

## Observing and monitoring usage

Many kinds of evaluation involve some form of monitoring or observation of the way that users interact with a product or a prototype. Observation can take place informally, in the users' workplace or formally in a laboratory.

The ways that data can be collected and analysed include:

### Direct Observation

The observer makes notes or records the user's performance in some way, such as by timing sequences of actions. Observing users may seem easy but it is not as straightforward as it might seem. Not only may you misinterpret what you see, but also the very fact that users know they are being observed will affect the way they perform. This phenomenon is called the **Hawthorne effect** after a 1939 study of workers in a factory in Hawthorne, Illinois.

### Indirect Observation

Direct observation is often a very intrusive method, and evaluators may choose to use indirect observation, with **video recording** of user activity or **automatic keystroke logging**. This needs careful planning to ensure that the cameras are placed correctly and will involve many hours afterwards to view and record the video recordings.

Often used alongside video recording is the use of verbal protocols. **Think aloud** or 'verbal protocol' is a way of getting extra information during an observation or recording of an evaluation. Here, users are encouraged to speak their thoughts as they go about the evaluation task. This allows the evaluator to discover a whole range of information such as:

- How the user has planned to go about the task
- Identification of menu names or icons within a system
- Users' reactions to when things go wrong
- Understanding of error messages produced by the system
- Users' subjective feelings (their attitude) about the activity, from the tone of voice used

Verbal protocols do place a greater strain on the user though, as they have to not just do the task but must also describe what they are thinking about at the same time. Evidence suggests that humans are poor at maintaining this kind of divided attention for more than a few minutes.

Alternatively, protocols may be obtained after the tasks have been completed. These involve the users viewing videos and providing a commentary on what they were trying to do, and are known as post-event protocols.

These techniques often produce large amounts of data which can be very time consuming to analyse. It is estimated that one hour of videotape could take five hours or more to analyse.

## Collecting users' opinions

Not only do we want to examine users' performance but we also want to know what they think about using the technology. If there is some reason for the users not liking the product, then they will not use it. Often, the cause is something trivial (to the designer) but can be a major cause of irritation to the user. Designers use surveys to collect users' opinions, and this is done through questionnaires or interviews with them.

You need to plan interviews carefully so that the questions are relevant to the issues being investigated:

### Structured interviews

These have pre-determined questions that have to be asked in a set way, much like the interviews used in public opinion surveys. This allows a fixed set of responses that can be analysed later by comparing the responses of different subjects – e.g. 20% of users said that they did not like the background colour used.

### Flexible interviews

These will have some set topics but no particular sequence and the interviewer is free to follow up the interviewee's responses and investigate personal attitudes. This kind of interview is good for finding out about opinions on a particular idea, so is often used in requirements gathering.

### Questionnaires

Questionnaires require the construction of clear and unambiguous questions. This is because the respondent may be filling in the questionnaire at a distance from the evaluator (e.g. in a postal questionnaire) and so cannot clarify any misunderstandings on the spot. It is important to carry out at least one pilot questionnaire in order to find and amend any badly designed questions. There are generally two types of question structure in a questionnaire:

- Closed questions where the respondent must choose from a set of alternative replies
- Open questions where the respondent can provide his or her own answer.

Closed questions have some sort of rating scale associated with them. The simplest closed question has the 'Yes/No' type of answer, but a more complex question might allow the respondent to choose from a range of possible choices or rank their answer on a scale of 1 to 5. One variant of this is the Likert scale, where strength of agreement with a statement is measured (for example: *strongly agree*, *agree*, *slightly agree*, *neutral*, *slightly disagree*, *disagree*, *strongly disagree*). Analysing the responses to a questionnaire will involve converting the responses given to a numeric scale and carrying out a statistical analysis on the data.

## Experiments and benchmarking

Traditional experimentation comes from scientific method, which is a process of proposing and verifying a **hypothesis**. The first step of the scientific method is to develop a theory. This can be something from your own observations or a conjecture borrowed from previous research. A hypothesis is formed next. This is a statement that can be tested.

For example, the statement *"Users can read a document faster when a small font is used versus a large font"* is a testable hypothesis. The statement *"Large screens are good"* is not a testable hypothesis. What is meant by good and how would it be measured? The experimenter forms the hypothesis and carries out the experiment.

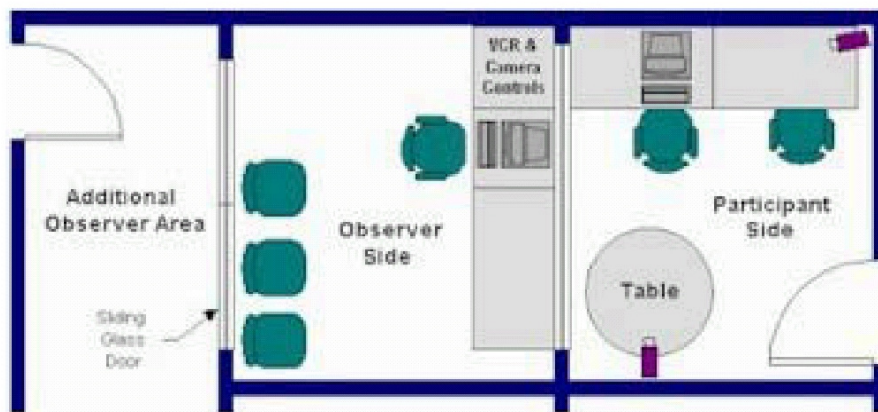
The experimenter has to set up conditions that allow him or her to control certain factors while testing other ones. Specially constructed **usability laboratories** are usually used for this kind of work. Users are observed and their interactions may be logged. The evaluation experiment is rigorously controlled to ensure that reliable data is collected for later analysis.

Only very large organisations can afford to have their own usability laboratories, but there are independent laboratories, which can work for smaller organisations.

When you plan an experiment, you need to think about the following:

- The purpose of the experiment – what is being changed (the independent variable), kept constant and what is being measured (the dependent variable)
- The hypothesis – which needs to be stated in a way that can be tested
- What statistical tests you will apply to the data that you collect

Experiments tend to have a narrow scope – they measure a small number of variables. They tend not to be done during design, and tend to produce detailed findings on small parts of the system.



*Plan of Usability Laboratory at Microsoft*



*A usability experiment in progress at Sun Microsystems*

## **Benchmark tests**

These are used to monitor users' performance on carefully constructed standard tests, usually performed in a usability laboratory. These tests measure **metrics** as laid out in a **usability specification**. Video recording of the test is usually performed, and generally, keystrokes are logged. Typically, these tests will reveal information about the time it takes users to perform tasks, error rates and types of error, usage of instruction manuals etc. The evaluator tries to standardise as much as possible by giving a specified set of users specific tasks to complete in a controlled environment.

## **Interpretive evaluation**

This kind of evaluation enables designers to understand better how users use systems in their natural environments, and how their use of this system will integrate with other activities that the users perform. It is important to try to collect the data in an unobtrusive manner, using informal methods, to try as cause as little disruption as possible to the work going on.

### **Contextual inquiry**

In laboratory benchmarking users have little or no control over the tasks they are performing, as they are simply doing what the evaluators ask them to do. This kind of setting is unnatural and quite different from the usual work environment in several ways. In contextual inquiry, users and researchers participate to identify and understand usability problems within the normal working environment of the user.

### **Cooperative evaluation**

This is one form of interpretive evaluation. Here, users are heavily involved in deciding what the evaluation should actually cover and in analysing the results of the evaluation. Cooperative evaluation is designed to be a low-cost technique that can be used by designers and users without specialist HCI knowledge. Very little training is needed for this. At the start designers identify the evaluation tasks but users are encouraged to comment and suggest appropriate alternatives, and to ask questions. During the evaluation, think aloud protocols are collected and this is followed by a debriefing session and possibly post-test surveys and discussions to check users' opinions.

### **Participative evaluation**

This differs from cooperative evaluation in that it is more open and subject to greater control by users, who are involved in the data collection and analysis process. Often researchers try to immerse themselves in the users' environment, using techniques borrowed from anthropology, where it is common to spend several months or longer living with the group of people being studied.

# Nielsen's Design Principles

Usability guidelines, or heuristics, are rules that distil out the principles of effective user interfaces. There are plenty of sets of guidelines to choose from – sometimes it seems like every usability researcher has their own set of heuristics. Most of these guidelines overlap in important ways, however. The experts don't disagree about what constitutes good UI. They just disagree about how to organize what we know into a small set of operational rules. In these notes, we'll use Jakob Nielsen's 10 heuristic.

Platform-specific guidelines are also important and useful to follow. Platform guidelines tend to be very specific, e.g. you should have a File menu, and there command called Exit on it (not Quit, not Leave, not Go Away). Following platform guidelines ensures consistency among different applications running on the same platform, which is valuable for novice and frequent users alike. However, platform guidelines are relatively limited in scope, offering solutions for only a few of the design decisions in a typical UI.

Heuristics can be used in two ways: during design, to choose among different alternatives; and during evaluation, to find and justify problems in interfaces ('heuristic evaluation').

Let's look at each of Nielsen's 10 heuristics in detail.

## 1. Visibility of system status



The system should always keep users informed about what is going on, through appropriate feedback within reasonable time. Probably the two most important things that users need to know are "Where am I?" and "Where can I go next?"

This heuristic used to be called, simply, "Feedback." Keep the user informed about what's going on. We've developed lots of idioms for feedback in graphical user interfaces. Use them:

- Change the cursor to indicate possible actions (e.g. hand over a hyperlink), modes (e.g. drag/drop), and activity (hourglass).
- Use highlights to show selected objects. Don't leave selections implicit.
- Use the status bar for messages and progress indicators.

But don't overdo it. This dialog box demands a click from the user. Why? Does the interface need a pat on the back for finishing the conversion? It would be better to just skip on and show the resulting documentation.

Depending on how long an operation takes, you may need different amounts of feedback:

- less than 0.1s – seems instantaneous;
- 0.1s to 1s – user notices, but no feedback needed;
- 1s to 5s – display busy cursor;
- over 5s – display progress bar.

## 2. Match between system and the real world



The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order. On the web, you have to be aware that users will probably be coming from diverse backgrounds, so figuring out their "language" can be a challenge.

Nielsen's original name for this heuristic was "Speak the user's language", which is a good slogan to remember. If the user speaks English, then the interface should also speak English, not Geekish. Technical jargon should be avoided. Use of jargon reflects aspects of the system model creeping up into the interface model, unnecessarily. How might a user interpret the dialog box shown here? One poor user actually read type as a verb, and dutifully typed M-I-S-M-A-T-C-H every time this dialog appeared. The user's reaction makes perfect sense when you remember that most computer users do just that, type, all day. But most programmers wouldn't even think of reading the message that way. Yet another example showing that You Are Not The User.

Technical jargon should only be used when it is specific to the application domain and the expected users are domain experts. An interface designed for doctors shouldn't dumb down medical terms.

If an interface allows users to name things, then users should be free to choose long, descriptive names. Artificial limits on length or content should be avoided. DOS used to have a strong limit on filenames, an 8 character name and a 3 character extension. Echoes of these limits persist in Windows even today.

When designing an interface that requires the user to type in commands or search keywords, support as many aliases or synonyms as you can. Different users rarely agree on the same name for an object or command. One study found that the probability that two users would mention the same name was only 7-18%.

Metaphors are one way you can bring the real world into your interface. A well-chosen, well-executed metaphor can be quite effective and appealing, but be aware that metaphors can also mislead. A computer interface must deviate from the metaphor at some point -- otherwise, why aren't you just using the physical object instead? At those deviation points, the metaphor may do more harm than good. For example, it's easy to say "a word processor is like a typewriter," but you shouldn't really use it like a typewriter. Pressing Enter every time the cursor gets close to the right margin, as a typewriter demands, would wreak havoc with the word processor's automatic word-wrapping.

### 3. User control and freedom



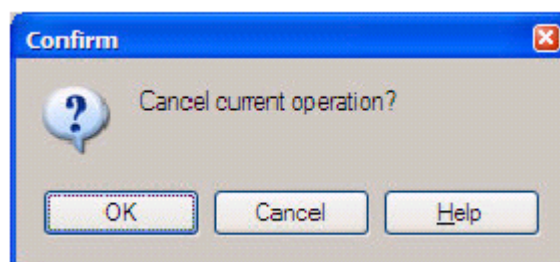
Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo. Some "emergency exits" are provided by the browser, for example the 'Back' button, but there is still plenty of room on your site to support user control and freedom. A "home" button on every page is a simple way to let users feel in control of your site.

This heuristic used to be called "Clearly Marked Exits. Users should not be trapped by the interface. Every dialog box should have a cancel button (where is it in this CuteFTP dialog box?), and long operations should be interruptible.

Users should be able to explore the interface without fear of being trapped in a corner. Undo is a great way to support exploration, but it's not the only way. Editing is important too. If the user is asked to provide any kind of data – whether it's the name of an object, a list of email attachments, or the position of a rectangle – the interface should provide a way to go back and change what the user originally entered – rename the object, add or remove attachments, move around that rectangle some more. Data that is initialized by the user but can never again be touched will frustrate user control and freedom.

Providing user control and freedom can have strong effects on your backend model. You'll have to worry about undo, and you'll have to make sure things are mutable. If you built your backend assuming that a user-provided piece of data would never change once it had been created, you're going to have trouble building a good UI.

### 4. Consistency and standards



Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions. "Platform conventions" on the web means realizing your site is not an island. Users will be jumping onto (and off of) your site from others, so you need to fit in with the rest of the web to some degree. "Standards" on the web means following HTML and other specifications. Deviations from the standards will be opportunities for unusable features to creep into your site.



This rule is often given the name the Principle of Least Surprise, which basically means that you shouldn't surprise the user with the way a command or interface object works. Similar things should look (in terms of size, colour, location, etc.), and act, in similar ways. Conversely, different things should be visibly different.

A very important kind of consistency is in wording. Use the same terms throughout your user interface. If your interface says "share price" in one place, "stock price" in another, and "stock quote" in a third, users will wonder whether these are three different things you're talking about. Conversely, use different terms when you mean different things.

Incidentally, we've only looked at two heuristics, but already we have a contradiction! Matching the Real World argued for synonyms and aliases, so a command language should include not only delete but erase and remove too. But Consistency argues for only one name for each command, or else users will wonder whether these are three different commands that do different things. One way around the impasse is to look at the context in which you're applying the heuristic. When the user is talking, the interface should make a maximum effort to understand the user, allowing synonyms and aliases. When the interface is speaking, it should be consistent, always using the same name to describe the same command or object. What if the interface is smart enough to adapt to the user – should it then favour matching its output to the user's vocabulary (and possibly the user's inconsistency) rather than enforcing its own consistency? Perhaps, but adaptive interfaces are still an active area of research, and not much is known.

Command & argument ordering is another kind of consistency. In **noun-verb** order, the conventional order in graphical user interfaces, the user first selects the object of the command, and then invokes the command. In **verb-noun** order, the command is invoked first, and then the arguments are selected. A drawing program in which some commands were noun-verb and others were verb-noun would be very hard to learn and use.

There are three kinds of consistency you need to worry about: internal consistency within your application external consistency with other applications on the same platform and metaphorical consistency with your interface metaphor or similar real-world objects.

## 5. Error prevention



Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Because of the limitations of HTML forms, inputting information on the web is a common source of errors for users. You can use JavaScript to prevent some errors before users submit.

Since humans make errors if they're given a chance (this is called Murphy's Law: "if something can go wrong, it will"), the best solution is to prevent errors entirely.

One way to prevent errors is to allow users to select rather type. Misspellings then become impossible. This attitude can be taken to an extreme, however, as shown in this example.

If a command is illegal in the current state of the interface – e.g., Copy is impossible if nothing is selected – then the command should be disabled (“grayed out”) so that it simply can’t be selected in the first place.

You can also reduce errors by making sure that dangerous functions (hard to recover from if invoked accidentally) are well-separated from frequently-used commands. Outlook 2003 makes this mistake: when you right-click on an email attachment, you get a menu that mixes common commands (Open, Save As) with less common and less recoverable ones – if you print that big file by mistake, you can’t get the paper back. And if you Remove the attachment, it’s even worse – undo won’t bring it back!

A description error occurs when two actions are very similar. The user intends accidentally substitutes the other. A classic example of a description error is a carton of milk, but instead picking up a carton of orange juice and pouring for pouring milk in cereal and pouring juice in a glass are nearly identical – open carton, open it, pour– but the user’s mental description of the action to execute for the milk.

Description errors can be fought off by applying the converse of the Consistency should look and act different, so that it will be harder to make description errors with very similar descriptions, like long rows of identical switches.

A capture error occurs when a person starts executing one sequence of actions, but then veers off into another (often more familiar) sequence that happened to start the same way. A good mental picture for this is that you’ve developed a mental groove from executing the same sequence of actions repeatedly, and this groove tends to capture other sequences that start the same way. In a computer interface, you can deal with capture errors by avoiding habitual action sequences that have common prefixes.

A third kind of error is a mode error. **Modes** are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands.

Mode errors occur when the user tries to invoke an action that doesn’t have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn’t notice that Caps Lock is enabled, then a mode error occurs.

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. When modes are necessary, it’s essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn’t actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user’s locus of attention. That’s why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn’t really work.

Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do something active to stay in the alternate mode, essentially eliminating the chance that they’ll forget what mode they’re in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another spring-loaded mode; you’re only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette,

that drawing mode is active only for one mouse gesture. Once you've drawn one rectangle, the mode automatically reverts to ordinary pointer selection.

Finally, you can also mitigate the effects of mode errors by designing action sets so that no two modes share any actions. Mode errors may still occur, when the user invokes an action in the wrong mode, but the action can simply be ignored rather than triggering any undesired effect.

## 6. Recognition rather than recall



Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Use menus not command languages, combo boxes not text boxes and generic commands (Open, Save, Copy, Paste) where possible. Instructions for use of the system should be visible or easily retrievable whenever appropriate. For the web, this heuristic is closely related to system status. If users can recognize where they are by looking at the current page, without having to recall their path from the home page, they are less likely to get lost.

There's another reason why selection is better than typing – it reduces the user's memory load. "Minimize Memory Load" was the original name for this heuristic, and it drives much of modern user interface design.

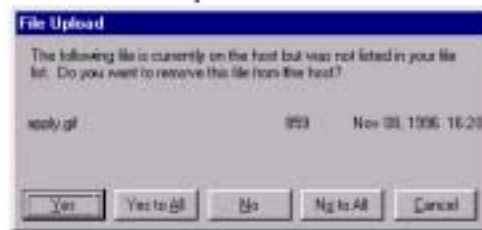
Norman (in *The Design of Everyday Things*) makes a useful distinction between **knowledge in the head**, which is hard to get in there and still harder to recover, and **knowledge in the world**, which is far more accessible. Knowledge in the head is what we usually think of as knowledge and memory. Knowledge in the world, on the other hand, means not just documentation and button labels and signs, but also **nonverbal** features of a system that constrain our actions or remind us of what to do. Affordances, constraints, and feedback are all aspects of knowledge in the world. Command languages demand lots of knowledge in the head, while menus rely on knowledge in the world.

**Generic commands** are polymorphic, working the same way across a wide variety of data objects and applications. Generic commands are powerful because only one command has to be learned and remembered.

Any information needed by a task should be visible or otherwise accessible in the interface for that task. The interface shouldn't depend on users to *remember* the email address they want to send mail to, or the product code for the product they want to buy.

This dialog box is a great example of over-reliance on the user's memory. It's a modal dialog box, so the user can't start following its instructions until after clicking OK. But then the instructions vanish from the screen, and the user is left to struggle to remember them. An obvious solution to this problem would be a button that simply executes the instructions directly! This message is clearly a last-minute patch for a usability problem.

## 7. Flexibility and efficiency of use



Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions. Some of the best accelerators are provided by the browser, such as bookmarks. Make pages at your site easy to bookmark. Do not use frames in a way that prevent users from bookmarking effectively. Design to be linked to. If the contents of your site can easily be linked to, others can create specialized views of your site for specific users and tasks. Amazon.com's associates program is just one example of the value of being easy to link to.

Provide easily learned shortcuts for frequent operations:

- keyboard accelerators
- command abbreviations
- styles
- bookmarks
- history

This heuristic used to be called “Shortcuts.” Frequent users need and want them.

## 8. Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. Make sure your content is written for the web and not just a repackaged brochure. Break information into chunks and use links to connect the relevant chunks so that you can support different uses of your content.

The final heuristic is a catch-all for a number of rules of good graphic design, which really boil down to one word: simplicity. Leave things out unless you have good reason to include them. Don't put more help text on your main window than what's really necessary. Leave out extraneous graphics. Most important, leave out unnecessary features. If a feature is never used, there's no reason for it to complicate your interface.



Google offers a great positive example of the less-is-more philosophy.

## 9. Help users recognize, diagnose, and recover from errors



Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution. For example, if a user's search yields no hits, do not just suggest broadening the search. Provide a link that will broaden the search for the user.

If you can't prevent the error, give a good error message. A good error message should (1) be precise; (2) speak the user's language, avoiding technical terms and details unless explicitly requested; (3) give constructive help; and (4) be polite. The message should be worded to take as much blame as possible away from the user and heap the blame instead on the system. Save the user's face; don't worry about the computer's. The computer doesn't feel it, and in many cases it is the interface's fault anyway for not finding a way to prevent the error in the first place.

The tooltip shown here came from a production version of AutoCad! As the story goes, it was inserted by a programmer as a joke, but somehow never removed before release.

## 10. Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, context-sensitive, focused on the user's task, list concrete steps to be carried out, and not be too large. However, exclusively task-oriented help (which has largely taken over in Microsoft Windows) makes it impossible to get a high-level overview of an interface from the manual. So it's possible to go too far. For the web, the key is to not just slap up some help pages, but to integrate the documentation into your site.

The sad fact about documentation is that most users simply don't read it, at least not before they try the interface. As a result, when they finally do want to look at the manual, it's because they've gotten stuck. Good help should take this into account.

## Analysis by Cognitive Walkthrough

Cognitive Walkthroughs provide a method of analysing designs in terms of *exploratory learning*. They can be applied to designs for systems that will be used by people without any prior training, perhaps in a walk-up-and-use manner. They are also useful in the analysis of systems whose designs have been changed or extended, because these changes and extensions may be encountered by users who have never been taught how to use them. In either case, the user must learn how to use the system by exploring its user interface. Cognitive Walkthroughs answer questions of the form, 'How successfully does this design guide the unfamiliar user through the performance of the task?'

We tend to use Cognitive Walkthroughs, then, when we want to assess operation by users who are exploring the system's user interface and learning as they go. Particular measures we will be looking for concern users' success rates in completing tasks, and their ability to recover from errors. We should not expect to measure users' speed of task performance. We need a fairly complete description of the user interface in order to conduct Cognitive Walkthroughs, because we need to cover all possible routes that the user may take. However, we don't need to know the user's sequence of operation, because the analysis itself helps us to discover what the sequence is likely to be.

### The underlying model of exploratory learning

Analysis by Cognitive Walkthrough involves simulating the way users explore and gain familiarity with interactive systems, with the aid of the simple step-by-step model:

- (0) The user starts with a rough plan of what he or she wants to achieve - a task to be performed;
- (1) The user explores the system, via the user interface, looking for actions that might contribute to performing the task;
- (2) The user selects the action whose description or appearance most closely matches what he or she is trying to do;
- (3) The user then interprets the system's response and assesses whether progress has been made towards completing the task.

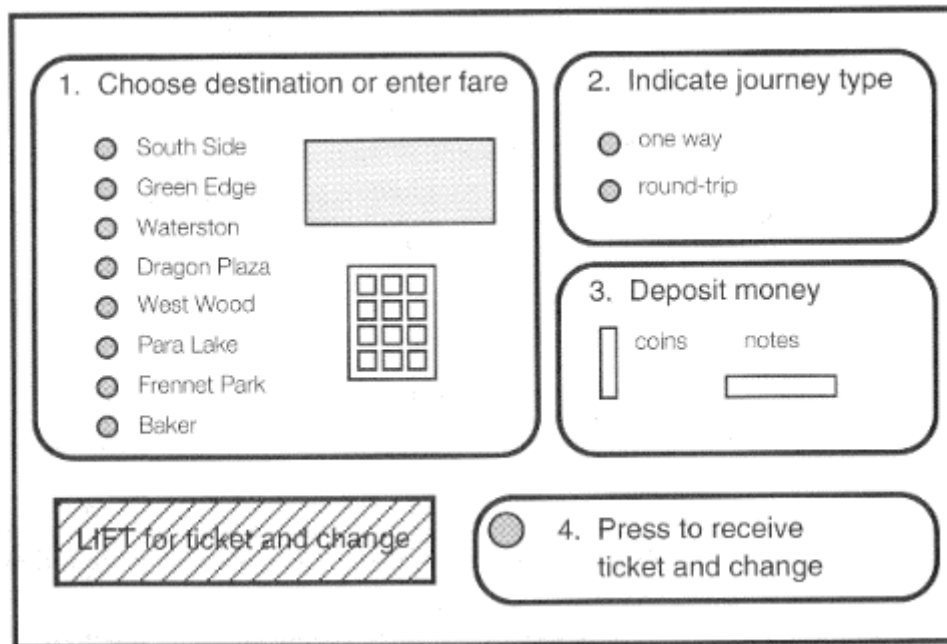
The analysis involves simulating steps 1, 2 and 3 at each stage of interaction, by asking questions of the form:

- Q1: Will the correct action be made sufficiently evident to the user?
- Q2: Will the user connect the correct action's description with what he or she is trying to do?
- Q3: Will the user interpret the system's response to the chosen action correctly, that is, will the user know if he or she has made a right or a wrong choice?

The result of performing a Cognitive Walkthrough is usually to discover problems in these three areas, that is, where the questions receive a 'No' answer. Solutions to these problems are fed into the next iteration of design.

## Cognitive Walkthrough: An example

To illustrate the use of Cognitive Walkthroughs, let us analyse the use of the rapid-transit ticket machine. The preliminary user interface design for the machine is reproduced in **Figure 1**.



**Figure 1.** The design for a rapid-transit ticket machine, to be analysed by Cognitive Walkthrough.

We will work through an analysis of the machine's use by a first-time user. Let us suppose this user wishes to purchase a round-trip ticket to Dragon Plaza. We will add a complication: the traveller has only ten dollars in cash, but doesn't know this at the outset.

Our first step in the walkthrough is to answer the task-definition question:

**Q0:** What does the user want to achieve?

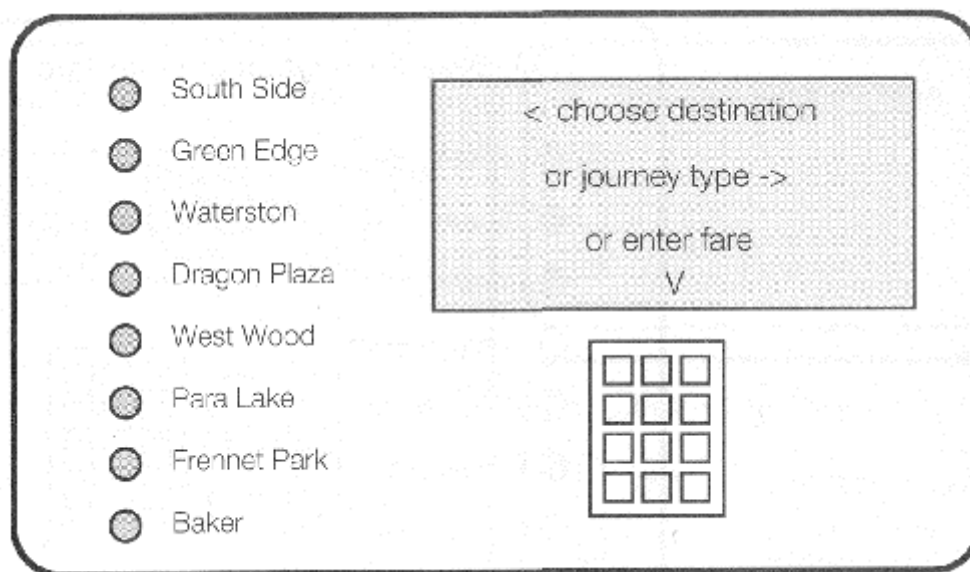
**Answer:** Purchase a round-trip ticket to Dragon Plaza.

Now we can enter into the walkthrough analysis itself. The initial display is as shown in **Figure 1**. We start by asking question Q1 about this display:

**Q1:** Will the correct action be made sufficiently evident to the user?

**Answer:** There are two possible correct actions, press the 'Dragon Plaza' button or press 'round-trip'. The design doesn't make this clear, for it instructs the user to choose the destination before indicating the journey type. This doesn't impede the user's progress, but it hides an available option.

Thus we have identified **Design Flaw no. 1: Option to indicate journey type first is not made sufficiently evident**. One possible way to rectify this flaw might be to provide a prompt via a larger display (see **Figure 2**).



**Figure 2.** *Making the range of methods more obvious to the user*

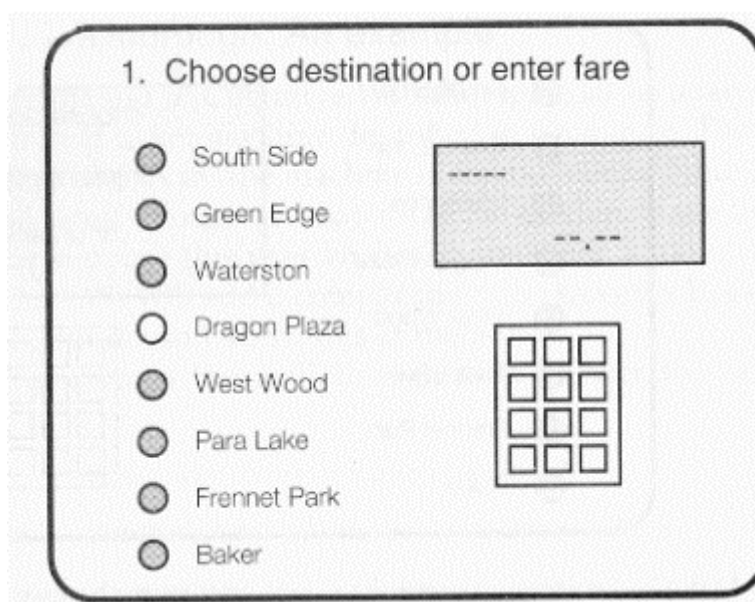
The user is thus likely to be aware of only one correct action, pressing the 'Dragon Plaza' button. Following the walkthrough analysis sequence, we will now ask questions Q2 and Q3 about this action:

**Q2:** Will the user connect the correct action's description with what he or she is trying to do?

**Answer:** Yes, the instructions for panel 1 and the button label will enable the user to make the connection.

**Q3:** Will the user interpret the system's response to the chosen action correctly, that is, will the user know if he or she has made a right or a wrong choice?

**Answer:** The machine will respond by lighting up the button pressed, as shown in **Figure 3**. This should appear to the user as confirmation of a correct action.



**Figure 3.** *Confirming the user's choice of destination with a lighted button.*



The user must now indicate the journey type, using panel 2. We apply the same walkthrough steps as before:

**Q1:** Will the correct action be made sufficiently evident to the user?

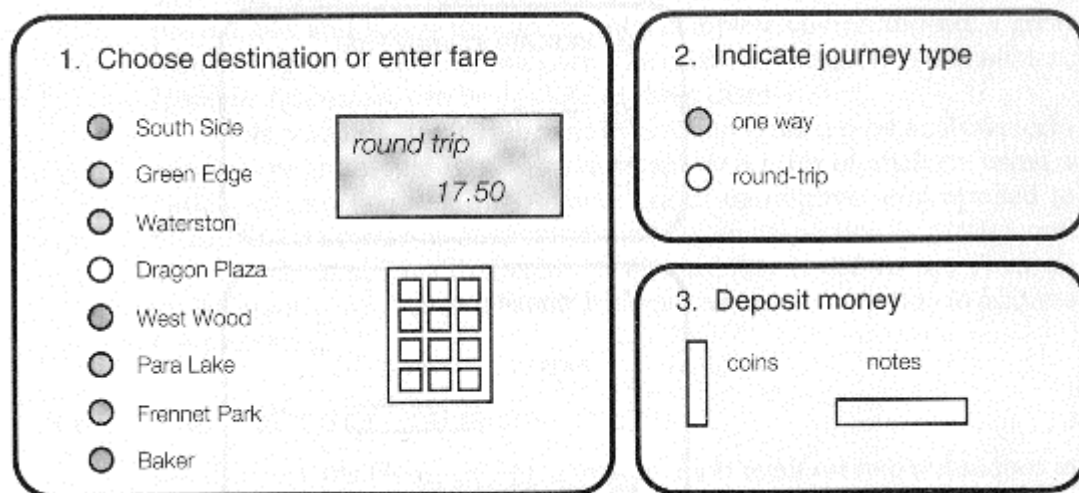
**Answer:** Yes. The correct action is to press the 'round-trip' button in panel 2, and the instructions labelling this panel make it clear that this is the next step.

**Q2:** Will the user connect the correct action's description with what he or she is trying to do?

**Answer:** Yes, the instructions and the labels on the buttons make the connection very clear.

**Q3:** Will the user interpret the system's response to the chosen action correctly, that is, will the user know if he or she has made a right or a wrong choice?

**Answer:** Yes. The machine will respond by lighting up the 'round-trip' button and by displaying the journey type and fare, as shown in **Figure 4**. This provides confirmation of this and the previous action. If the user should press the 'one-way' button by mistake, this too will be made clear.



**Figure 4.** *The machine's response to selecting the journey type.*

We'll now proceed to the next action, depositing money.

**Q1:** Will the correct action be made sufficiently evident to the user?

**Answer:** Yes. Again, the user's main source of assistance is the numbered sequence of instructions. According to this sequence the next step is to deposit money.

**Q2:** Will the user connect the correct action's description with what he or she is trying to do?

**Answer:** Yes, a request to deposit money is consistent with purchasing a ticket.

**Q3:** Will the user interpret the system's response to the chosen action correctly, that is, will the user know if he or she has made a right or a wrong choice?

**Answer:** Unclear. We need to know what kind of response the machine will provide to a correct action, that is, to depositing the first portion of the fare. If the machine merely swallows the money, the user will be little the wiser. According to **Figure 4**, there is no means of indicating receipt of the money, and thus no means for the user to keep track of the amount deposited.

This could be considered **Design Flaw no. 2: No display of total money received**. A solution might be to extend the display as shown in **Figure 5**.

**Figure 5.** Improving the design to show the amount paid so far.

The completion of the walkthrough of the normal ticket-purchase sequence is carried out in a similar fashion. We conclude this example by instead introducing the slight complication we've held in reserve - the passenger who has only ten dollars, and who discovers this only after depositing all of it.

**Q1:** Will the correct action be made sufficiently evident to the user?

**Answer:** No, because there is no action that the user can take to retrieve the money deposited.

We have discovered **Design Flaw no. 3: No means of retrieving money deposited.** The design should include a 'return money' button as shown in **Figure 6.**

**Figure 6.** Adding the capability to retrieve money deposited.

This initial analysis has been very useful, in pointing out three design flaws:

- The option to indicate the journey type before the destination is not made clear.
- No display is provided of the amount of money deposited.
- There is no means to retrieve money deposited.

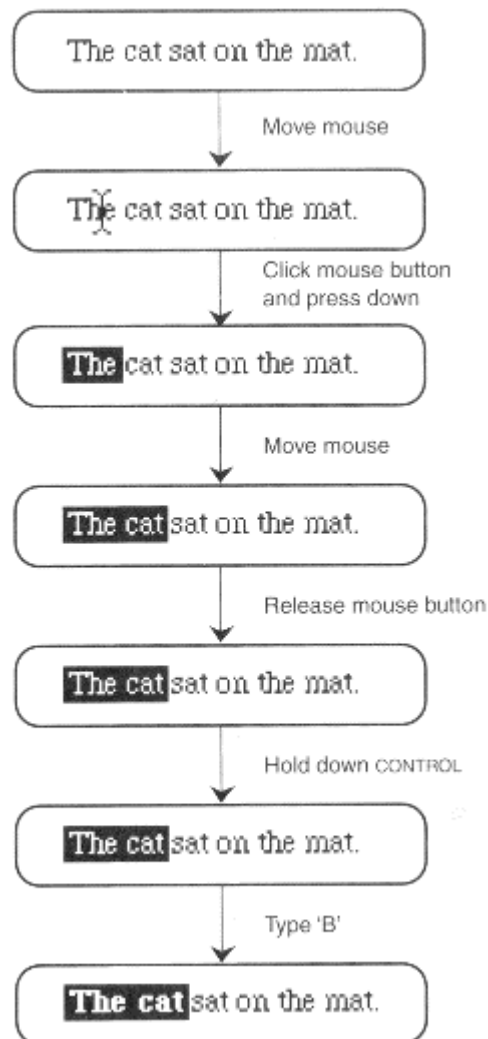
The analysis will not stop there, however. A design such as this will need to be subjected to a full range of walkthrough analyses, covering all of the likely tasks that users will want to perform, under all likely conditions.

## Analysis techniques based on the GOMS model.

### The GOMS model

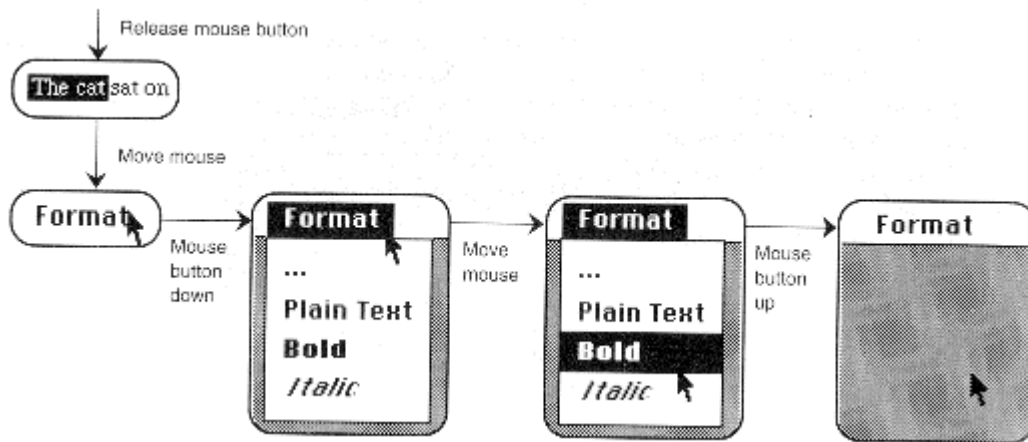
The GOMS model of task performance has four parts. Each part relates to one of the letters in the acronym, which stand for 'Goals, Operators, Methods and Selection rules'. These are the four kinds of variable that affect the outcome of the analyses. Thus:

- The **Goal** defines the end-state that the user is trying to achieve; for example, the goal may be to change to boldface the two words 'The cat' of the sentence 'The cat sat on the mat'. Top-level goals are subdivided into subgoals, and each of these is then analysed separately. Thus we can divide this task into two subtasks: selecting the two words, and setting the selected text to boldface.
- **Operators** are the basic actions available to the user for performing a task, such as moving the mouse pointer, clicking the mouse button, pressing a key.
- **Methods** are sequences of operators, or procedures, for accomplishing a goal. A method for selecting the words 'The cat' is to move the pointer to 'The', hold the mouse button down, move to 'cat' and release the button. A method of setting the selected words to boldface is to hold down the CONTROL key and press 'B'. These two methods are both employed in the sequence shown in **Figure 1**.



**Figure 1.** A method for selecting the words 'The cat' and changing them to boldface.

- **Selection rules** are invoked when there is a choice of method. The user interface doesn't always offer a choice. However, there is an alternative to the boldfacing method shown in the bottom half of **Figure 1**, involving the use of a pull-down menu to select the **Bold** reformatting command, and this is shown in **Figure 2**.



**Figure 2.** An alternative method, using a pull-down menu, for changing the selected words 'The cat' to boldface after selecting them.

GOMS models of task performance can help find problems in a design, for example, *inefficiency* (to achieve certain goals, the user has to apply unnecessarily many or unnecessarily time-consuming operators), *inconsistency* (in some cases, similar goals can only be achieved through dissimilar means), *poor learnability* (the methods for attaining certain goals are hard to discover and/or to remember) and *susceptibility to errors*.

### Example GOMS Analysis of a Text Editing Task

In order to understand GOMS models that have arisen in the last decade and the relationships between them, an analyst must understand ~~each of~~ the components of the model (goals, operators, methods, and selection rules), the concept of level of detail, and the different computational forms that GOMS models take. In this section, we will <sup>define</sup> each of these concepts; in subsequent sections we will categorize existing GOMS models according to these concepts.

*The example task: Editing a marked-up manuscript*

To apply GOMS we generate a task description, pick a high-level user Goal, write a Method for accomplishing the Goal, which may invoke subgoals and write Methods for subgoals. The process is recursive and only stops when Operators are reached. Finally, the GOMS description of the task is evaluated.

```

GOAL: EDIT-MANUSCRIPT
.  GOAL: EDIT-UNIT-TASK ...repeat until no more unit tasks
.  .  GOAL: ACQUIRE UNIT-TASK ...if task not remembered
.  .  .  GOAL: TURN-PAGE ...if at end of manuscript page
.  .  .  GOAL: GET-FROM-MANUSCRIPT
.  .  GOAL: EXECUTE-UNIT-TASK ...if a unit task was found
.  .  .  GOAL: MODIFY-TEXT
.  .  .  .  [select: GOAL: MOVE-TEXT* ...if text is to be moved
.  .  .  .  GOAL: DELETE-PHRASE ...if a phrase is to be deleted
.  .  .  .  GOAL: INSERT-WORD] ...if a word is to be inserted
.  .  .  .  VERIFY-EDIT

```

\*Expansion of MOVE-TEXT goal

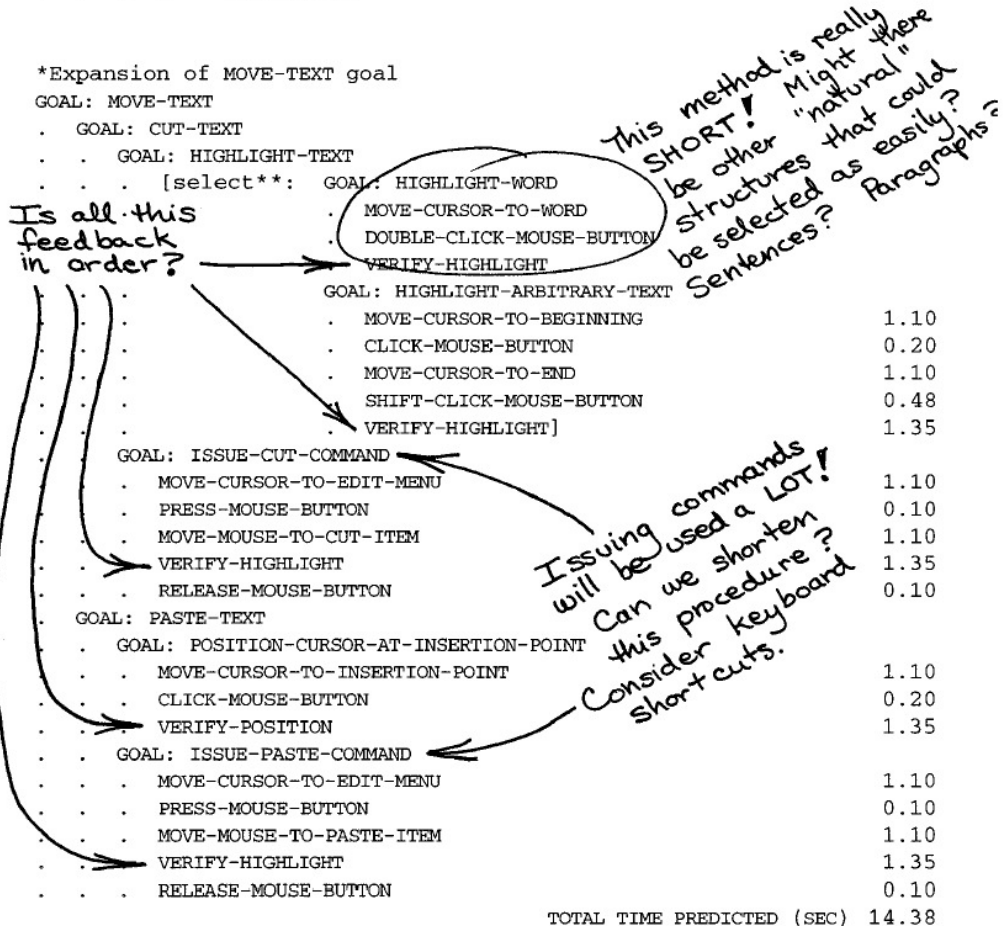
```

GOAL: MOVE-TEXT
.  GOAL: CUT-TEXT
.  .  GOAL: HIGHLIGHT-TEXT
.  .  .  [select**: GOAL: HIGHLIGHT-WORD
.  .  .  .  MOVE-CURSOR-TO-WORD
.  .  .  .  DOUBLE-CLICK-MOUSE-BUTTON
.  .  .  .  VERIFY-HIGHLIGHT
.  .  .  GOAL: HIGHLIGHT-ARBITRARY-TEXT
.  .  .  .  MOVE-CURSOR-TO-BEGINNING
.  .  .  .  CLICK-MOUSE-BUTTON
.  .  .  .  MOVE-CURSOR-TO-END
.  .  .  .  SHIFT-CLICK-MOUSE-BUTTON
.  .  .  .  VERIFY-HIGHLIGHT]
.  .  GOAL: ISSUE-CUT-COMMAND
.  .  .  MOVE-CURSOR-TO-EDIT-MENU
.  .  .  PRESS-MOUSE-BUTTON
.  .  .  MOVE-MOUSE-TO-CUT-ITEM
.  .  .  .  VERIFY-HIGHLIGHT
.  .  .  .  RELEASE-MOUSE-BUTTON
.  .  GOAL: PASTE-TEXT
.  .  .  GOAL: POSITION-CURSOR-AT-INSERTION-POINT
.  .  .  .  MOVE-CURSOR-TO-INSERTION-POINT
.  .  .  .  CLICK-MOUSE-BUTTON
.  .  .  .  VERIFY-POSITION
.  .  .  GOAL: ISSUE-PASTE-COMMAND
.  .  .  .  MOVE-CURSOR-TO-EDIT-MENU
.  .  .  .  PRESS-MOUSE-BUTTON
.  .  .  .  MOVE-MOUSE-TO-PASTE-ITEM
.  .  .  .  VERIFY-HIGHLIGHT
.  .  .  .  RELEASE-MOUSE-BUTTON

```

1.10  
0.20  
1.10  
0.48  
1.35  
1.10  
0.10  
1.10  
1.35  
0.10  
1.10  
0.20  
1.35  
1.10  
0.10  
1.10  
1.35  
0.10

TOTAL TIME PREDICTED (SEC) 14.38



### GOMS Analysis of the text-editing task

However, GOMS models are at their simplest and most effective when predicting speeds of task performance in those cases where the method of operation is known, that is, when selection rules are not an issue. This is done through *Keystroke-Level Analysis*, a technique described in the next section. Other kinds of quantitative analysis concern method selection (by estimating the speed of alternative methods we can compare them, and judge whether the user is likely to prefer one to another) and transfer time (how much time does a user require to learn Method B, once he's already learned the related Method A?)

## Keystroke-Level Analysis

The purpose of the Keystroke-Level Model is to predict the user's speed of execution of tasks. It can be used in situations where the user's method of performing the task is known: for example, it could be applied to dialling a telephone number, since this tends to follow a standard sequence very closely (lift handset, check for dial tone, press buttons in sequence, wait). It could not be applied to the task of drawing a freehand portrait with the aid of a painting program, since everyone does this differently.

The basis of the method is to divide each task-performance method into components, and assign execution times in seconds to each component, as shown in **Table 1**. These times have been derived from repeated experiments, but they can be shown to correspond to predictions derived from models such as Fitts' Law.

Operator	Description and remarks	Time (sec)
<b>K</b>	PRESS KEY OR BUTTON. Pressing the SHIFT or CONTROL key counts as a separate K operation. Time varies with the typing skill of the user; the following shows the range of typical values: Best typist (135 wpm) Good typist (90 wpm) Average skilled typist (55 wpm) Average non-secretary typist (40 wpm) Typing random letters Typing complex codes Worst typist (unfamiliar with keyboard)	          0.08 0.12 0.20 0.28 0.50 0.75 1.20
<b>P</b>	POINT WITH MOUSE TO TARGET ON A DISPLAY. The time to point varies with distance and target size according to Fitt's Law, ranging from .8 to 1.5 sec, with 1.1 being an average. This operator does not include the (.2 sec) button press that often follows. Mouse pointing time is also a good estimate for other efficient analogue pointing devices, such as joysticks.	       1.10
<b>H</b>	HOME HAND(S) ON KEYBOARD OR OTHER DEVICE.	0.40
<b>D</b> ( $n_d, l_d$ )	DRAW $n_d$ STRAIGHT-LINE SEGMENTS OF TOTAL LENGTH $l_d$ cm. This is a very restricted operator; it assumes that drawing is done with the mouse on a system that constrains all lines to fall on a square .56cm grid. Users vary in their drawing skill; the time given is an average value.	       $.9n_d + .16l_d$
<b>M</b>	MENTALLY PREPARE.	1.35
<b>R</b> (t)	RESPONSE BY SYSTEM. Different commands require different response times. The response time is counted only if it causes the user to wait.	    t

**Table 1** Performance times for keystroke-level operators; from Card et al. (1983).

Use of the Keystroke-Level Model involves applying certain rules, or heuristics, in the introduction of **mental preparation** components. These **M**-components should usually, but not always, be introduced before keystrokes or pointing operations that are not part of a sequence. The rules proposed by Card et al. (1983) are given in **Figure 3**.

Begin with a method of encoding that includes all physical operations and response operations. Use Rule 0 to place candidate **M**'s, and then cycle through Rules 1 to 4 for each **M** to see where it should be deleted.

*Rule 0.*

Insert **M**'s in front of all **K**'s that are not part of text or numeric argument strings proper (e.g., text or numbers). Place **M**'s in front of all **P**'s that select commands (not arguments).

*Rule 1.*

If an operator following an **M** is *fully anticipated* in an operator just previous to **M**, then delete the **M** (e.g., **PMK** - **PK**).

*Rule 2.*

If a string of **MK**'s *belongs to a cognitive unit* (e.g., the name of a command), then delete all **M**'s but the first.

*Rule 3.*

If a **K** is a *redundant terminator* (e.g., the terminator of a command immediately following the terminator of its argument), then delete the **M** in front of it.

*Rule 4.*

If a **K** *terminates a constant string* (e.g., a command name), then delete the **M** in front of it; but if the **K** terminates a variable string (e.g., an argument string) then keep the **M** in front of it.

**Figure 3.** Rules for placing **M** operators. From Card et al. (1983).

Keystroke assignments are simple and quick to apply to sequences of actions, but some practice is needed in order to apply the rules for placing **M** operators correctly.

## A simple keystroke-level comparison

We can use keystroke-level analysis to compare the two methods of selecting a pair of words and setting them to boldface, shown in **Figures 1** and **2** above.

The method of **Figure 1** is analysed in the following table. Here the rules of **Figure 3** have been applied to remove unneeded **M** operators (the number of the rule applied is shown in the right-hand column). A value of 0.50 seconds has been chosen for **K** operators, since these are mostly random keys:

Select words			
Reach for mouse	<b>H</b>	0.40	
Point to word 'The' with mouse	<b>P</b>	1.10	
Double-click and hold down mouse button	<b>K</b>	0.50	(1)
Move mouse to word 'cat'	<b>P</b>	1.10	
Finish selection by releasing mouse button	<b>K</b>	0.50	(1)
Set to boldface:			
Press CONTROL	<b>K</b>	0.50	(2)
Type 'b'	<b>K</b>	0.50	(2)
Release CONTROL	<b>K</b>	0.50	(3)
<b>Total</b>		<b>5.10</b>	<b>secs</b>

Note that no **H** operator is included before the CONTROL-b action, which can be given with the left hand only.

In comparison, the sequence of **Figure 2**, in which the **Format** pull-down menu is used, involves the following operators:

Select words (unchanged from previous example)			
Reach for mouse	<b>H</b>	0.40	
Point to word 'The' with mouse	<b>P</b>	1.10	
Double-click and hold down mouse button	<b>K</b>	0.50	(1)
Move mouse to word 'cat'	<b>P</b>	1.10	
Finish selection by releasing mouse button	<b>K</b>	0.50	(1)
Set to boldface:			
Point to Format menu with mouse	<b>P</b>	1.10	
Press and hold down mouse button	<b>K</b>	0.50	(1)
Move down to Bold	<b>P</b>	1.10	(1)
Release mouse button	<b>K</b>	0.50	(1)
<b>Total</b>		<b>6.80</b>	<b>secs</b>

Thus the use of the keyboard 'short cut' does indeed reduce the task's performance time, by over a second and a half. This is due mainly to avoidance of two **P** operators.