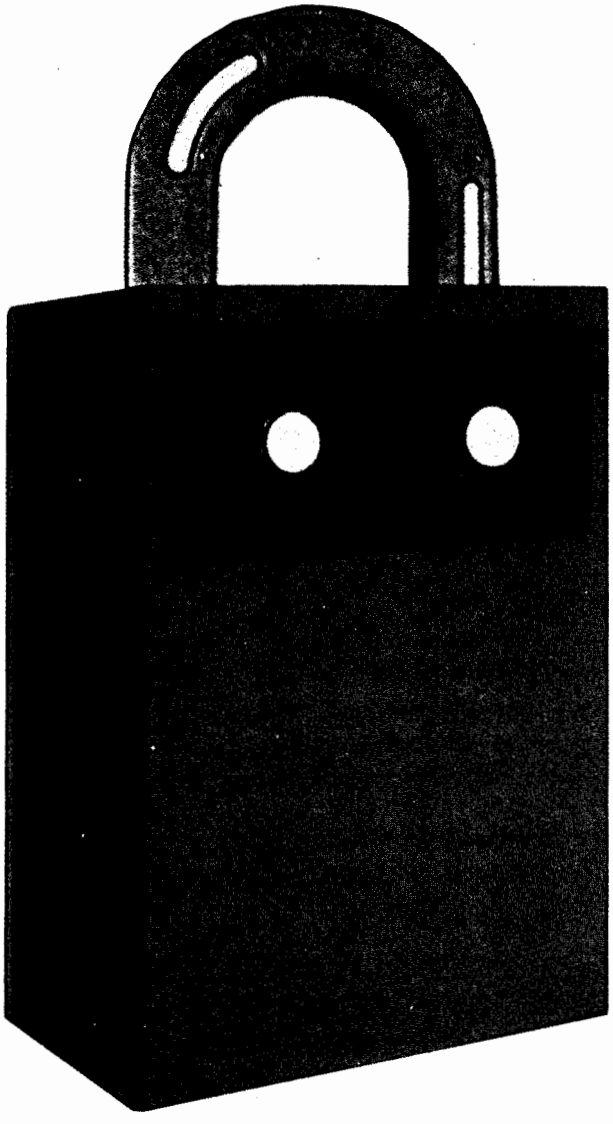
October • 1989

Journal of

Systems Management

Published by the Association for Systems Management



Information Systems And Data Security

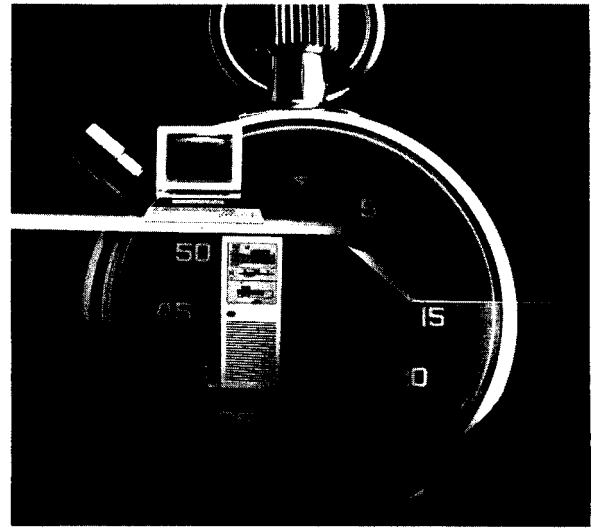
- Computer Security: New Managerial Concern For The 1980s And Beyond
- Computer Fraud in Commercial Banks: Management's Perception Of Risk
- Conceptual Feedback Effects In Information Systems Design
- The Day-One Systems Changeover Tatic
- Is Data Processing A Shepherd Or A Servant?
- Don't Trust The Numbers?

■ The rather old-fashioned phrase "data processing" provides an appropriate starting point for understanding the elements of information systems design. The essence of any information system is that it is given data ("datum" literally means given) which it then transforms to a state ("information") suitable for performing an operation or making a decision.

Data (such as "customer balance" or "product code") are essentially static while processes (such as "calculate net pay") are primarily dynamic. That is why data are commonly described in terms of nouns, while processes are associated with verbs. But data also have dynamic qualities. This becomes clear when we talk of "data flows" within a system. Data flows from one person to another, from a person to a computer system, and even within a computer system from the terminal to the central processing unit. Hence, data may be conceptualized not only statically but also dynamically (such as "transmit customer number").

On the other hand, processes may be conceptualized not only dynamically but also statically. A river is a process, but one can talk about its static and structural aspects such as shape and length. Likewise, an assembly line is a dynamic conversion process, but it also possesses an overall structure. Processes involving data can also be viewed structurally. A long and complex computer program (which is the coded embodiment of a process) can be divided into various sections and subsections that can then be related logically in an overall structure.

Having argued that data and processes each have both static and dynamic qualities, we are led to a four-fold categorization. The following four elements are characteristic of any information system:



The five phases of information systems design cannot be followed in a strictly linear, sequential means.

1. The dynamic aspects of data. This covers forms and computer screens (both used in the flow of data from the user to the system and vice versa).
2. The static aspects of data. This covers the various files or the entire database, as the case may be.
3. The static aspects of processes. This covers the structure charts that show the constituent modules of the process at different levels.
4. The dynamic aspects of processes. This refers to the algorithmic details inside of each module. The most common tools for representation of logic are flowcharts and pseudocode.

To the above four elements we need to add a fifth one, which must be superimposed on the above framework in that it permeates all

of the four elements discussed above.

5. Controls. This refers to the comparison of the actual against a standard and the taking of a corrective action in case there are deviations.

The five elements just discussed constitute the principal elements that must be taken into account in designing an information system. To see how these "elements" become "phrases," we now turn to the following section.

Linearity vs. Interactivity

Conventional wisdom, as manifested in many textbooks on systems design, has it that the process of designing an information system follows a somewhat linear, sequential methodology. The argument runs something like this. The first phase of design — which somewhat overlaps with analysis — is developing screens (or forms) via which the user enters the data into the system and is also provided with information by the system. The overlap with the analysis phase is that this phase is dominated by users' requirements and specifications. It is what the users want.

Having determined the contents of screens (or forms), conventional wisdom leads us to the next phase in which the contents of system-user interfaces (screens/forms) are used as the basis for designing the record layouts in the database. Such a transition (from screen design to database design) is guided by the principles of economy, lack of redundancy, etc. which collectively come under the rubric of "normalization." In a normalized database, we have a storage place for all of the data fields that appear on screens/forms, and each field is stored in an appropriate record type. Now the question is: how does data get transferred back and forth between screens/forms and the database?

The above question leads us logically to the next two phases, where we are concerned with the design of programs (algorithms). Before the program details are developed, a program overview is needed. This happens in phase three, where a structure chart shows the various constituent modules of a program and how they are interrelated. According to conventional wisdom, the fourth phase is then one in which the algorithmic details of each module are developed. These algorithms generally start with a series of READ (input) statements and produce a series of WRITE (output) statements. In between these types of statements are sandwiched a series of data operations governed by the three control structures of sequence, selection and iteration. The data parameters input into (and output from) a module must be exactly the same as those shown on the structure chart, except that on a structure chart the module is treated like a black box in that its algorithmic details are suppressed.

According to the traditional view of information systems design, the final phase has to do with fine-tuning the design by incorporating controls into its elements. Controls are designed into screens in terms of error messages that appear when incorrect data (types) are entered. Controls can be designed into databases via control fields ("flags") that serve special switching (on/off) purposes. Controls enter program structure charts through validation modules and error-message writing modules. They enter algorithm design via program testing techniques.

The sequential framework described above may be represented diagrammatically as in Figure 1. Such a linearity is seldom feasible in the real practice of systems design. In the domain of practical systems design, we are more likely to see the above five elements forming a network and affecting each other interactively. This may be

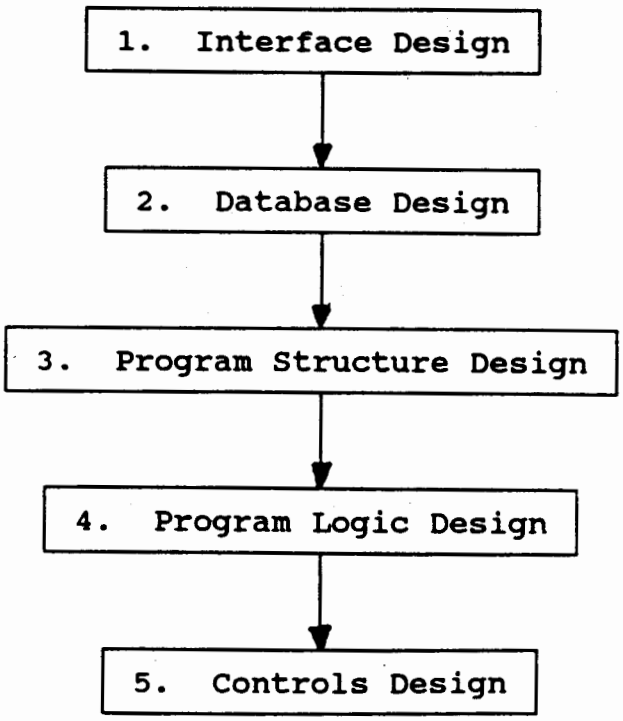


Figure 1

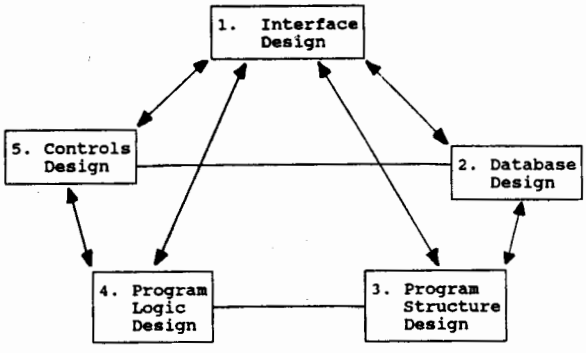


Figure 2

diagrammatically represented as in Figure 2. A clue to the credibility of the above scheme is provided by the manner in which the traditional framework itself treats controls design. Recall that in designing controls, we go *back* to other phases and refine the work already done in those phases. This is essentially the thrust of this article's agrument: that the five phases of information systems design cannot be followed as neatly as a purely sequential pattern would suggest. Design practitioners have found that after supposedly "completing" one phase, it may become necessary to go back to it and revise it in the light of the work done in the subsequent stages.

This iterative principle applies not only to the various substages of the design activity, but also to the major phases of the system development life cycle, of which design is one phase. The other phases are analysis, programming, implementation and maintenance. For instance, traditional wisdom used to dictate that one would complete the analysis phase before proceeding to the design phase. The popularity of prototyping attests to the fact that such linearity is not always ideal. In prototyping, the systems specialist would design a quick and dirty version of the desired system based on the findings in the analysis phase, only to go back to the analysis phase and further complete it according to the users' refined formulations of the requirements after they have worked with the prototype system. It is exactly this type of phenomenon that is referred to as "conceptual feedback effects" in the title of this article. "Feedback" because one phase feeds the earlier (back) phases; "conceptual" because what is under discussion is feedback of a conceptual (rather than personal) nature.

An Illustrative Case

In this section, several examples of feedback effects from a hotel reservations/check-in/check-out system will be presented. The system does not yet exist; rather, we pretend it is in the process of being developed by the designer. This way, we can capture the designer's thought processes in going back and forth between different phases. In what follows, we will discuss six examples of such iterative thinking.

Suppose the hotel policy is that any time a guest (making a reservation) requests a room type that is not available, the more expensive type of room will be assigned, although the guest will be charged the same rate as for the room type he requested. For instance, if he requested a single room and we are going to be out of single rooms during his staying period, then we will assign him a double room while charging him for a single room. When it comes to drawing the structure chart for making reservations, it is natural to have a module "validate room type availability" and another one "validate alternate room type availability." But when we proceed to writing the logic, we realize that both follow an identical logic, except that one takes as input the room type requested, while the other takes as input the more expensive room type requested. So, we decide to write the logic for a room type requested, and if it turns out that the room type requested is not available, then we increment the room type by one (say from single to double) and repeat exactly the same logic. At this time, we find it necessary to go back to the structure chart and combine the two modules into one.

When writing the structure charts, we realize we have too many error-writing modules with very minor differences among them. This may make us go back to the screens and bring more standardization to our error messages. For example, both "incorrect guest number" and "incorrect room type" can be replaced by the more generic "incorrect data entered." Of course, for certain reasons, the analyst and the users may insist on the specifics, in which case this strategy will be avoided.

The check-in screen may not have the check-in date because this date happens to be today's date (= system date). As a result, we may not capture and store it in our database until we need to write logic for computing customer balance.

To compute this amount, we need the duration of stay, which is derived from check-in and check-out dates. How could customer balance be computed if we had not sorted the check-in date? We may be tempted to use the (expected) arrival date (as provided by the guest while making the reservation) to compute the balance. But this is not generally wise for the check-in date, and the (supposed) arrival date may not be identical. Having realized that we need to track the check-in date, we go back to the database design and place it there.

It is necessary to track the total number of rooms of each type in order not to overbook. It is tempting to enter this set of figures in the code at the point where they are actually used. However, due to physical construction, expansion and continual maintenance, this set of numbers changes rather frequently. It is, therefore, much easier to store these fields in the database and change them from there, rather than be forced to go into the program and revise the code. The same argument applies to room rates. When the number of distinct room types is rather small, it is tempting to "hard code" this data. The realization of frequent change may make us, instead, use variables in the code and store the actual room rates in the database.

As another example, imagine a situation in which the (access) key to a record is changed. For instance, when a guest walks in to register, instead of verifying the reservation by name, we may decide to verify it by the confirmation number (which is unique, unlike the guest name). If so, this means we need to change that screen where an error message regarding the key appears. Therefore, instead of "Error: Invalid Name," we will need a message like "Error: Invalid Confirmation Number."

Next, imagine a situation in which there is a check-in record distinct from a check-out record. Each may contain very little data. Therefore, due to the same programming considerations, it may be wise to combine them into a single record type. Meanwhile guest balance, which we may have put in the check-out record, will need to be transferred to a different record type. This is because the new, combined check-in/check-out record type needs to be purged rather frequently, whereas guest balance has a longer useful life. For that reason, it may be advisable to transfer it to the guest file.

When designing the interface, the designer may consider the guest who is to check in as an entity, and devote one line on the form or one blank area on the screen to capture the guest's name. This implies that the name will consist of only one field in the database. However, when the database is designed, and field size of the name is considered, the designer may realize that the option of splitting the name into first name, middle initial, last name and salutation will enhance the design. At this point, the designer must return to the interface design and modify it to reflect four fields where one had previously been acceptable. This change may also happen with address. A hotel system may be enhanced if it allows an extra field to capture the company name of the guest checking in. Inclusion of company name should lead the designer to include company title. Each of these changes requires changes to the interface design and also to the program logic design so that all fields will be captured within the program. While the designer may not consider it at the moment, the resulting database that contains the name separated into four separate fields and an address that includes company name or title becomes a powerful advertising tool because now this data can become the basis for personalized letters and special promotions.

Designing of controls reinforces the iterative nature of designing systems. Technically, validation is not considered until after all phases are completed. If a field must be validated because it is entered in code form, then that fact must be included in the interface design. The database should also reflect this validation; therefore, valid ranges for codes, or related database that contains this data, must be included in the database design and in the interface design.

Validation of fields that require a "yes" or "no" answer brings up a related problem. If the field can only have a yes/no answer, does it make sense to have one of those answers as a default on the screen so that the user can accept that value rather than enter it? If the designer decides to make this change, then the screen design must reflect the default value and the database must also be modified to include this detail. In addition, program logic design must include default values. *(continued on page 37)*

The contrasting management style still supports an interest in the facts and figures. However, this manager treats his/her employees with an open-door policy and with a concern about personal feelings. The employees are dedicated to a company "spirit" and work for the good of the company because the company looks out for their good. The crucial difference, as far as this article is concerned, is that this type of management places a higher priority on honesty and integrity than it does on numbers. If an employee is not an achiever in a certain area, a better fit is searched for; the employee is not automatically "dumped." In other words, success in the company requires honesty, whereas in the first management style there is some question as to whether honesty may not be detrimental to success.

Record Keeping — One "rule" of systems analysis that should be placed in all of the textbooks is the following:

> By making the desired action easier to do than the undesired action, the need for control is eliminated.

This means simply that if you want to track or measure a level of type of performance, do it in a way that doesn't hinder performance. Don't make employees fill out forms when another method, even though more expensive, may be available. To justify this, reflect once again on the goals of the organization. Is the goal to fill out reports or to improve productivity?

> This general principle can be applied to many other areas. For example, if it is easier to be honest (proper and accurate reporting) than to be dishonest (guessing at the numbers because filling out the report is a hassle), the numbers will be meaningful. System and data security is also an area where this line of thinking should be applied.

Summary

These are a few suggestions on what can be done to improve the ethics and thereby the quality of the numbers of an organization:

1. Output, not the generation of numbers, is the goal of the company.
2. Use measurements to motivate, not control.
3. Technology should make the employee's life easier, not put him at risk. Technology should be the employee's tool, not management's toy.
4. Management must be as "good as their word," especially where commitments to employees are concerned. (Parents should always be honest to their children if they want to be trusted.)
5. Plan the installation of new technology with the employees, then controlling it becomes easy.
6. Any change, including the ones suggested here, should start small and simscare rather than motivate if they aren't sold properly.

Conclusion

Controls are not the answer to solving the ethics and data integrity problem. Motivation is the answer. Attitudes need to be changed, not for the employees, but for the managers who design and monitor the control systems under which employees work. The employees simply respond to the controls that are placed upon them, feeling that what they are controlled on must be what is important to the organization. The solution is simple: Management must change the signals that are being sent to their employees. •jsm

(continued from page 10)

Program structure frequently highlights an area that has been neglected in both interface design and database design. This area is archiving of data. Any hotel system will be gathering large amounts of data. How much must be kept? Which data is relevant? Should it be kept only until the bill is paid? The necessity for archiving data requires a new database to be designed. This first modification requires not only a new database but also the decision of which fields are necessary in the archive database. The new database will utilize existing data as input, so no input screens are required, but it is necessary to allow the user to restrict the data being archived. This is a second modification, which affects the interface design. These two modifications then require that the program logic include both a way to archive data and to retrieve the data as desired. An additional step is next required. Once the data has been archived, they must be deleted from any other databases. •jsm