The suite has evolved considerably as research and experience have accumulated [Constantine, 1996e; 1997e; Noble and Constantine, 1997]. Currently, five metrics are included that together cover an assortment of measurements likely to be significant to designers seeking to improve the usability of their software:

1. Essential Efficiency
2. Task Concordance
3. Task Visibility
4. Layout Uniformity
5. Visual Coherence

The first three of these—Essential Efficiency, Task Concordance, and Task Visibility—are procedural or task-sensitive metrics based on essential use cases. These three metrics can be used to measure the quality either of specific parts of the user interface or of complete user interface architectures. The simple structural metric of Layout Uniformity assesses aspects of a single interaction context taken in isolation. The suite is rounded out by a powerful content-sensitive or semantic metric, Visual Coherence, which can be used to evaluate either isolated interaction contexts or complete user interface architectures. With one exception, which will be explained, the metrics are normalized to a range of 0 to 100 so that they can be interpreted like percentages, with 100 meaning your design is perfect or as good as it can get in terms of whatever quality is being measured.

Essential Efficiency

The essential use case narrative is an ideal against which the actual interaction with a given design can be compared. In keeping with the Simplicity Principle, short narratives ought to be realized through designs that support brief, straightforward interaction. **Essential Efficiency** (*EE*) is a simple measure of how closely a given user interface design approximates the ideal expressed in the essential use case model. *EE* is just the ratio of the essential length to the enacted length—that is, the ratio of the number of user steps in the essential use case narrative to the number of **enacted steps** needed to perform the use case with a particular user interface design:

$$EE = 100 \cdot \frac{S_{essential}}{S_{enacted}}$$

The essential use case narrative is an ideal against which the actual interaction enacted with a given design can be compared.

Enacted steps are defined by counting rules that govern what constitutes a single discrete user action. (See sidebar, Counting Steps.)

For example, consider the ATM interface introduced in Chapter 5. For the number of essential steps, we just count the number of lines in the left column of

COUNTING STEPS

Many software metrics in common use, including the venerable and widely used Function Points [Albrecht, 1979; Albrecht and Gaffney, 1983], require analysts to count various kinds of the constituents of problems, programs, or designs. To reduce the latitude of judgment and increase the reliability of calculations, counting rules have been developed to define just what counts as what. For comparisons to be valid, consistent rules must be applied. The suite of user interface design metrics requires being able to consistently count several components of interactions and interfaces, including the number of steps required for carrying out tasks. Like most such counting rules, the rules for counting enacted steps are largely heuristic—pragmatic but not grounded in deep theory.

To compute Essential Efficiency, Task Concordance, or Task Visibility, it is necessary to count the number of steps that must be taken by a user enacting some use case in interaction with a particular user interface. What counts as a step? Is every keystroke a step, or does entering an entire name count as one step? Under the counting rules worked out with our clients, each of the following constitutes a single enacted step in the completion of a task using a software user interface.

Enacted Steps

1. Entering data into one field by continuous typing that is terminated by an enter, a tab, or some other field separator
2. Skipping over an unneeded field or control by tabbing or by means of any other navigation key
3. Selecting a field, an object, or a group of items by clicking, double-clicking, or sweeping with a pointing device
4. Selecting a field, an object, or a group of items with a keystroke or series of connected keystrokes
5. Switching from keyboard to pointing device or from pointing device to keyboard
6. Triggering an action by clicking or double-clicking with a pointing device on a tool, a command button, or some other visual object
7. Selecting a menu or a menu item by a pointing device
8. Triggering an action by typing a shortcut key or key sequence, including activating a menu item through keyboard access keys
9. Dragging and dropping an object with a pointing device

The objective under these counting rules is to capture basic conceptual units of interaction, rather than the strictly operational or manual units, such as the keystroke level of analysis employed in the GOMS method. (See later sidebar, Good Old GOMS.) In an effort to make them more useful for comparison with the abstract steps of essential use case narratives, enacted steps strike a compromise between conceptual and concrete measurement. For the most part, enacted steps reflect what users experience or think of as discrete actions, such as selecting, moving, entering, deleting, and the like. Because completion times for enacted steps can vary tremendously, they are better indicators of relative interaction complexity than of expected task performance times, although there will be some correlation. Where a more precise and reliable estimate of expected times is of interest, a GOMS-type analysis could be carried out. (See sidebar, Good Old GOMS.)

When there is more than one way to enact a given use case, as there typically is, the steps for the shortest and longest enactment can be counted to estimate a range. In general, the longest enactment will be more typical of novice or untrained users, while the shortest sequence can be regarded as a potential performance for expert or highly trained users.

the essential use case narrative, the "User Intention" model. For the enacted use case, we count the number of user actions according to the counting rules established for enacted steps. The essential use case in this ATM example has three steps (identify self, choose, take money), while the enacted use case illustrated involves eight discrete user actions, which means $EE = 37.5\%$ for the existing system. Without changing to exotic new technologies, an improved interface could be

designed that offered the customer the choice of selecting "the usual" to initiate a cash withdrawal from the usual account in the amount usually requested. This would result in only five steps for the enacted use case (insert card, enter PIN, select "the usual," take card, take money) for an Essential Efficiency of EE = 62.5%, a substantial improvement.

Because Essential Efficiency compares enacted steps to the essential narrative, the results are dependent on having a good essential use case model. Sloppy or incomplete modeling can yield numbers that look better than they are. In practice, the use case narratives should be reviewed to see whether additional simplification and generalization are possible before computing Essential Efficiency. Of course, the degree of simplification in the essential narrative will not affect comparisons of alternative designs in terms of EE for the same use cases.

EE can also be computed for a mix of tasks. If the overall efficiency of a design for an entire mix of tasks is of interest, the Essential Efficiencies of the various tasks can be weighted by the probability (expected relative frequency) or relative importance of each task:

$$EE_{weighted} = \sum_{\forall i} p_i \cdot EE_i$$

where

p_i = probability (or weighted importance) of task i
EE_i = essential efficiency for task i

In practice, however, designers are often not in a position to make good estimates of the expected frequencies of the various tasks. For this reason, the average Essential Efficiency for the most common or most important few tasks is often substituted.

Computing the weighted EE is especially useful for considering design tradeoffs. In most cases, simplifying one task or part of the interface will make things more difficult somewhere else. By considering the average EE or the weighted EE, the overall impact of a change can be evaluated.

GOOD OLD GOMS

A mainstay of mainstream usability engineering is an analysis technique known as "GOMS" (Goals, Operators, Methods, and Selections) [Card, Moran, & Newell, 1983]. GOMS is based on the so-called model human processor, a theoretical model of how people carry out cognitive-motor tasks and interact with systems. GOMS analyses involve breaking a task down into very small cognitive and motor steps needed to perform the task with a particular user interface. The total task time can be estimated by adding up measured times for such operations as shifting the eyes from one part of the screen to another, recognizing an icon, moving the hand to the mouse, moving the pointer to a particular spot, and then clicking the mouse button twice. Years of research have established ranges of times for many different basic mental and motor operations, such as choosing among a set of options, remembering a code, clicking on an icon, or typing a character. (See the sidebar, In Theory, in Chapter 3.)

For an excellent overview of GOMS research and application, see Olson and Olson [1990].

TASK CONCORDANCE

Task Concordance (*TC*) is another metric based on use cases that evaluates support of efficiency and simplicity. *TC* is an index of how well the distribution of task difficulty using a particular interface design fits with the expected frequency of the various tasks. Good designs will generally make the more frequent tasks easier. *TC* is computed from the correlation between tasks ranked by anticipated frequency in use and by enacted difficulty. Task Concordance is the exception to the rule that metrics in the suite behave as percentages; *TC* ranges from −100 to +100%. When a design is perfect in terms of Task Concordance—that is, when more frequent tasks are always shorter than less frequent tasks—*TC* = 100%. If the design is basically backwards, with more frequent tasks taking more steps, then *TC* will be negative, with *TC* = −100% for a completely wrong-headed design. *TC* will be 0% or close to it whenever the design is essentially random or unrelated to the tasks to be supported.

Good designs will generally make the more frequent tasks easier. Task Concordance is an index of how well the expected frequencies of tasks match their difficulty.

Although *TC* could be defined in a number of different ways, for simplicity we use the rank-order correlation between task frequency and task length employing a statistic called *Kendall's* τ (Greek tau). To compute τ, it is only necessary to list all the tasks in order of their estimated or expected frequency along with their enacted difficulty. Use cases are compared for difficulty according to the number of enacted steps required for completion.

Ranking tasks by expected frequencies has several advantages over trying to estimate actual frequencies. Absolute frequencies of tasks, as required to compute Layout Appropriateness (See sidebar, Other Yardsticks) or weighted Essential Efficiency, are difficult to estimate prior to implementation. Most analysts find it easier to judge whether one task is likely to be relatively more or less common than another, without regard to the actual numbers. For example, you may know with absolute confidence that initializing a new database will be much rarer than entering a new customer and yet have not the slightest idea of the exact percentage of time either will occur. Rank orderings are also typically more

INFINITE INEFFICIENCY

Math mavens will no doubt have noticed that *EE* does not exactly cover the range of values from 0 to 100%. For incredibly inept interface implementations, it can become very small but cannot reach 0% in practice. One arguable interpretation is that *EE* = 0% means a use case is not possible with a given user interface design; it would take an infinite number of steps.

In practice, *EE* could exceed 100% when a clever and highly efficient design supports a poorly worked out essential use case. We would take *EE* >100 as a call to review and refine the essential narrative, but other designers might prefer to let the impressive if improbable results stand.

dependable than absolute frequency estimates, which often have to be pulled from thin air. Use cases can be ranked by expected frequency using the same kind of simple card-sorting techniques that are used to rank user roles and to identify focal use cases (see Chapter 4).

Absolute frequencies of tasks are difficult to estimate. Prior to implementation, it is easier to judge whether one task is likely to be relatively more or less common than another.

Kendall's τ offers an appealing way to compute Task Concordance because it is a statistic that is simple to understand. In its basic form, it is just the fraction of all pairs of items that are correctly ordered versus incorrectly ordered. More precisely, the formula for Task Concordance is a ratio:

$$TC = 100 \cdot \tau = 100 \cdot \frac{D}{P}$$

where

D = discordance score: number of pairs of tasks ranked in correct order by enacted length less number of pairs out of order
P = number of possible task pairs

If every task has a different difficulty or enacted length, P is simply

$$P = \frac{N(N-1)}{2}$$

where

N = number of tasks being ranked

The formula for Kendall's τ gets considerably more complicated if there are ties in either the ranking by expected frequency or by enacted length. We find it generally better to find some way to break ties than to resort to the more complex calculation, although nearly any good statistics software will do the work for you. Ties in rankings by enacted length can be broken by taking into account differences in the complexity of individual steps or in how they are combined.

For user interface designs with lots of tasks, using a program, such as any standard statistical software package, is recommended, but, for simple problems, *TC* can easily be calculated by hand. Consider a screen with five representative tasks that the analysts figure will probably occur ranked as follows:

Tasks *(ranked in order of descending expected frequency)*	**Enacted Length** *(number of user steps in enacted use case)*
Task A	7
Task E	7
Task B	5
Task D	8
Task C	6

The current design results in the enacted lengths shown in the right-hand column. We start by breaking the tie between the first and second ranked tasks. Let us say that task A is judged to be marginally more difficult to carry out than task E, even though they have the same number of enacted steps.

For each pair of numbers in the right-hand column, we ask whether the pair is in the right order or the wrong order relative to the ranking by expected frequency. For each pair in the right order, we add 1 to the discordance score, D, in the formula for TC; for each pair in the wrong order, we subtract 1. In other words, we compare 7+ to 7, which gives a –1, then 7+ to 5, which gives another –1, then 7+ to 8, which adds 1, and so forth, until we have counted all the comparisons. Adding it all up gives $D = -2$. Then, we find P:

$$P = \frac{5 \cdot 4}{2} = 10$$

$$TC = -20\%$$

The design being evaluated is, all in all, pretty poor since it means the user interface is quite backwards. What if we could improve a bit on the more frequent tasks? We might try to change the design to eliminate a few steps in the two most frequent tasks, only to find that this makes task B, the third ranked, more difficult:

Tasks *(ranked in order of descending expected frequency)*	**Enacted Length** *(number of user steps in enacted use case)*
Task A	4
Task E	6
Task B	7
Task D	8
Task C	6

Computing Task Concordance for the revised design, we get

$$TC = 52.7\%$$

which is probably quite an improvement at the end of the day.

Task Visibility

Task Visibility (TV) is another relatively simple procedural metric based on use cases. It is grounded in the Visibility Principle, the notion that user interfaces should show users exactly what they need to know or need to use to be able to complete a given task. It measures the fit between the visibility of features and the capabilities needed to complete a given task or set of tasks.

Task Visibility is a metric grounded in the Visibility Principle, the notion that user interfaces should show users exactly what they need to know or to use to complete a given task.

Quantifying visibility is a more subtle challenge than it first appears to be, and several revisions have been required to devise a metric that is simple yet reflective of sound design practice. The visibility of user interface features is a matter of degree. Things that are immediately obvious from looking at the current screen are more visible than those you have to open a menu to find, which are more visible than those located in other interaction contexts. The relative importance of visibility also depends on aspects of the task. It is more vital to have immediate access to those things that are always required to complete a use case than those that may or may not be needed for a particular enactment. It may be acceptable, for example, to place features needed to enact an extension use case one level removed on a separate interaction context, as reflected in the rules for deriving content models covered in Chapter 6.

Ultimately, it proved easiest to define Task Visibility in terms of the enacted steps rather than interface features, separating out those steps that use hidden capabilities or that are taken in order to gain access to features. A feature is visible if you can see it when you need it, so enacted steps performed in order to see or gain access to parts of the user interface must reflect reduced task visibility. The formula for Task Visibility is

$$TV = 100 \cdot \left(\frac{1}{S_{total}} \cdot \sum_{\forall i} V_i \right)$$

where

S_{total} = total number of enacted steps to complete use cases
V_i = feature visibility (0 to 1) of enacted step i

The formula for *TV* is expressed as a percent of the total number of steps because longer, more complex tasks may legitimately need to be distributed across more than one interaction context. Task Visibility reaches a maximum of 100% when everything needed for a step is visible directly on the user interface as seen by the user at that step. Visibility would reach 0% for a workable design only under very exceptional circumstances. One example might be a high-security interface for remote access to highly sensitive information. When the user connects, there is no prompt for name, identification, or password; the user must know how to type these, in what order, and with what separators. Successful log-on is indicated only by the cursor's moving down a line, after which the user must type the correct series of commands on the blank screen without prompting or feedback. In such a command-line interface, every enacted step must be accomplished entirely on the basis of "knowledge in the head" [Norman, 1988], without visible cues or prompting.

Visibility would reach 0% only under very exceptional circumstances, such as a high-security interface for remote access to highly sensitive information.

Task Visibility can be evaluated for individual use cases or for extended task scenarios that might incorporate any number of use cases. To calculate *TV*, an essential use case or set of use cases is enacted with a given user interface design. The total number of enacted steps is tallied. For each enacted step, the analyst determines whether the enacted step was performed to gain access to features that were not visible on the user interface as it would appear to the user at that point or whether the step used hidden features not visible on the interface. The counting rules for enacted steps have already been covered in a sidebar; the rules for counting feature visibility are presented in another one. (See sidebar, Visibility Rules.)

Task Visibility takes into account only one side of the concept of WYSIWYN, or What You See Is What You Need. It ignores whether things that are *not* needed are also found on the user interface. In principle, we could reduce Task Visibility whenever unused or unnecessary features are incorporated into the user interface. In practice, this refinement makes sense only if all use cases supported by the system are considered in the calculation, which is more often than not quite impractical.

For an example, consider preparing a slide for an on-screen presentation. The presenter wants an object to enter automatically from the right of the screen when the slide first appears. How does one accomplish this in PowerPoint 97?

Enacted Step	*Type*	*Visibility*
select object	direct	1.0
open **Slide Show** menu	exposing	0.5
open **Custom Animation** dialogue	suspending	0.0
open drop-down list	exposing	0.5
select **Fly From Right**	direct	1.0
click on **Timing** tab	exposing	0.5
set **Automatically** option button	direct	1.0
click **OK** to close dialogue	suspending	0.0
	Total	4.5

Since there are eight steps in this enactment, $TV = 56.25\%$. Other enactments are possible, but Task Visibility varies little. Task Visibility might be improved in several ways. For instance, one can argue that the conditions under which animation takes place and the style of animation are closely related and ought to be found on the same dialogue tab. Animation could also be considered a property of the object, to be made available on a property inspector instead of within a modal dialogue that blocks other interaction until dismissed.

Layout Uniformity

Not every developer who ends up responsible for user interface design necessarily has a graphics designer's eye for layout. **Layout Uniformity** (*LU*) is a structural metric that gives a quick handle on one important aspect of visual layout. It was devised as a more practical and simplified replacement for Layout Complexity. (See sidebar, Other Yardsticks.)

For developers who do not have a graphics designer's eye for layout, Layout Uniformity is a structural metric that gives a quick handle on one important aspect of visual design.

Layout Uniformity measures only selected aspects of the spatial arrangement of interface components without taking into account what those components are or how they are used; it is neither task sensitive nor content sensitive. As the name suggests, this metric assesses the uniformity or regularity of the user interface layout. Layout Uniformity—or *LU*—is based on the rationale that usability is hindered by highly disordered or visually chaotic arrangements. The influence of regularity on usability is probably not terribly large, but it is one factor. Complete regularity is not the goal, however. Too much uniformity not only can look unappealing but also can make it harder for users to distinguish different features and different parts of the interface. We can expect that moderately uniform and orderly layouts are likely to be the easiest to

VISIBILITY RULES

Feature visibility associated with an enacted step is considered to vary from 0 to 1. In practice, enacted steps are classified according to function and method of performance into one of four categories: hidden, exposing, suspending, or direct.

Hidden. Hidden operations draw on the user's internal knowledge of the application and its use apart from any information communicated by the visible user interface. Hidden steps include

- Typing a required code or shortcut in the absence of any visual prompting or cue
- Accessing a feature or features having no visible representation on the user interface (for example, the Windows 95 Task Bar when hidden)
- Any action involving an object or a feature that may be visible but the choice of which is neither obvious nor evident based on visible information on the user interface

Opening a generic context menu by clicking on blank background with the right mouse button or typing a keyboard shortcut without being prompted is an example of a hidden step. Hidden enacted steps are assigned a visibility of 0.

Exposing. An enacted step is exposing if its function is to gain access to or make visible some other needed feature without causing or resulting in a change of interaction context. Exposing actions include

- Opening a drop-down list
- Opening a menu or submenu
- Opening a context menu by right-clicking on some object
- Opening a property sheet dialogue for an object
- Opening an object or drilling down for detail
- Opening or making visible a tool palette
- Opening an attached pane or panel of a dialogue
- Switching to another page or tab of a tabbed dialogue

Exposing actions have an intermediate effect on task visibility and are assigned a visibility of 0.5, unless they are or must be accomplished using hidden features, in which case they are classified as hidden and given a visibility of 0.

Suspending. An enacted step is suspending if its function is to gain access to or make visible some other needed feature and it causes or results in a change of interaction context. Suspending actions include

- Opening a dialogue box
- Closing a dialogue or message box
- Switching to another window
- Switching to or launching another application

Suspending or context-switching actions that occur as the first or last enacted step of extensions or other optional interactions have an intermediate effect on task visibility since they provide access to features that may not be needed in all enactments; they are assigned a visibility of 0.5, unless they are or must be accomplished using hidden features, in which case they are classified as hidden and given a visibility of 0. Context changes that are nonoptional, that are required in most or all enactments, have a strong effect on task visibility; these are assigned a visibility of 0.

Direct. An enacted step is a direct action if it is not hidden, exposing, or suspending. In other words, direct actions are accomplished through visible features whose choice is evident and which do not serve to gain access to or make visible other objects. Examples of direct actions include applying a tool to an object to change it, typing a value into a visible field, or altering the setting of an option button. Direct enacted steps are assigned a visibility of 1.

understand and to use. Layout Uniformity is defined as

$$LU = 100 \cdot \left(1 - \frac{(N_h + N_w + N_t + N_l + N_b + N_r) - M}{6 \cdot N_c - M}\right)$$

where

N_c = total number of visual components on screen, dialogue box, or other interface composite

N_h, N_w, N_t, N_l, N_b, and N_r are, respectively, the number of different heights, widths, top-edge alignments, left-edge alignments, bottom-edge alignments, and right-edge alignments of visual components. M is an adjustment for the minimum number of possible alignments and sizes needed to make the value of LU range from 0 to 100 (note the "ceiling" function, $\lceil\ \rceil$, which means the smallest integer greater than the enclosed value):

$$M = 2 + 2 \cdot \left\lceil 2\sqrt{N_{components}} \right\rceil$$

Layout Uniformity goes up when visual components are lined up with one another and when there are not too many different sizes of components. The role of Layout Uniformity can best be appreciated by example. In Figure 17-1 are three alternative layouts for a dialogue box. The widgets are left blank because Layout Uniformity does not care what the components are or do. For the layout with no consistency in size or position (Figure 17-1a), $LU = 0\%$, as expected; likewise, for the completely uniform layout (Figure 17-1c), $LU = 100\%$. Neither one of these is typical of good user interface designs. The intermediate design (Figure 17-1b) is more typical of real dialogue layouts, with $LU = 82.5\%$, which, in our experience, is quite acceptable.

To compute Layout Uniformity, some rules of thumb are needed for determining what counts as a visual component and how to judge when components are aligned with one another. These counting rules are discussed in the sidebar, Counting Components.

As a structural metric concerned only with appearance, Layout Uniformity should not be given undue weight in evaluating designs. It can, however, be useful to the designer who lacks an eye for layout to know when a visual arrangement

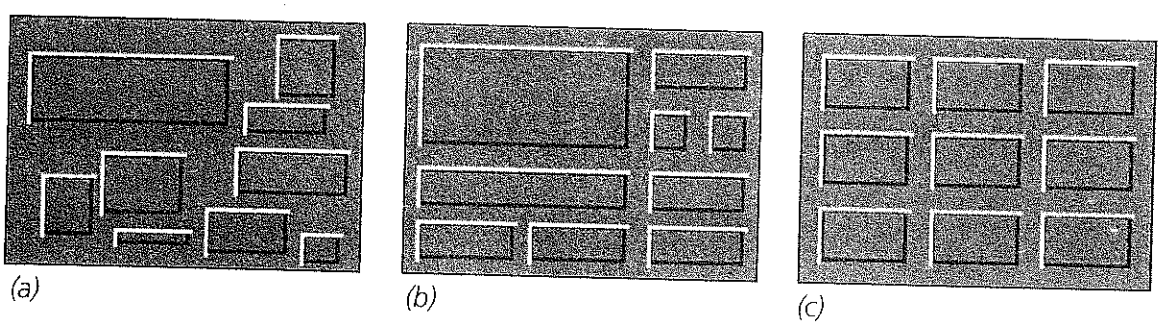

FIGURE 17-1 *Layout Uniformity illustrated.*

Like a good filing system, a well-structured user interface makes it easy to find things because related things are consolidated and unrelated things are separated.

might be improved. A review of well-designed dialogues suggests that, in general, a value of *LU* anywhere between 50% and 85% is probably reasonable, other things being equal. Outside that range, the designer may want to do some thoughtful shuffling of the visual components to make the layout either a little more or a little less uniform.

VISUAL COHERENCE

A well-designed screen or window "hangs together." A good nested set of dialogue boxes collects in one place all those things you think of together and keeps the less related things apart. Like a good filing system, a well-structured user interface makes it easy to find things because related things are consolidated and unrelated things are separated. This is just the Structure Principle in operation. **Visual Coherence** (*VC*) measures how well a user interface keeps related things together and unrelated things apart. More specifically, it is a semantic or content-sensitive measure of how closely an arrangement of visual components matches the semantic relationships among those components. Based on the principle that well-structured interfaces group together components that represent closely related concepts, Visual Coherence reflects important and fundamental aspects of user interface architecture that strongly affect comprehension, learning, and use.

Visual Coherence extends to user interfaces the well-established software engineering metric of cohesion (see sidebar, Cohesion), which gauges how closely interrelated are the contents of software units. A strict application of cohesion to dialogue boxes or other user interface composites would be simplistic since the classic notion of cohesion does not take into account the way component parts are arranged or grouped, only whether they are present or not. Determining which features to place on a given interaction context is, of

COUNTING COMPONENTS

Both Layout Uniformity and Visual Coherence require counting the number of visual components in an interaction context. A visual component is

- Any user interface widget
- An external label not on or embedded in a user interface widget
- A pane, panel, or frame around any one or more widgets or labels

Simple lines separating one part of the visual interface from another are not considered to be visual components in themselves.

Since the arrangement and alignment of components as perceived by the human visual system are of interest, component edges are considered to be aligned if they appear to the unaided eye to be aligned. Text labels, which can vary in height, are a more complicated matter.

Standard practices in user interface layout allow the top edge of text to be slightly below the top edge of an adjacent component and the bottom edge to be slightly above an adjacent bottom edge. Text may also be centered vertically relative to an adjacent component of similar but not identical height. Both edges of text that are aligned according to these conventional practices can be considered to be aligned for purposes of computing Layout Uniformity.

course, one design consideration, but, for the more challenging questions regarding specific layout and visual arrangement of components, the broader notion of Visual Coherence is needed.

For example, the two dialogue box designs shown in Figure 17-2, used in research on Visual Coherence, group visual components very differently by using empty space, lines, boxes, and other visual techniques to define visual groups. The overall Visual Coherence of each design depends on the semantic relatedness among the features or components contained or enclosed within each of its visual groups.

To be able to evaluate Visual Coherence, we have to be able to look at any two visual elements on the interface and determine whether or not they are closely related semantically. We can do this in more than one way, but for now let us imagine we have a table that we can use to determine whether a particular pair of elements are sufficiently closely related to justify putting them in the same visual grouping on the user interface.

COHESION

The classic software engineering notion of cohesion [Constantine, 1968; Yourdon and Constantine, 1978] is a quality metric that gauges the semantic or conceptual interrelatedness of elements within a particular program component or module. The more closely interrelated are the parts, the easier it will be to perceive and understand the collection of parts as a unified whole or gestalt. Cohesion is a form of a complexity (or simplicity) metric in that components high in cohesion are simpler to understand, whether for purposes of construction, use, reuse, extension, or modification. The concept of cohesion has been operationalized in a variety of ways and widely applied in software engineering practice and research. Most recently, it has been extended from its original application within procedure-based programming and traditional structured methods into forms usable in modern object-oriented software engineering practice [Embley and Woodfield, 1987; Chidamber and Kemerer, 1994; Henderson-Sellers, Constantine, and Graham, 1996].

Visual Coherence for any particular visual grouping is simply the ratio of the number of closely related pairs of visual elements to the total number of enclosed pairs. We compute this ratio for the innermost visual groups, and then we just repeat the same thing for the next level outward, until we have covered the complete interaction context. In determining what is related to what at outer levels, we

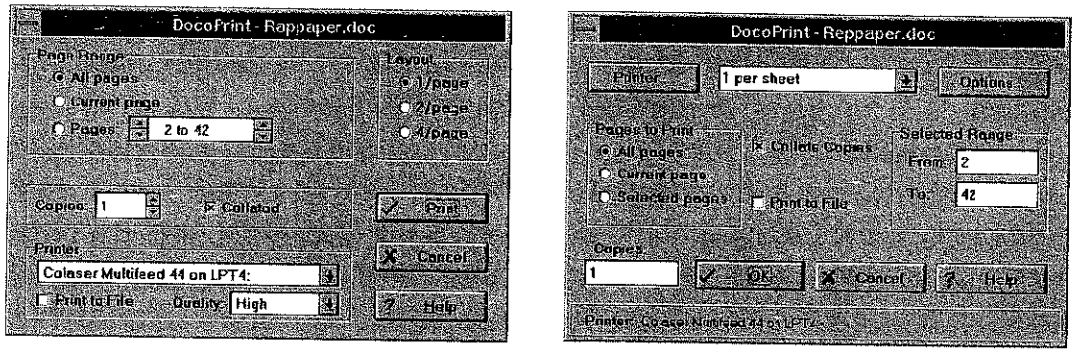

FIGURE 17-2 *Same problem, different Visual Coherence.*

may end up comparing visual groups to other visual groups or to simple visual components grouped at that level.

Total Visual Coherence of a design for an interaction context is computed by summing recursively over all the groups, subgroups, and so forth, at each level of grouping:

$$VC = 100 \cdot \left(\frac{\sum_{\forall k} G_k}{\sum_{\forall k} N_k \cdot (N_k - 1)/2} \right)$$

with

$$G_k = \sum_{\forall i,j \mid i \neq j} R_{i,j}$$

where

N_k = number of visual components in group k
$R_{i,j}$ = semantic relatedness between components i and j in group k,
$0 \leq R_{i,j} \leq 1$

In practice, semantic relatedness can be simplified to just two values: $R_{i,j} = 1$ if components i and j belong to the same semantic cluster and are, therefore, substantially related; $R_{i,j} = 0$, otherwise. This formulation reduces the calculations to counting the substantially related pairs and appears to work quite well in discriminating real designs. It has the correct behavior as a metric in that it favors organizing user interface components into subgroups, but only so long as those groupings make sense—that is, enclosing substantially related components associated with a cluster of closely related semantic concepts.

Semantic clusters, being invisible and intangible, must be discovered; a good starting point is the glossary, domain object model, entity model, or data dictionary for the application.

Semantic clusters, being invisible and intangible, must be discovered by the designers. Fortunately, this process of concept sorting need only be done once for a given project. The starting point is the glossary, domain object model, entity model, or data dictionary for the application. A good domain object model is probably the best starting point since the domain classes, their methods, and their attributes define an overview of the semantic organization associated with a given application [Constantine, 1997e].

Each of the concepts in the problem domain, from whatever source, is written onto a separate index card. The concepts are then sorted into clusters of closely related terms using a card sort or affinity clustering technique such as that

described in Chapter 4. If desired, the clustering task can be completed collaboratively, or several people can complete the task and then discuss and resolve differences in their clusters. Including one or more users among the sorters is desirable. Once the semantic clustering is complete, each cluster can be given a name or heading and the whole collection converted to a list.

To evaluate Visual Coherence, the list of concepts is scanned to determine with which concept each visual component or group is most closely related. If two components or groups are both determined to be most closely related to concepts in the same cluster, then they are considered to be substantially related and are assigned an $R_{i,j} = 1$. If they are associated with concepts from different clusters, then $R_{i,j} = 0$. For a finer-grained measure, components that are closely associated with concepts in separate but related semantic clusters can be assigned an intermediate value for $R_{i,j}$.

Ultimately, it should be possible to derive semantic clusters directly from a complete domain object model, with the strength of semantic relationship based on the nature of the object relationship, such as superclass-subclass, method-of, attribute-of, and so forth. Realization of an object-oriented version of Visual Coherence is under way.

METRICS IN PRACTICE

An important factor in the effectiveness of any metrics initiative is how the numbers are put to use. Utilized inappropriately, metrics can take on an exaggerated significance and may come to dominate design decisions. For example, the effect of immediate feedback on design quality has been investigated at the University of Technology, Sydney [Noble and Constantine, 1997]. Some design metrics can be computed automatically within a visual development tool for user interface layout. Given instant numeric feedback on their layouts, designers can sometimes unconsciously work to maximize their scores rather than derive the best design. The result can be good-looking numbers but poorer interfaces when all factors are taken into account.

The technique of dynamic metric visualization [Noble and Constantine, 1997] was devised to provide "live" feedback during user interface layout without leading the designer astray through a tyranny of numbers. Instead of metrics values displayed numerically or graphically, the designer is shown the underlying basis for the metric of interest. For example, the paths representing enacted use cases can be displayed overlaid on the user interface layout. The designer sees how the lengths of these path lines are affected by the placement of visual components. Tasks that are more frequent are represented by thicker path lines, visually reminding the designer of the relative importance of different use cases. Similarly, the basis of Layout Uniformity can be communicated through light grid lines