction.

This chapter begins with a review of several theories, concentrating on the syntactic–semantic object–action model. Section 2.4 then deals with frequency of use, task profiles, and interaction styles. Eight principles of interaction are offered in Section 2.5. Strategies for preventing errors are described in Section 2.6. Specific guidelines for data entry and display appear in Sections 2.7 and 2.8. Testing strategies are introduced in Section 2.9; they are covered in detail in Chapter 13. In Sections 2.10 and 2.11, we address the difficult question of how to balance automation and human control. Section 2.12 covers some legal issues.

2.2 High-Level Theories

Many theories are needed to describe the multiple aspects of interactive systems. Some theories are *explanatory*: They are helpful in observing behavior, describing activity, conceiving of designs, comparing high-level concepts of two designs, and training. Other theories are *predictive*: They enable designers to compare proposed designs for execution time or error rates. Some theories may focus on perceptual or cognitive subtasks (time to find an item on a display, or time to plan the conversion of a bold-faced character to an italic one), whereas others concentrate on motor-task performance times. Motor-task predictions are the most well established and are accurate for predicting keystroking or pointing times (see Fitts' Law, Section 6.3.5). Perceptual theories have been successful in predicting reading times for free text, lists, or formatted displays. Predicting performance on complex cognitive tasks (combinations of subtasks) is especially difficult because of the many strategies that might be employed. The ratio for times to perform complex task between novices and experts or between first-time and frequent users can be as high as 100 to 1. Actually, the contrast is even more dramatic because novices and first-time users often are unable to complete the tasks.

A *taxonomy* is a kind of theory. A taxonomy is the result of someone trying to put order on a complex set of phenomena; for example, a taxonomy might be created for input devices (direct versus indirect, linear versus rotary)

(Card et al., 1990), tasks (structured versus unstructured, controllable versus immutable) (Norman, 1991), personality styles (convergent versus divergent, field dependent versus independent), technical aptitudes (spatial visualization, reasoning) (Egan, 1988), user experience levels (novice, knowledgeable, expert), or user-interfaces styles (menus, form fillin, commands). Taxonomies facilitate useful comparisons, enable education, guide designers, and often indicate opportunities for novel products.

Any theory that could help designers to predict performance even for a limited range of users, tasks, or designs would be a contribution (Card, 1989). For the moment, the field is filled with hundreds of theories competing for attention while being refined by their promoters, extended by critics, and applied by eager and hopeful—but skeptical—designers. This development is healthy for the emerging discipline of human–computer interaction, but it means that practitioners must keep up with the rapid developments, not only in software tools, but also in theories.

Another direction for theoreticians would be to try to predict subjective satisfaction or emotional reactions. Researchers in media and advertising have recognized the difficulty in predicting emotional reactions, so they complement theoretical predictions with their intuitive judgments and extensive market testing. Broader theories of small-group behavior, organizational dynamics, sociology of knowledge, and technology adoption may prove to be useful. Similarly, the methods of anthropology or social psychology may be helpful in understanding and overcoming barriers to new technology and resistance to change.

There may be "nothing so practical as a good theory," but coming up with an effective theory is often difficult. By definition, a theory, taxonomy, or model is an abstraction of reality and therefore must be incomplete. However, a good theory should at least be understandable, produce similar conclusions for all who use it, and help to solve specific practical problems.

2.2.1 Conceptual, semantic, syntactic, and lexical model

An appealing and easily comprehensible model is the four-level approach that Foley and van Dam developed in the late 1970s (Foley and Wallace, 1974):

1. The *conceptual level* is the user's mental model of the interactive system. Two conceptual models for text editing are line editors and screen editors.
2. The *semantic level* describes the meanings conveyed by the user's command input and by the computer's output display.
3. The *syntactic level* defines how the units (words) that convey semantics are assembled into a complete sentence that instructs the computer to perform a certain task.
4. The *lexical level* deals with device dependencies and with the precise mechanisms by which a user specifies the syntax.

This approach is convenient for designers because its top-down nature is easy to explain, matches the software architecture, and allows for useful modularity during design. Designers are expected to move from conceptual to lexical, and to record carefully the mappings between levels.

2.2.2 GOMS and the keystroke-level model

Card, Moran, and Newell (1980, 1983) proposed the *goals, operators, methods, and selection rules* (GOMS) model and the *keystroke-level model*. They postulated that users formulate goals (edit document) and subgoals (insert word) that they achieve by using methods or procedures for accomplishing each goal (move cursor to desired location by following a sequence of arrow keys). The operators are "elementary perceptual, motor, or cognitive acts, whose execution is necessary to change any aspect of the user's mental state or to affect the task environment" (Card, et al. 1983, p. 144) (press up-arrow key, move hand to mouse, recall file name, verify cursor is at end-of-file). The selection rules are the control structures for choosing among the several methods available for accomplishing a goal (delete by repeated backspace versus delete by placing markers at beginning and end of region and pressing delete button).

The keystroke-level model is an attempt to predict performance times for error-free expert performance of tasks by summing up the time for keystroking, pointing, homing, drawing, thinking, and waiting for the system to respond. These models concentrate on expert users and error-free performance, with less emphasis on learning, problem solving, error handling, subjective satisfaction, and retention.

Kieras and Polson (1985) built on the GOMS approach, and used production rules to describe the conditions and actions in an interactive text editor. The number and complexity of production rules gave accurate predictions of learning and performance times for five text-editing operations: insert, delete, copy, move, and transpose. Other strategies for modeling interactive-system usage involve transition diagrams (Kieras and Polson, 1985) (Figure 2.1). These diagrams are helpful during design, for instruction, and as a predictor of learning time, performance time, and errors.

Kieras (1988), however, complains that the Card, Moran, and Newell presentation "does not explain in any detail how the notation works, and it seems somewhat clumsy to use. Furthermore, the notation has only a weak connection to the underlying cognitive theory." Kieras offers a refinement with his *Natural GOMS Language* (NGOMSL) and an analysis method for writing down GOMS models. He tries to clarify the situations in which the GOMS task analyst must make a *judgment call*, must make assumptions about how users view the system, must bypass a complex hard-to-analyze task (choosing wording of a sentence, finding a bug in a program), or must check for consistency. Applying NGOMSL to guide the process of creating

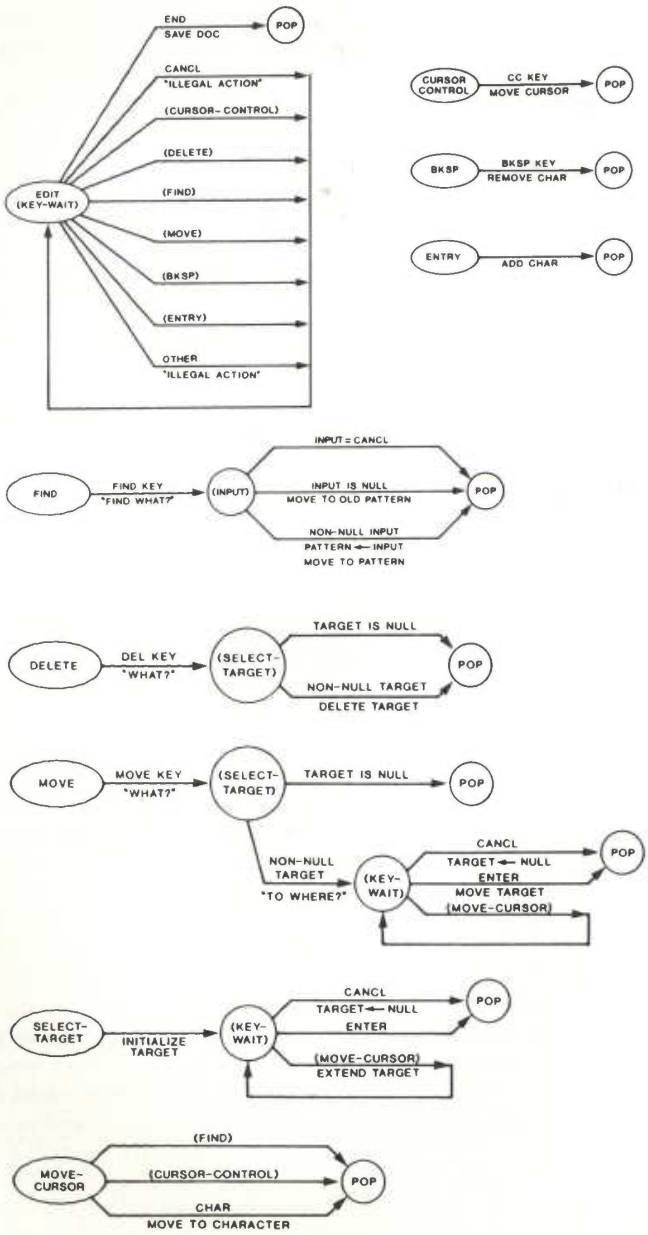

END
SAVE DOC
POP
CANCL
"ILLEGAL ACTION"
(CURSOR-CONTROL)
(DELETE)
(FIND)
(MOVE)
(BKSP)
(ENTRY)
OTHER
"ILLEGAL ACTION"
EDIT
(KEY-WAIT)
CURSOR CONTROL
CC KEY
MOVE CURSOR
BKSP
BKSP KEY
REMOVE CHAR
ENTRY
ADD CHAR
FIND
FIND KEY
"FIND WHAT?"
(INPUT)
INPUT = CANCL
INPUT IS NULL
MOVE TO OLD PATTERN
NON-NULL INPUT
PATTERN ← INPUT
MOVE TO PATTERN
DELETE
DEL KEY
"WHAT?"
(SELECT-TARGET)
TARGET IS NULL
NON-NULL TARGET
DELETE TARGET
MOVE
MOVE KEY
"WHAT?"
NON-NULL TARGET
"TO WHERE?"
(KEY-WAIT)
CANCL
TARGET ← NULL
ENTER
MOVE TARGET
(MOVE-CURSOR)
SELECT-TARGET
INITIALIZE TARGET
(MOVE-CURSOR)
EXTEND TARGET
MOVE-CURSOR
(FIND)
(CURSOR-CONTROL)
CHAR
MOVE TO CHARACTER

Figure 2.1

This generalized transition network for the Displaywriter shows the sequence of permissible actions. If the users begin at the EDIT state and issue a FIND command, they follow the paths in the FIND subdiagram. (Kieras, David and Polson, Peter, "An approach to the formal analysis of user complexity," *International Journal of Man-Machine Studies* 22 (1985), 365–394. Used by permission of Academic Press, Inc. [London] Limited.)

online help, Elkerton and Palmiter (1991) created *method descriptions*, in which the actions necessary to accomplish a goal are broken down into steps. They also developed *selection rules*, by which a user can choose among alternative methods. For example, there may be several methods to delete fields.

Method to accomplish the goal of deleting the field:
- Step 1: Decide: If necessary, then accomplish the goal of selecting the field
- Step 2: Accomplish the goal of using a specific field delete method
- Step 3: Report goal accomplished

Method to accomplish the goal of deleting the field:
- Step 1: Decide: If necessary, then use the Browse tool to go to the card with the field
- Step 2: Choose the field tool in the Tools menu
- Step 3: Note that the fields on the card an background are displayed
- Step 4: Click on the field to be selected
- Step 5: Report goal accomplished

Selection rule set for goal of using a specific field delete method
- If you may want to paste the field somewhere else, then choose "Cut Field" from the Edit menu
- If you want to permanently delete the field, then choose "Clear Field" from the Edit menu

Report goal accomplished.

The empirical evaluation with 28 subjects demonstrated that the NGOMSL version of help halved the time users took to complete information searches in the first of four trial blocks.

2.2.3 Seven stages of action

Norman (1988) offers *seven stages of action* as a model of human–computer interaction:

1. Forming the goal
2. Forming the intention

3. Specifying the action
4. Executing the action
5. Perceiving the system state
6. Interpreting the system state
7. Evaluating the outcome

Some of Norman's stages correspond roughly to Foley and van Dam's separation of concerns; that is, the user forms a conceptual intention, reformulates it into the semantics of several commands, constructs the required syntax, and eventually produces the lexical token by the action of moving the mouse to select a point on the screen. Norman makes a contribution by placing his stages in the context of *cycles of action* and *evaluation*. This dynamic process of action distinguishes Norman's approach from the other models, which deal mainly with the knowledge that must be in the users's mind. Furthermore, the seven-stages model leads naturally to identification of the *gulf of execution* (the mismatch between the users's intentions and the allowable actions) and the *gulf of evaluation* (the mismatch between the system's representation and the users' expectations).

This model leads Norman to suggest four principles of good design. First, the state and the action alternatives should be visible. Second, there should be a good conceptual model with a consistent system image. Third, the interface should include good mappings that reveal the relationships between stages. Fourth, the user should receive continuous feedback. Norman places a heavy emphasis on studying errors. He describes how errors often occur in moving from goals to intentions to actions and to executions.

2.2.4 Consistency through grammars

An important goal for designers is a *consistent* user interface. However, the definition of consistency is elusive and has multiple levels that are sometimes in conflict; also, it is sometimes advantageous to be inconsistent. The argument for consistency is that a command language or set of actions should be orderly, predictable, describable by a few rules, and therefore easy to learn and retain. These overlapping concepts are conveyed by an example that shows two kinds of inconsistency (A illustrates lack of any attempt at consistency, and B shows consistency except for a single violation):

Consistent	Inconsistent A	Inconsistent B
delete/insert character	delete/insert character	delete/insert character
delete/insert word	remove/bring word	remove/insert word
delete/insert line	destroy/create line	delete/insert line
delete/insert paragraph	kill/birth paragraph	delete/insert paragraph

Each of the actions in the consistent version is the same, whereas the actions vary for the inconsistent version A. The inconsistent action verbs are all acceptable, but their variety suggests that they will take longer to learn, will cause more errors, will slow down users, and will be harder for users to remember. Inconsistent version B is somehow more malicious because there is a single unpredictable inconsistency that stands out so dramatically that this language is likely to be remembered for its peculiar inconsistency.

To capture these notions, Reisner (1981) proposed an *action grammar* to describe two versions of a graphics-system interface. She demonstrated that the version that had a simpler grammar was easier to learn. Payne and Green (1986) expanded her work by addressing the multiple levels of consistency (lexical, syntactic, and semantic) through a notational structure they call *task–action grammars* (TAGs). They also address some aspects of completeness of a language by trying to characterize a complete set of tasks; for example, *up*, *down*, and *left* comprise an incomplete set of arrow-cursor movement tasks, because *right* is missing. Once the full set of task–action mappings is written down, the grammar of the command language can be tested against it to demonstrate completeness. Of course, a designer might leave out something from the task–action mapping and then the grammar could not be checked accurately, but it does seem useful to have an approach to checking for completeness and consistency. For example, a TAG definition of cursor control would have a dictionary of tasks:

move-cursor-one-character-forward	[Direction = forward, Unit = char]
move-cursor-one-character-backward	[Direction = backward, Unit = char]
move-cursor-one-word-forward	[Direction = forward, Unit = word]
move-cursor-one-word-backward	[Direction = backward, Unit = word]

Then, the high-level rule schemas that describe the syntax of the commands are as follows:

1. task [Direction, Unit] —> symbol [Direction] + letter [Unit]
2. symbol [Direction = forward] —> "CTRL"
3. symbol [Direction = backward] —> "ESC"
4. letter [Unit = word] —> "W"
5. letter [Unit = char] —> "C"

These schemas will generate a consistent grammar:

move cursor one character forward	CTRL-C
move cursor one character backward	ESC-C
move cursor one word forward	CTRL-W
move cursor one word backward	ESC-W

Payne and Green are careful to state that their notation and approach are flexible and extensible, and they provide appealing examples in which their approach sharpened the thinking of designers.

Reisner (1990) extends this work by defining consistency more formally, but Grudin (1989) points out flaws in some arguments for consistency. Certainly, consistency is subtle and has multiple levels; there are conflicting forms of consistency, and sometimes inconsistency is a virtue (for example, to draw attention to a dangerous operation). Nonetheless, understanding consistency is an important goal for designers and researchers.

2.2.5 Widget-level theories

Many of the theories and predictive models that have been developed follow an extreme reductionist approach. It is hard to accept the low level of detail, the precise numbers that are sometimes attached to subtasks, and the assumptions of simple summations of time periods. Furthermore, many of the detailed models take an extremely long time to write, and many judgment calls plus assumptions must be made, so there is little trust that several analysts would come up with the same results.

An alternative approach is to follow the simplifications made in the higher-level UIMSs (Chapter 14). Instead of dealing with individual buttons and fields, why not create a model based on the widgets (interface components) supported in the UIMS? Once a scrolling-list widget was tested to determine user performance as a function of the number of items and the size of the window, then future widget users would have automatic generation of performance prediction. The prediction would have to be derived from some declaration of the task frequencies, but the description of the interface would emerge from the process of designing the interface.

A measure of layout appropriateness (frequently used pairs of widgets should be adjacent, and the left-to-right sequence should be in harmony with the task sequence description) would also be produced to guide the designer in a possible redesign. Estimates of the perceptual and cognitive complexity plus the motor load would be generated automatically (Sears, 1992). As widgets become more sophisticated and more widely used, the investment in determining the complexity of each widget will be amortized over the many designers and projects.

2.3 Syntactic–Semantic Model of User Knowledge

Distinctions between syntax and semantics have long been made by compiler writers who sought to separate out the parsing of input text from the operations that were invoked by the text. Interactive system designers can benefit

from a syntactic–semantic model of user knowledge. In outline, this explanatory model suggests that users have syntactic knowledge about device-dependent details, and semantic knowledge about concepts. The semantic knowledge is separated into task concepts (objects and actions) and computer concepts (objects and actions) (Figure 2.2). A person can be an expert in the computer concepts, but a novice in the task concepts, and vice versa.

The *syntactic–semantic object–action* (SSOA) *model* of user behavior was originated to describe programming (Shneiderman, 1980) and has been applied to database-manipulation facilities (Shneiderman, 1981), as well as to direct manipulation (Shneiderman, 1983).

2.3.1 Syntactic knowledge

When using a computer system, users must maintain a profusion of device-dependent details in their human memory. These low-level syntactic details include the knowledge of which action erases a character (delete, backspace, CTRL-H, rightmost mouse button, crossing gesture, or ESCAPE), which action inserts a new line after the third line of a text file (CTRL-I, INSERT

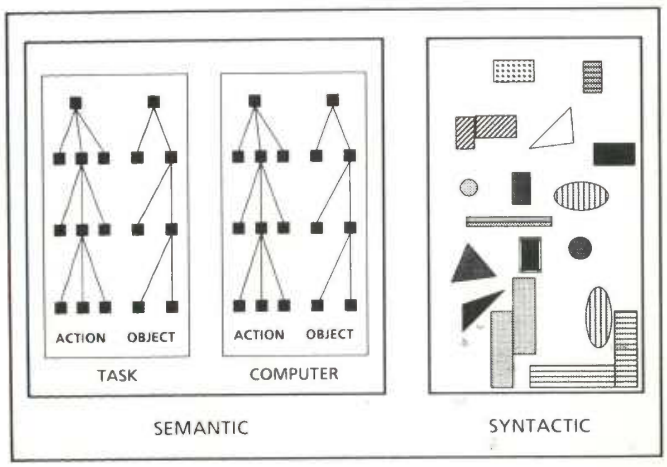


Figure 2.2

Syntactic–semantic model of objects and actions (SSOA model); a representation of the user's knowledge in long-term memory. The syntactic knowledge is varied, device dependent, acquired by rote memorization, and easily forgotten. The semantic knowledge is separated into the computer and task domains. Within these domains, knowledge is divided into actions and objects. Semantic knowledge is structured, device independent, acquired by meaningful learning, and stable in memory.

key, insert gesture between lines three and four, `I3`, `I 3`, or `3I`), which icon scrolls text forward, which abbreviations are permissible, and which of the numbered function keys produces the previous screen.

The learning, use, and retention of this knowledge is hampered by two problems. First, these details vary across systems in an unpredictable manner. Second, acquiring syntactic knowledge is often a struggle because the arbitrariness of these minor design features greatly reduces the effectiveness of paired-associate learning. Rote memorization requires repeated rehearsals to reach competence, and retention over time is poor unless the knowledge is applied frequently. Syntactic knowledge is usually conveyed by example and repeated usage. Formal notations, such as Backus–Naur form, are useful for knowledgeable computer scientists but are confusing to most users.

A further problem with syntactic knowledge, in some cases, lies in the difficulty of providing a hierarchical structure or even a modular structure to cope with the complexity. For example, how is a user to remember these details of using an electronic-mail system: press RETURN to terminate a paragraph, CTRL-D to terminate a letter, `Q` to quit the electronic mail subsystem, and `logout` to terminate the session. The knowledgeable computer user understands these four forms of termination as commands in the context of the full system, but the novice may be confused by four seemingly similar situations that have radically different syntactic forms.

A final difficulty is that syntactic knowledge is system dependent. A user who switches from one machine to another may face different keyboard layouts, commands, function-key usage, and sequences of actions. Certainly, there may be some overlap. For example, arithmetic expressions might be the same in two languages; unfortunately, however, the small differences can be the most annoying. One system uses `K` to keep a file and another uses `K` to kill the file, or `S` to save versus `S` to send.

Expert frequent users can overcome these difficulties, and they are less troubled by syntactic knowledge problems. Novices and knowledgeable users, however, are especially troubled by syntactic irregularities. Their burden can be lightened by use of menus (see Chapter 3), a reduction in the arbitrariness of the keypresses, use of consistent patterns of commands, meaningful command names and labels on keys, and fewer details that must be memorized (see Chapter 4).

In summary, within the SSOA model, syntactic knowledge is arbitrary, system dependent, and ill structured. It must be acquired by rote memorization and repetition. Unless it is used regularly, it fades from memory.

2.3.2 Semantic knowledge—computer concepts

Semantic knowledge in human long-term memory has two components: computer concepts and task concepts (see Figure 2.2). Semantic knowl-

edge has a hierarchical structure ranging from low-level actions to middle-level strategies to high-level goals (Shneiderman, 1980; Card et al., 1983). This presentation enhances the earlier SSOA model and other models by decoupling computer concepts from task concepts. This enhancement accommodates the two most common forms of expertness: task experts who may be novice computer users, and computer experts who may be new to a task. Different training materials are suggested for task or computer experts. Novices in both domains need yet a third form of training.

Semantic knowledge is conveyed by showing examples of use, offering a general theory or pattern, relating the concepts to previous knowledge by analogy, describing a concrete or abstract model, and indicating examples of incorrect use. There is an attraction to showing incorrect use to indicate clearly the bounds of a concept, but there is also a danger, since the learner may confuse correct and incorrect use. Pictures are often helpful in showing the relationships among semantic-knowledge concepts.

Computer concepts include *objects* and *actions* at high and low levels. For example, a central set of *computer-object* concepts deals with storage. Users come to understand the high-level concept that computers store information. The concept of stored information can be refined into the object concepts of the directory and the files of information. In turn, the directory object is refined into a set of directory entries that each have a name, length, date of creation, owner, access control, and so on. Each file is an object that has a lower-level structure consisting of lines, fields, characters, fonts, pointers, binary numbers, and so on.

The *computer actions* also are decomposable into lower-level actions. The high-level actions or goals, such as creating a text data file, may require load, insertion, and save actions. The midlevel action of saving a file is refined into the actions of storing a file and backup file on one of many disks, of applying access-control rights, of overwriting previous versions, of assigning a name to the file, and so on. Then, there are many low-level details about permissible file types or sizes, error conditions such as shortage of storage space, or responses to hardware or software errors. Finally, there is the low-level action of issuing a specific command, carried out by the syntactic detail of pressing the RETURN key.

These computer concepts were designed by highly trained experts in the hope that they were logical, or at least "made sense" to the designers. Unfortunately, the logic may be a complex product of underlying hardware, software, or performance constraints, or it might be just poorly chosen. Users are often confronted with computer concepts that they have great difficulty absorbing; but we have reason to hope that designers are improving and computer-literacy training is raising knowledge levels. For example, the action terminating a command by pressing RETURN is more and more widely known.

Users can learn computer concepts by seeing a demonstration of commands, hearing an explanation of features, or conducting trial-and-error sessions. A common practice is to create a model of concepts—abstract, concrete, or analogical—to convey the computer action. For example, with the file-saving concept, an instructor might draw a picture of a disk drive and a directory to show where the file goes and how the directory references the file. Alternatively, the instructor might make a library analogy or metaphor and describe how the card catalog acts as a directory for books saved in the library.

Since semantic knowledge about computer concepts has a logical structure and since it can be anchored to familiar concepts, we expect it to be relatively stable in memory. If you remember the high-level concept of saving a file, you will be able to conclude that the file must have a name, a size, and a storage location. The linkage to other objects and the potential for a visual presentation support the memorability of this knowledge.

These computer concepts were once novel, and were known to only a small number of scientists, engineers, and data-processing professionals. Now, these concepts are taught at the elementary-school level, argued over during coffee breaks in the office, and exchanged in the aisles of corporate jets. When educators talk of computer literacy, part of their plans cover these computer concepts.

In summary, according to the SSOA model, users must acquire semantic knowledge about computer concepts. These concepts are organized hierarchically, are acquired by meaningful learning or analogy, are independent of the syntactic details, should be transferable across different computer systems, and are relatively stable in memory. Computer concepts can be usefully subdivided into objects and actions.

2.3.3 Semantic knowledge—task concepts

The primary method that people use to deal with large and complex problems is to decompose them into several smaller problems in a hierarchical manner until each subproblem is manageable. Thus, a book is decomposed into the task objects of chapters, the chapters into sections, the sections into paragraphs, and the paragraphs into sentences. Each sentence is approximately one unit of thought for both the author and the reader. Most designed objects have similar decompositions: computer programs, buildings, television sets, cities, paintings, and plays, for example. Some objects are more neatly and easily decomposed than are others; some objects are easier to understand than are others.

Similarly, task actions can be decomposed into smaller actions. A construction plan can be reduced to a series of steps; a baseball game has innings, outs, and pitches; and a business letter comprises an address, date, addressee, body, signature, and so on.

In writing a business letter using computer software, users have to integrate smoothly the three forms of knowledge. They must have the high-level concept of writing (task action) a letter (task object), recognize that the letter will be stored as a file (computer object), and know the details of the `save` command (computer action and syntactic knowledge). Users must be fluent with the middle-level concept of composing a sentence and must recognize the mechanism for beginning, writing, and ending a sentence. Finally, users must know the proper low-level details of spelling each word (task), comprehend the motion of the cursor on the screen (computer concept), and know which keys to press for each letter (syntactic knowledge). The goal of minimizing syntactic knowledge and computer concepts while presenting a visual representation of the task objects and actions is the heart of the direct-manipulation approach to design (see Chapter 5).

Integrating the three forms of knowledge, the objects and actions, and the multiple levels of semantic knowledge is a substantial challenge that requires great motivation and concentration. Educational materials that facilitate the acquisition of this knowledge are difficult to design, especially because of the diversity of background knowledge and motivation levels of typical learners. The SSOA model of user knowledge can provide a guide to educational designers by highlighting the different kinds of knowledge that users need to acquire (see Chapter 12).

Designers of interactive systems can apply the SSOA model to systematize their efforts. Where possible, the semantics of the task objects should be made explicit and the user's task actions should be laid out clearly. Then, the computer objects and actions can be identified, leaving the syntactic details to be worked out later. In this way, designs appear to be more comprehensible to users and more independent of specific hardware.

2.4 Principles: Recognize the Diversity

The remarkable diversity of human abilities, backgrounds, cognitive styles, and personalities challenges the interactive-system designer. When multiplied by the wide range of situations, tasks, and frequencies of use, the set of possibilities becomes enormous. The designer can respond by choosing from a spectrum of interaction styles.

A preschooler playing a graphic computer game is a long way from a reference librarian doing bibliographic searches for anxious and hurried patrons. Similarly, a professional programmer using a new operating system is a long way from a highly trained and experienced air-traffic controller. Finally, a student learning a computer-assisted instruction lesson is a long way from a hotel reservations clerk serving customers for many hours per day.

These sketches highlight the differences in users' background knowledge, training in the use of the system, frequency of use, and goals, as well as in the impact of a user error. No single design could satisfy all these users and situations, so before beginning a design, we must make the characterization of the users and the situation as precise and complete as possible.

2.4.1 Usage profiles

"Know the user" was the first principle in Hansen's (1971) list of user-engineering principles. It is a simple idea, but a difficult goal and, unfortunately, an often-undervalued goal. No one would argue against this principle, but many designers assume that they understand the users and users' tasks. Successful designers are aware that other people learn, think, and solve problems in very different ways (Section 1.5). Some users really do have an easier time with tables than graphs, with words instead of numbers, with slower rather than faster display rates, or with a rigid structure rather than an open-ended form.

It is difficult for most designers to know whether Boolean expressions are too difficult a concept for library patrons at a junior college, fourth graders learning programming, or professional controllers of electric power utilities.

All design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, training, motivation, goals, and personality. There are often several communities of users for a system, so the design effort is multiplied. In addition to these profiles, users might be tested for such skills as comprehension of Boolean expressions, knowledge of set theory, fluency in a foreign language, or skills in human relationships. Other tests might cover such task-specific abilities as knowledge of airport city codes, stockbrokerage terminology, insurance-claims concepts, or map icons.

The process of knowing the user is never ending, because there is so much to know and because the users keep changing. Every step in understanding the users and in recognizing them as individuals whose outlook is different from the designer's own is likely to be a step closer to a successful design.

For example, a generic separation into novice or first-time, knowledgeable intermittent, and expert frequent users might lead to these differing design goals:

Novice or first-time users: The first user community is assumed to have no syntactic knowledge about using the system and probably little semantic knowledge of computer issues. Whereas first-time users know the task semantics, novices have shallow knowledge of the task and both may arrive with anxiety about using computers that inhibits learning. Over-