Design of a Self-evolving Decision Support System

TING-PENG LIANG and CHRISTOPHER V. JONES

TING-PENG LIANG is an Assistant Professor in the Department of Accountancy, College of Commerce and Business Administration, University of Illinois at Urbana-Champaign. He received an MBA degree in 1982 from National Sun Yat-sen University, Taiwan, Republic of China, and an M.A. in 1985 and a Ph.D. in information systems in 1986 from The Wharton School, University of Pennsylvania. His research interests include decision support systems, expert systems, model management systems, and implementation issues of information systems.

CHRISTOPHER V. JONES is an Assistant Professor in Decision Sciences, The Wharton School, University of Pennsylvania. He received an M.A. in 1983 and a Ph.D. in 1985 from Cornell University. His research interests include computer graphics, decision support systems, and computer simulation.

ABSTRACT: The paper presents a self-evolving approach to decision support systems (DSS) design. The basic premise of this approach is that a DSS should be aware of how it is being used and, then, automatically adapt to the evolution of its users. With self-evolving capabilities, a DSS will be able to provide a flexible menu hierarchy and a dynamic user interface.

The major difference between the self-evolving design and a DSS developed by current approaches such as system development life cycle and user-involved evolutionary design is that the former has an extra component—the evolutionary mechanism—to control the evolution of the system. In order to develop self-evolving capabilities, the following three components must be developed: (1) a database of user profiles to keep track of related system usage data, (2) a knowledge base to store rules for determining appropriate system default policy, and (3) a control mechanism to control the evolution of the system.

KEY WORDS AND PHRASES: Self-evolving design, decision support systems, information systems development.

Introduction

THE DESIGN AND IMPLEMENTATION STRATEGY is crucial to the success of information systems. Currently two approaches are widely used: system development life cycle (SDLC) and evolutionary design. SDLC focuses on the formal procedure of system design and requires that the system designer determine all relevant

user requirements before a system is actually developed. The evolutionary design, on the other hand, appreciates the evolutionary nature of information systems and the importance of user involvement in the system development process. The designer develops a simple system in a short time and then gradually modifies the system to meet the user's requirements.

Because decision support systems (DSS) are focused on the support of semi-structured or unstructured decisions with ambiguous user requirements, most literature in decision support systems argues that evolutionary design is more appropriate to DSS design since it provides the flexibility needed in the system development process [1, 8, 9, 18, 22, 24]. For example, Alavi and Henderson's research showed that evolutionary design is more effective than SDLC in both system utilization and user satisfaction of DSS [1]. In a recent survey, Watson and his colleagues also reported that evolutionary design was the dominant strategy for developing DSS [24].

The evolutionary approach, based on dividing responsibility for system evolution and development between the user and the designer, can be further divided into two different categories: user-involved and user-developed system design. User-involved evolutionary design is the more popular approach of the two, and in most DSS literature "evolutionary design" usually implies user-involved evolutionary design. The approach places primary responsibility for DSS evolution on the designer, with some help from the user. In the user-developed approach, however, users develop their own DSS with the help of the designer and are responsible for the evolution of the system. Basically, both of these approaches require heavy human involvement in the process of system evolution, although they do provide flexibility in the system development process. The system plays no role except as a passive target to be modified.

The purpose of this article is to present a third approach, the self-evolving system design. It focuses on applying artificial intelligence techniques to develop self-evolving capabilities which allow a DSS to adapt to the evolution of user requirements automatically. A DSS with self-evolving capabilities is called a self-evolving DSS. Examples of self-evolving capabilities include:

(1) a dynamic menu that provides different menu hierarchies to fulfill different user requirements,

(2) a dynamic user interface that provides different output representations for different users, and

(3) a model selection scheme that facilitates the model selection process to satisfy different preferences.

In developing a DSS generator that serves as a DSS development environment, such capabilities are particularly important.

Motivations for developing a self-evolving decision support system are as follows:

1. Increasing the Flexibility of a System. Evolution is one of the most important characteristics of the DSS development process. In order to reduce the burden on the user and the designer for system evolution and to increase the adaptability of the system, some capabilities of self-evolution should be built into a DSS.

2. *Reducing the Effort Required to Use the System.* For most decision makers, learning how to use an information system needs much time and effort. Therefore, it would be more reasonable to have a system adapt to its user rather than to ask the user to adapt his behavior to the system [9].

3. *Enhancing Organizational Control over the Organization's Information Resource.* Because different users usually have different requirements for developing DSS to support their decision making, it is very unlikely that a single DSS will be able to support all users without sacrificing user satisfaction or decision performance. Therefore, many DSS are tailored to meet their users' specific requirements. Developing different DSS for different users, however, may cause two major problems. First, the user may hide the information generated by the DSS, which would jeopardize organizational control over its information resource. Second, the effort invested for system development may be redundant, which is uneconomical. A self-evolving DSS, one which generates different versions of the DSS for different users, will be able to effectively offset these two drawbacks [3, 4, 19].

4. *Encouraging System Sharing.* In a self-evolving DSS, different users may share a system, which can avoid wasting cost and time for developing different DSS for different users.

The key idea of the self-evolving approach presented in this article is that a system can control its evolution by adjusting its default values for performing operations. Its implementation is basically an extension of the current DSS development approaches. Although progress in artificial intelligence has not created any technique that can provide a system with full capability for self-evolution, it is already possible, based on current technology, to develop a system with a partial capability for self-evoluiton [18]. For instance, using techniques that adjust system defaults based on previous and anticipated usage behavior, a self-evolving DSS can adapt to a user's changing behavior.

In the remainder of this article, current approaches, including SDLC, user-involved, and user-developed systems, will be briefly reviewed. Then, the philosophy, architecture, and implementation of the self-evolving design will be discussed. Finally, an example that illustrates the application of the self-evolving approach to DSS design will be described.

Overview of Current Design Approaches

SDLC Approach

The primary objective of the traditional SDLC approach is to structure the development process by offering a specific sequence of procedures and guidelines to the system designer. A typical development cycle includes the following steps:

(1) feasibility study,
(2) system analysis,
(3) logical system design,

(4) physical system design,
(5) documentation,
(6) testing operation and maintenance, and
(7) post audit.

Since the formal specification of the system to be developed and the precise procedure for system development can better organize the development process and increase the efficiency of system development, the SDLC approach has been widely used in the development of many information systems. Unfortunately, for the development of DSS, this approach normally requires a lengthy development time and has limited user involvement [9, 24], which limits its capability to meet the unstructured and changing environment of DSS.

User-involved Evolutionary Approach

A typical user-involved evolutionary design develops a simple system in a short time and then, based on feedback from the user, iteratively refines, expands, and modifies the system through the cycle of analysis–design–implementation–evaluation. In the development process, the system designer and the user design and implement the first version of the system. Then the user evaluates the system in cooperation with the designer. If the system is useful, the user continues to use it. Otherwise, the designer and the user specify additional requirements and modify the system accordingly. After completing the modification, the user evaluates the new version and determines whether it needs further modification. The process continues as needed.

Although the user-involved evolutionary design approach provides both extensive user involvement and the required flexibility and adaptability to the system, it still has the following drawbacks [3, 6]:

1. A large amount of user time is required, and users of DSS are often very busy;
2. A highly talented system designer is required, but such experts are scarce;
3. The DSS may need to be redesigned and reprogrammed for efficiency after several generations of evolution;
4. The process is susceptible to user and implementor turnover; such turnover is high in most companies;
5. The continual change makes the user feel that an outsider—the system designer—is interfering with the decision process; and
6. It is inefficient to develop a DSS with more than two or three users, each with different requirements, because much time is needed to coordinate different requirements.

User-developed System Approach

The premise of the user-developed system approach is that no one knows more about the user's requirements than the user, and, therefore, it is better to have users design their own DSS. In order to apply this approach, a system development environment

that implements Sprague's three-level framework for DSS development is very helpful. DSS are divided into three levels in the framework: specific DSS, DSS generator, and DSS tool [21, 22]. A specific DSS is a DSS that actually supports the user in decision making. For example, a DSS for production scheduling is a specific DSS. A DSS generator is a package of hardware and software that provides a set of capabilities to build specific DSS quickly and easily. For example, spreadsheet software may be considered as a DSS generator because it can support many specific DSS such as cash management systems and inventory control systems. DSS tools are hardware or software elements that facilitate the development of specific DSS or DSS generators. A graphics package is one example of a DSS tool. In the user-developed system approach, users work with a DSS generator to develop their specific DSS for decision making.

Although having users develop their own DSS has been considered promising, it still has disadvantages [2, 4, 16, 19]:

1. It will significantly increase the burden on the system user, whose time is already very limited;

2. The user may not have enough technical expertise to develop a good quality system; even if such development is possible, a long training period may be required;

3. The elimination of the division of labor between the designer and the user may cause inappropriate or inefficient system development;

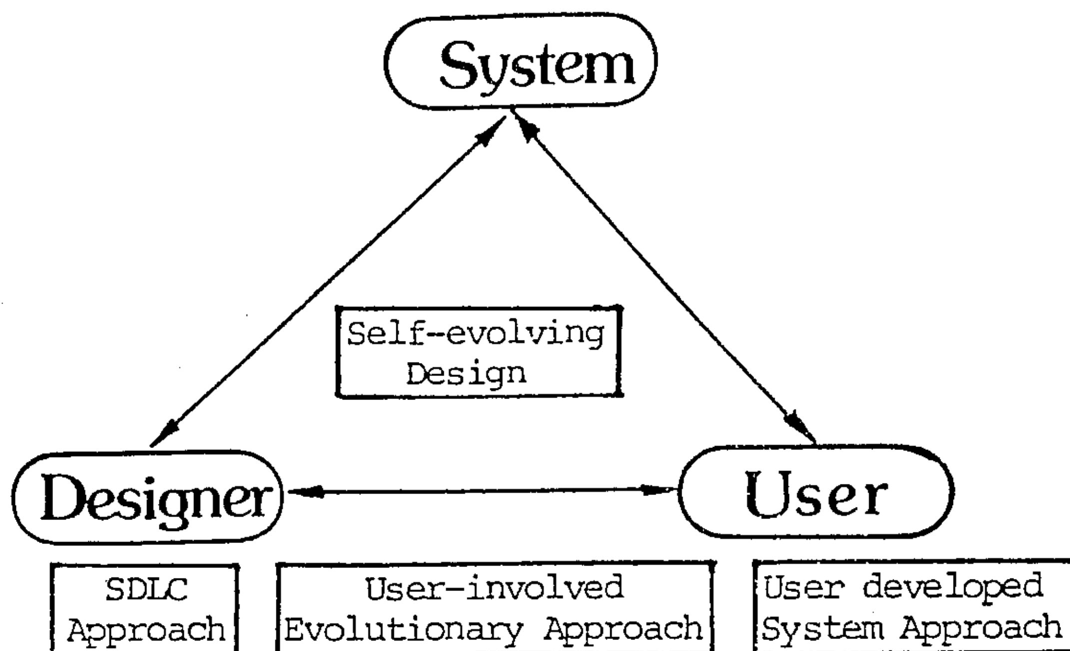
4. The approach may encourage the growth of private information systems, that is, users holding their own information, which would hinder the flow of information within an organization;

5. The approach may cause information control problems in an organization; for example, different systems developed by different users may generate inconsistent information for the same decision; and

6. The system a user can develop is heavily limited by the capability of the employed DSS generator; therefore, the development of a good DSS generator becomes crucial to the success of this approach.

Philosophy of the Self-evolving Design

WITH A BRIEF REVIEW of current design approaches, it is obvious that the scarcity of users' and designers' time is among the major limitations for both the user-involved evolutionary design and the user-developed system approach. In order to effectively offset this limitation, designing a system with self-evolving capabilities bcomes a natural choice. In Figure 1, the philosophies of various approaches are illustrated in terms of the division of labor among the system, the designer, and the user. On the one hand, the user has limited responsibility for design and implementation in the SDLC approach, some responsibility in the user-involved evolutionary approach, and major responsibility in the user-developed system approach. The self-evolving design, on the other hand, requires that the system share the responsibility with the user and the system designer.



Note: SDLC = system development life cycle.

Figure 1. Division of Labor among Various Agents

The self-evolving design assumes that the evolution of a system is not completely unpredictable. Some types of evolution are more structured and are partially predictable, while others are more unstructured and less predictable. The system must adapt to predictable evolution automatically and leave the unpredictable evolution to system designers and users. That is, if the evolution is within the anticipated range, then the system will handle it automatically. Otherwise, a major revision of the system must be undertaken by the system designer and the user to handle the unanticipated evolution. Either the user-involved or the user-developed approach must be used.

The advantages of the self-evolving design are two-fold:

(1) For a Single-User DSS. So long as the evolution is within the range the DSS can handle, the system can automatically adapt to the user's changing requirements, which will significantly increase the flexibility of the system.

(2) For a Multiple-User DSS. The system can identify different users and provide different versions of the DSS for them, which will be able to enhance organizational control over its information resource and facilitate system sharing in an organization. For example, based on their revealed preferences, a self-evolving DSS might display sales forecasts as bar charts for manager A, whereas it might display the same information as line charts for manager B.

Implementation of the Self-evolving Design

AN IDEAL SELF-EVOLVING DSS must provide self-evolving capabilities in the following three areas:

(1) changes in the problem/task domain,
(2) changes in technology, and
(3) changes in the user's behavior.

Based on current technology, it may not be easy to develop a system which can automatically adapt to a change in the problem domain or technology. For example, a DSS cannot adopt a more powerful graphics terminal unless the designer has installed the required tool to support the terminal. Nor can a DSS automatically solve a new type of problem without employing a new model developed either by the user or the designer. It is possible, however, to design a system able to adapt automatically to changes in the user's behavior without requiring the involvement of either the designer or the user. Therefore, the self-evolving approach proposed in this article is focused on building capabilities to meet changes in the user's behavior.

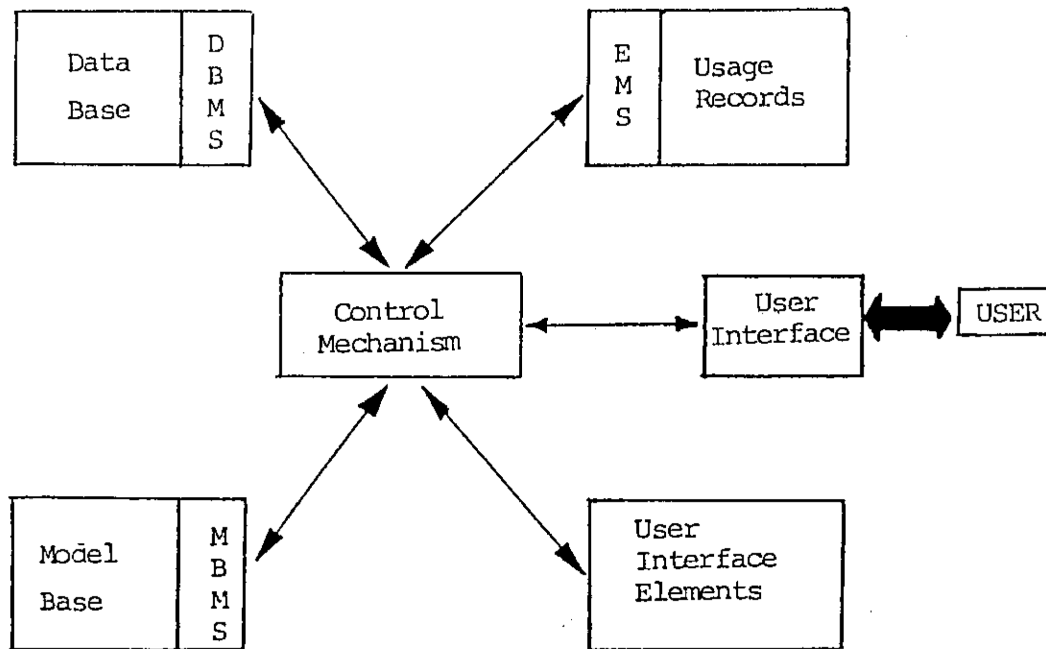
The key idea of the approach is that a system can control its evolution by adjusting its default policy. In most information systems, defaults have been widely employed for operating a system in the case where a user entry is lacking. The default policy is a policy that controls the set of allowed actions, results produced, and the way those results are presented by adjusting system defaults. In other words, the major parameter to be adjusted to control the evolution of the system is a set of system defaults. The system dynamically adjusts those defaults based on the anticipated requirements identified by a set of predetermined rules for system evolution. This is an application of the parameter adjustment approach widely used in machine learning, which changes the value of a certain parameter to control the behavior of the system [18]. By adjusting its default values, a system can learn how it has been used and then adapt its behavior to meet changes in its user requirements.

In the remainder of this section, first, the architecture of a self-evolving DSS will be portrayed. Then, design of a control mechanism which integrates various components in a DSS and an evolution management system that maintains system usage data will be discussed. Finally, possible applications of this approach will be presented.

Architecture of the System

A self-evolving DSS is composed of the following five major components:
(1) database subsystem,
(2) model base subsystem,
(3) user interface subsystem,
(4) evolution subsystem, and
(5) a central control mechanism for coordinating those subsystems.

The database subsystem consists of the database management system (DBMS) and the database which contains all data related to the decisions to be supported. The model base subsystem consists of models in the model base and the model base management system (MBMS). The user interface subsystem consists of elements for building a user interface and the user interface for DSS-user communication. The evolution subsystem consists of system usage data pertinent to the evolution of the



Notes: DBMS = database management system; EMS = evolution management system; MBMS = model base management system.

Figure 2. Architecture of a Self-evolving Decision Support System

system and the evolution management system (EMS) that manages the evolution of the system. Although the system usage data and the decision data may actually be stored in the same database, conceptually they are considered two separate databases because of their different purposes. Figure 2 illustrates the relationships among these components.

The core of a self-evolving DSS includes the control mechanism and the evolution subsystem. The control mechanism passes parameters between different subsystems and integrates them to support the user. The evolution management system handles system usage data and rules that determine appropriate default policy for a specific version of a DSS. The functions of the database subsystem, the model base subsystem, and the user interface subsystem for the self-evolving DSS are almost the same as those for a DSS developed by current design approaches.

Functions of the Control Mechanism

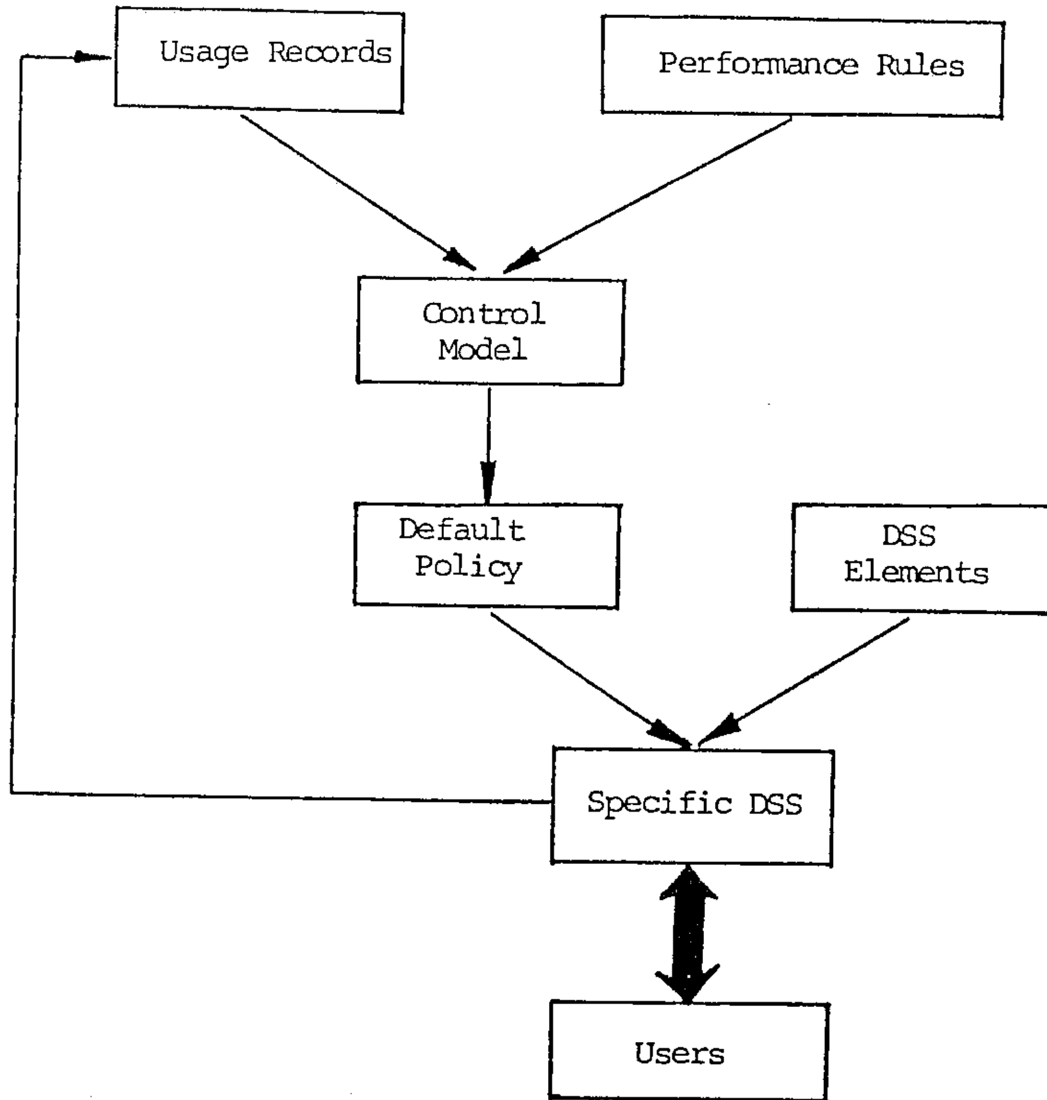
The control mechanism coordinates all operations of the self-evolving DSS. When a user accesses the system, the mechanism works as follows:

(1) evokes the evolution management system, which retrieves the usage record of the user from the database of usage records;

(2) obtains rules for determining appropriate defaults;

(3) assigns the most appropriate default policy for the system, based on the rules obtained at Step 2;

(4) collects new usage records and stores them in the database of usage records;



Note: DSS = decision support system.

Figure 3. Mechanism of Self-Evolution

(5) assigns the defaults determined by the evolution management system to the system, integrates the database subsystem, the model base subsystem, and the user interface subsystem, and then

(6) provides the integrated DSS to the user.

Figure 3 illustrates the process for self-evolution. First, the user accesses the specific DSS. The system then collects the user's usage data and stores them in the usage record base. The control model analyzes those data and adjusts the default policy based on the stored performance rules in order to develop a new version of the specific DSS. The next time the user accesses the system, the new version of the system will be provided. Of course, the new version may be the same as the old one.

Design of the Evolution Management System

An evolution management system (EMS) is the expert unit in a self-evolving DSS. It makes recommendation on system evolution to the control mechanism. Based on

analysis of a user's usage profile, a default policy and a set of defaults for formulating a specific version of a DSS for the user can be determined. In order to perform this function, the EMS must keep track of every user's usage profile. That is, a database of usage records must be maintained to store related records to identify the usage pattern of a particular user. In designing such a database, the EMS must consider issues such as what data should be kept and what patterns can be identified. In addition, given the usage profile the system also needs a set of rules and performance measures to analyze the usage data in order to determine appropriate behavior of the system. In this section, the following major issues in developing an EMS will be discussed:

1. determining appropriate performance measures,
2. finding available default policies,
3. identifying patterns of usage behavior,
4. designing a database of system usage records, and
5. developing rules controlling system evolution.

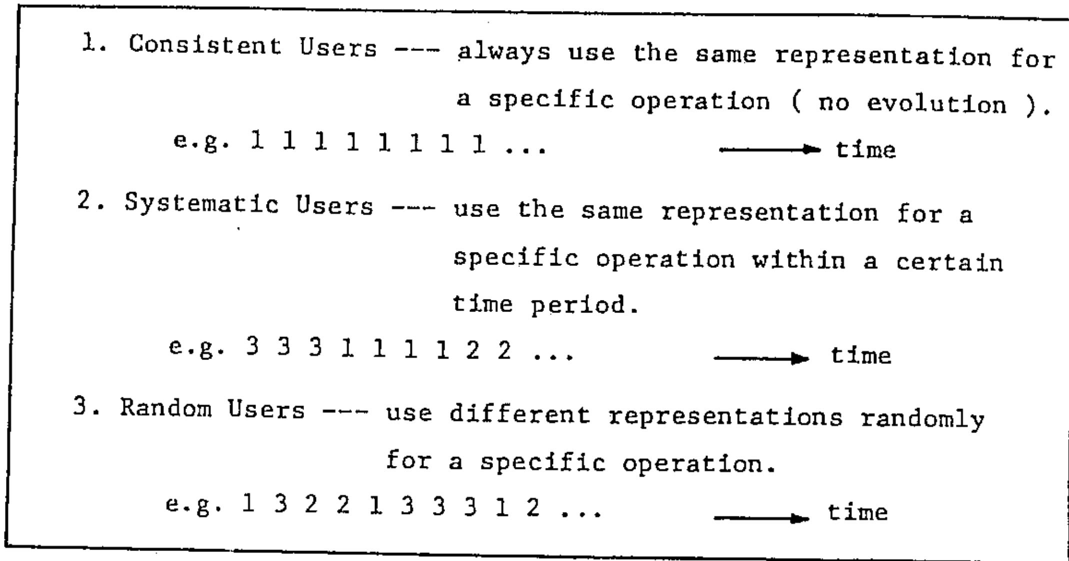
1. Performance Measures

Although a self-evolving DSS may lead to higher user satisfaction, the most significant advantage of the system would be to reduce the effort required to use the system. In order to accomplish this goal, performance measures are needed to direct the evolution of the system. They can be either subjective or objective or both. The selection of appropriate performance measures depends upon the objective of the system. For example, if we want to develop a system which minimizes the effort required to execute a job, the number of keystrokes for performing an operation may be an appropriate measure.

2. Default Policies

For developing a self-evolving DSS, there are three kinds of default policies: fixed default, dynamic default, and no default. A fixed default policy offers a fixed operation (e.g., sensitivity analysis) or representation (e.g. a bar chart) to the user unless the default is changed. In a self-evolving DSS, the system default can be changed by the designer, the user, or the system. A dynamic default policy allows the default to be changed automatically by the system if some specific conditions are fulfilled. A no default policy provides no default for operations or representations; that is, the user must request any desired action either through a menu or through the command language.

The advantage of a fixed default policy is that a user can correctly anticipate what is going to appear. However, the fixed default may not reflect the user's current preference or requirements. A dynamic default policy provides flexibility that the fixed default policy lacks but loses some consistency since the behavior of a system can change over time. A no default policy allows a user to select the preferred operation or representation but requires more action and effort to obtain the result. Sometimes a user may have to traverse four or five levels in a hierarchy of menus



* Note: 1 ... Bar chart
2 ... Line chart
3 ... Table

Figure 4. Types of Evolution

before discovering the desired operation. A DSS can adapt itself by changing its default policy and default value.

3. Patterns of Usage Behavior

The implementation of the self-evolving design approach also requires a DSS to have the capability to identify users' patterns of usage in order to determine the appropriate default policy and adjust the system default. Although previous research has tried to classify users by their cognitive styles or their roles in decision making, different studies usually draw conflicting conclusions [11, 15, 22]. Therefore, these studies provide very limited insight from a prescriptive point of view. For the purpose of this research, a better approach would be to examine the evolution pattern of system use. There are certainly other alternatives which may lead to different design. A complete discussion on this issue is, however, out of the scope of this research.

From the evolution perspective, patterns of usage behavior can be divided into three categories: consistent, systematic, and random [13]. A consistent user almost always prefers the same set of operations or representations in dealing with a specific problem. A systematic user requires different operations or representations at different times, but the change is partially predictable. A random user changes preferences randomly, and the change is entirely unpredictable. Figure 4 illustrates three sample patterns. The user who always requests a bar chart is a consistent user. The user who changes preference systematically from a table to a bar chart and then from a bar chart to a line chart is a systematic user. The user who, at any point in time, is equally likely to choose a table, bar chart, or line chart is a random user.

Whereas different users may demonstrate different types of evolution, an individual user may also change usage behavior over time [7, 13]. For instance, a random user may evolve into a systematic user and then, conditioned by the system, become a consistent user. The identification of a user's profile is based on data collected in a predetermined length of time. In terms of model management, a random user may choose different models to forecast future sales. After several times of use, however, the user may develop preference for a particular sales forecasting model and then evolve into a consistent user.

In order to develop a self-evolving DSS to meet the various types of users, the EMS needs two components: a database of usage records to track usage patterns and a knowledge/rule base to determine the appropriate default policy.

4. The Database of Usage Records

Related records of usage must be collected in a database in order to identify the pattern of usage of a particular user. Since the huge amount of data makes it prohibitive to collect all usage records, the system designer must first determine what kinds of usage records are needed to control the evolution of the system, that is, to determine the evolutionary variables. For example, if the representation format preferred by a user (e.g., bar chart or pie chart) is expected to change over time, then the designer should treat the representation format as an evolutionary variable in designing the DSS, and the database of usage records should be designed to keep those data.

Because the identification of evolutionary variables also determines what features of the system will be able to evolve, it has crucial effects on the validity of the system. Therefore, the designer must work very carefully with the help of the user at this stage.

After determining the evolutionary variables, the designer must design a database structure to maintain all usage data related to the identified evolutionary variables. All principles for developing a database management system are applicable to the design of this database; all functions for data management, including data storage, update, and retrieval, must also be implemented in this database.

5. A Knowledge/Rule Base for Determining Actions

The knowledge/rule base contains knowledge about when and how a system should evolve. A rule is usually used to connect a given situation with appropriate actions under that situation. It provides a natural way for describing processes driven by a complex and rapidly changing environment. An example would be, "If humidity is greater then 99% then it is going to rain."

A set of rules specifies how the system should react to the changing data without requiring detailed knowledge about the flow of control. These rules can be divided into two categories: domain rules and meta-rules. Domain rules represent knowledge pertinent to a particular application, whereas meta-rules describe how domain rules should be used or modified. For example, "If the expected number of key-

strokes of a default policy is less than 1/2 then the default policy should be adopted'' is a domain rule; but ''If more than one rule applies to a particular situation then the system should use them in descending order'' is a meta-rule. In developing a self-evolving mechanism, the system designer must develop domain rules in the rule base since many artificial intelligence or expert systems tools provide meta-rules for tracing the execution of the knowledge base. For example, PROLOG provides back-tracking capabilities. This can save much time and effort. For more discussion, please see [23].

Assuming the usage records have been collected, four kinds of domain rules are needed to analyze those records and determine the appropriate default policy:

(1) rules for identifying patterns of usage,

(2) rules for measuring performance,

(3) rules for determining appropriate default policy, and

(4) rules for assigning appropriate time for evolution.

1. Rules for Identifying Relevant Usage Records in Order to Determine Patterns of Usage. This kind of rule is designed to identify the usage profile of a particular user. It includes rules for identifying patterns of usage and rules determining how to use those rules for identifying patterns of usage. An example would be, ''The analysis of the usage pattern should be based on the ten most recent uses of the system.''

2. Rules for Computing the Performance of Various Default Policies. This set of rules determines how performance must be measured. A typical rule here would be, ''The performance of a default policy is measured by the expected number of keystrokes required to perform an operation.''

3. Rules for Determining Appropriate Actions. This set of rules specifies conditions under which a default policy is applicable—for example, ''If the expected number of keystrokes of a default policy is less than 0.5 then take the default policy as a candidate.''

4. Rules for Determining an Appropriate Time for Evolution. If a system evolves too frequently, the user may be confused by the changes and hence not trust it. Therefore, rules are needed to determine when the evolution should take place. The determination of this kind of rule must take into account not only system performance but also system consistency. An example would be, ''The system evolves every time the system is initiated, but collects records of usage throughout the time the DSS is used.''

An example that illustrates different rules and the process of the self-evolving design will be discussed in the next section.

Applications of the Self-evolving Design

The self-evolving design can be applied to many areas. In this section, the authors describe three of these promising applications for decision support: intelligent user interface design, model management system design, and DSS generator design. In

fact, this approach can also be used to develop an adaptive expert system which interacts with different users in different ways.

1. Intelligent User Interface Design

The user interface of a system is the channel through which a user communicates with the system. Most current information systems either have a fixed user interface, which interacts with users by a predetermined format and sequence, or provide menus to users and allow users to select the preferred format and presentation sequence [22]. They have little capability of identifying different users, not to mention learning the changes of a user's preference.

In the real world, if a Chief Executive Officer (CEO) needs a sales report comparing sales for the past ten years, he does not have to tell his secretary whether he prefers a table, a pie chart, or a bar chart. However, if he works with a traditional DSS, he will either be given the same report format every time he uses the system or will have to specify many parameters required for generating a new format. Neither way can be considered satisfactory.

By applying the previously described self-evolving mechanism to the user interface design, the system will be able to identify a particular user's preference on representation formats and perform intelligent interaction by analyzing the user's previous usage records. Examples of considering user's behavior for the user interface design can be found in [5, 13], although the approach described in [5] still requires human involvement in order to analyze usage records.

2. Model Management System Design

Model management systems are focused on facilitating the management of decision models. Since different users may prefer different models for solving a problem and different models may have different validities in different situations, model selection is one of the most important issues in model management [14]. Applying the self-evolving approach to model management system design can incorporate a user's revealed preference into the model selection process and increase the usefulness of the system.

For example, manager A prefers using the regression model to forecast future sales, whereas manager B trusts the exponential smoothing model. In a self-evolving model management system, the system will consider this fact and provide appropriate sales forecasting models to the managers in case they want to forecast future sales or want to work with the system to develop a new model involving the sales forecasting model.

3. DSS Generator Design

Facilitating the development of DSS generators is one of the primary goals of the self-evolving design approach. A DSS generator provides an environment for developing specific DSS which support various decisions. Through integrating the self-evolving

capabilities, a DSS generator will be able to provide different versions of a specific DSS to different users. This can significantly reduce the cost for developing many separate specific DSS and encourage different users to share a system.

Although the self-evolving mechanism provides a general approach to system evolution for DSS generators, the user's usage data are associated with a specific DSS. In other words, those data are problem-dependent. We cannot use the data collected from a sales forecasting DSS to determine the appropriate representation format for a cash flow analysis DSS. Each specific DSS should have its own database of usage records. The DSS generator, however, provides functions for analyzing the data and tracing the evolution of the system.

An Illustrative Example

IN ORDER TO ILLUSTRATE the process of self-evolving design, the development of a DSS that provides monthly sales forecasts to the decision maker is presented in this section. The first step for designing this system is to determine appropriate performance measures and evolutionary variables, that is, characteristics that will change over time. After identifying the evolutionary variables, the designer must develop rules to be used to analyze the usage records and to set the appropriate default policy. Finally, the designer implements the system.

Performance Measures

Suppose the user wants a system that minimizes the actions required to produce the desired information; that is, the objective of the design is to reduce the effort in using the system. A proper performance measure might be the expected number of actions (selections or keystrokes) required to produce the desired information. Assume further that two variables are evolutionary: (1) whether or not the user wants to perform sensitivity analysis, and (2) the representation format preferred by the user.

After identifying the performance measure of the system, the designer decides that the performances of various default policies will be measured based on the five most recent uses of the system.

Policies and Rules

The next step is to determine proper default policies and rules. The following three default policies will be adopted as the domain of default policies in this example.

Policy 1: Fixed menus will be provided for requesting sensitivity analysis and representation format (no default policy).

Policy 2: Sensitivity analysis will be performed automatically and tables will

```
Rule 1: IF (For all policy i: Performance(i) >= 1/2) THEN
        (Put policy 1 into Candidate_default_policy)

Rule 2: IF (There exists a policy i: Performance(i) < 1/2) THEN
        (Put policy i into Candidate_default_policy)

Rule 3: IF size(Candidate_default_policy) > 1 THEN
        Default policy = policy with minimum performance

Rule 4: IF more than one policy fulfills Rule 3 THEN
        default policy = policy with minimum ID number
```

```
Where:

   - i: Identification (ID) number of a default policy.

   - Performance(i): The expected number of keystrokes if
        policy i is adopted as the default policy.

   - Candidate_default_policy: A set which contains all
        default policies that fulfill Rule 1 and Rule 2.

   - Size(Candidate_default_policy): The number of elememts
        in the Candidate_default_policy.
```

Figure 5. Rules Used in the Illustrative Example

present the output (fixed default policy).

Policy 3: The decisions as to whether the sensitivity analysis will be performed and whether the representation format will be changed are based on previous usage records (dynamic default policy).

Following the specification of the default policies, the designer determines that four rules are appropriate for adjusting the default policy, as in Figure 5.

Rule 1 means if, for all default policies in the domain, the expected number of keystrokes required to produce the information is greater than or equal to 1/2, then the default policy of the system will be policy 1, menu selection. In this example the hurdle number in the rule (1/2) is arbitrary. It means that the user may have to select the operation or representation every other time he accesses the system. In other words, the default policy with performance of 1/2 is expected to have a probability of 0.5 in correctly predicting the user's preference. Since the usage data are problem-dependent, different performance measures and hurdle numbers may be used in different systems.

Rule 2 means if there is a default policy, i, with an expected number of required actions less than 1/2, then it will be considered as a candidate for the new system default. Rule 3 means that the candidate default policy with the best performance (i.e., the policy with minimum expected keystrokes) will become the new system default if more than one policy is qualified by rule 2. Rule 4 means if more than one

	Manager A	Manager B
Sensitivity Analysis	Y, Y, Y, Y, Y	Y, N, N, Y, N
Representation Format	B, B, T, T, T	P, -, -, B, -

Note: Y = Yes, N = No,
T = Table, P = Pie chart, B = Bar chart.
- = Select none

Figure 6. Sample Usage Records

policy fulfills rule 3 then the new system default will be the policy with the smallest policy identification number. In this example, we set menu selection (no default policy) to be policy 1, fixed default policy to be policy 2, and dynamic default policy to be policy 3. This means that given the same performance we prefer menu selection to a fixed default policy and prefer a fixed default policy to a dynamic default policy. In the Appendix, a PROLOG implementation of this self-evolving mechanism is illustrated. It shows how these rules determine the appropriate default policy and default value.

System Behavior

If the usage data of managers A and B for each of their previous five usages of the DSS are as shown in Figure 6 then the performances of various policies after analyzing the usage records are as shown in Figure 7. Each value in Figure 7 is the expected number of selections the user has to make given the policy and the user's pattern of usage. For example, the performance of the fixed default policy for sensitivity analysis is equal to zero for manager A and 2/5 for manager B. In other words, manager A should require no action since the system will perform sensitivity analysis automatically every time he accesses the system, but manager B is expected to take action to bypass the execution of sensitivity analysis two times out of five if the fixed default policy is adopted.

According to the previously specified performance rules, the DSS would behave as follows: the system adopts the fixed default policy for both the sensitivity analysis (default = yes) and the representation format (default = table) when manager A is on-line. That is, the DSS will perform sensitivity analysis and report the results automatically in tabular form. If manager B uses the system, however, because the fixed default policy (default = no) is considered appropriate in performing sensitivity analysis, the system will not perform sensitivity analysis unless manager B asks it to do so. If manager B requests sensitivity analysis, the system will provide a menu allowing manager B to indicate the preferred representation format, because the no default policy is considered appropriate here.

	Manager A	Manager B
Sensitivity Analysis		
Fixed Default	0	2/5
Dynamic Default	1/5	4/5
No Default	1	1
Representation Format		
Fixed Default	2/5	1/2
Dynamic Default	2/5	1
No Default	1	1

Figure 7. Performance of Various Policies

If manager B's behavior evolves to the same pattern as that of manager A's, then the DSS provided to manager B will automatically change the default policy and evolve to a system identical to the one currently provided to manager A. Although the example is highly simplified, it does demonstrate the mechanism for self-evolution and the behavior of a self-evolving system. A more sophisticated system would require more evolutionary variables and more complex performance rules.

Conclusion

THE PAPER HAS PRESENTED a self-evolving approach for developing decision support systems. This approach would build learning capabilities into a DSS to offset some drawbacks of current design approaches such as the scarcity of user's and designer's time and the lack of adaptability to users' behavior. Figure 8 summarizes the differences among the SDLC approach, the user-involved evolutionary approach, and the self-evolving design we have described [see also references 4, 5, 6, 8, 9, 11, 17, 19, 20]. The evaluations illustrated in Figure 8 reflect relative rather than absolute measures. For example, although the user involvement is important to all three approaches, the degree of user involvement is relatively low in the SDLC approach compared with the degree in the evolutionary design and the self-evolving design.

From the cost-benefit point of view, the initial development cost for adopting this approach may be higher than that of the user-involved evolutionary design because a self-evolving mechanism must be built. After the system has been developed, how-

FACTORS	SDLC	EVOLUTIONARY	SELF-EVOLVING
1. Initial development cost	High	Low	Medium
2. Evolutionary cost	High	Medium	Low
3. Interference with the decision process	Low	High	Medium
4. System flexibility	Low	Medium	High
5. User's control over the system	Low	High	Medium
6. User involvement required	Low	High	Medium
7. Encouraging system sharing	Low	Low	High

Note: SDLC = system development life cycle.

Figure 8. A Comparison of Various Approaches

ever, the adaptive cost will be lower, the flexibility of the DSS will be higher, and the interference in the user's decision process will be less. In addition, since the self-evolving approach tailors the system to fit different users' preferences, it would encourage system sharing and hence reduce the possibility of information inconsistency that may have been generated by many user-developed systems. Although empirical study for comparing different approaches is needed to support these arguments, successful implementation of this approach could shed much light on the development of a useful DSS.

The self-evolving approach presented in this paper has several limitations, which also indicate promising areas for future research. First, users may have a strong desire to control a system and cognitively do not like a system with self-evolving capabilities. Second, the validity of the default adjustment model is crucial to the success of a self-evolving DSS. If the system evolves in an unexpected, undesigned way, user satisfaction will diminish. Therefore, lack of knowledge of human decision processes may significantly reduce the usefulness of this approach. The progress in artificial intelligence and cognitive sciences will be able to offset this limitation and hence increase the value of this approach. Third, the implementation of the approach requires highly intelligent designers (probably with the help of the user) to design the self-evolving mechanism in order to control the evolution of the system. Even with the limitations mentioned, the self-evolving approach still has good potential in many application areas, including the development of DSS generators and model management systems.

Appendix: A PROLOG Implementation

1. Description

The program is composed of the following four functions:

(1) retrieval of usage data,
(2) analysis of usage data,
(3) determination of default, and
(4) report of the default.

Assuming that the usage data represented by the predicate ''data'' are already available in the database, the first step of the mechanism is to retrieve the usage data stored in the database. Then, the predicate ''analyze'' analyzes the usage data of the user. A predicate is similar to a subroutine in a high-level language. The predicate ''analyze'' computes the performance of the fixed default policy by activating the predicate ''fixed_default'' and analyzes the performance of the dynamic default policy by activating the predicate ''dynamic_default.'' Finally, the predicate ''selection''is activated to select the most appropriate default policy, and the predicate ''report'' reports the result. Other predicates in the program are subroutines of the predicates described above.

To use the system, the user activates the predicate ''default.'' If the user's usage data are available in the database, the system will analyze the data and report a proper default. For example, if the user enters ''default(manager_a)'', then the system will analyze the usage data of manager A in the database and generate the most appropriate default and its expected performance, which is the fixed default policy with the expected number of keystrokes equal to zero.

2. A PROLOG Program Listing

```
/* Sample database */

  data(manager_a,[y,y,y,y,y]).
  data(manager_b,[y,n,n,y,n]).

/* Main Program */

  default(User):-
     data(User,Data),
     analyze(Data,Default_candidates),
     select(Default_candidates,Best),
     report(User,Best).
  default(User):-
     nl,write('Sorry—usage data not available !!'),
     nl,nl,!.
```

```
analyze(Data,Result):-
    fixed__default(Data,Candidacy__1),
    dynamic__default(Data,Candidacy__2),
    append([Candidacy__1],[Candidacy__2],Result1),
    delete([],Result1,Result).

select(Default__candidates,Best):-
    list__length(Default__candidates,L),
    L=1,
    first(Best,Default__candidates).
select(Default__candidates,Best):-
    list__length(Default__candidates,L),
    L=2,
    first(Fixed,Default__candidates),
    delete__head(Fixed,Default__candidates,Default__candidates__2),
    first(Dynamic,Default__candidates__2),
    compare(Fixed,Dynamic,Best).
select([],[menu,no,1.0]).

compare(Fixed,Dynamic,Best):-
    third(P2,Dynamic),
    third(P1,Fixed),
    P1>P2,
    append(Dynamic,[],Best).
compare(Fixed,Dynamic,Best):-
    append(Fixed,[],Best).

report(User,Best):-
    write('The optimum default policy, default value, and '),
    nl,write('expected keystrokes for '),write(User),
    write(' = '),nl,nl,
    write(' '),
    write(Best),nl,nl.

fixed__default(Data,Candidacy__1):-
    list__length(Data,L),L>0,
    count(Data,L,Performance),
    candidate__1(Performance,Candidacy__1).
fixed__default(Data,[]).

count([],__,[]).
count(Data,L,Performance):-
    first(F1,Data),list__length(Data,L1),
    delete(F1,Data,Data__1),
```

```
    list_length(Data_1,L2),
    P1 is 1 - (L1 - L2) / L,
    P1 > 0.5,!,
    count(Data_1,L,Performance).

count(Data,L,Performance):-
    first(Default_value,Data),list_length(Data,L1),
    delete(Default_value,Data,Data_1),
    list_length(Data_1,L2),
    Perf is 1 - (L1 - L2) / L,
    append([Default_value],[Perf],Performance).

candidate_1(Performance,Candidacy_1):-
    list_length(Performance,L),!,
    L>0,
    append([fixed],Performance,Candidacy_1).
candidate_1([],[]).

dynamic_default(Data,Candidacy_2):-
    compute(Data,Performance),
    candidate_2(Performance,Candidacy_2).
dynamic_default(Data,[]).

compute(Data,[Default_value,Performance]):-
    list_length(Data,L),L>0,
    keystroke(Data,Key),
    P is Key/L,
    P<0.5,
    first(Default_value,Data),
    Performance is P.
compute(_,[]).

keystroke(Data,Key):-
    first(F1,Data),
    delete_head(F1,Data,Data_1),
    match_head(F1,Data_1),
    keystroke(Data_1,Key).
keystroke(Data,Key):-
    first(F1,Data),
    delete_head(F1,Data,Data_1),
    not match(F1,Data_1),
    keystroke(Data_1,Key1),
    Key is Key1 + 1.
keystroke([],0):-!.
```

```
  candidate_2(Performance,Candidacy_2):-
     list_length(Performance,L),!,
     L>0,
     append([dynamic],Performance,Candidacy_2).
  candidate_2([],[]).

/* Utilities */

  first(X,[X|Anything]).

  third(X,[A,B,X]).

  append([],List_1,List_1).
  append([A_head|Tail_1],List_2,[A_head|Tail_1_then_list_2]):-
     append(Tail_1,List_2,Tail_1_then_list_2).

  delete_head(Element,[Element|Tail],Tail):-!.

  delete(_,[],[]).
  delete(Element,[Element|Tail],List1):-!,
     delete(Element,Tail,List1).
  delete(Element,[Any|Tail1],[Any|Tail2]):-
     delete(Element,Tail1,Tail2).

  list_length([],0).
  list_length([X|Tail],Length):-list_length(Tail,K),
     Length is K+1.

  match_head(Element,[Element|Anything]).
```

REFERENCES

1. Alavi, M., and Henderson, J. C. An evolutionary strategy for implementing a decision support system. *Management Science*, 27, 11 (1981), 1309–1323.

2. Alavi, M., and Weiss, I. R. Managing the risks associated with end-user computing. *Journal of Management Information Systems*, 2, 3 (Winter 1985–86), 5–20.

3. Alter, S. Transforming DSS jargon into principles for DSS success. In Young, Donovan, and Keen, Peter G. W., eds. *DSS-81 Transactions*, Execucom System Corp., 1981, 8–27.

4. Davis, G. B. Caution, user-developed DSS can be dangerous to your organization. *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, Honolulu, 1982, 750–763.

5. Good, M. D., et al. Building a user-derived interface. *Communications of the ACM*, 27, 10 (1984), 1032–1043.

6. Hiltz, S. R., and Turoff, M. Office augmentation systems: The case for evolutionary

design. *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, Honolulu, 1982, 737–749.

7. Hurst, E. G., Jr. The role of humans in decision support systems. Decision Sciences working paper 78-02-01, The Wharton School, University of Pennsylvania, 1978.

8. Hurst, E. G., Jr., et al. Growing DSS: A flexible, evolutionary approach. In Bennett, J. L., ed. *Building Decision Support Systems*. Reading, Mass.: Addison-Wesley, 1983, 111–132.

9. Keen, P. G. W. Adaptive design for decision support systems. *Database*, 12, 1-2 (1980), 15–25.

10. Keen, P. G. W., and Gambino, T. J. Building a DSS: The mythical man-month revisited. In Bennett, J. L., ed. *Building Decision Support Systems*. Reading, Mass.: Addison-Wesley, 1983, 133–172.

11. Keen, P. G. W., and Scott Morton, M. S. *Decision Support Systems: An Organizational Perspective*. Reading, Mass.: Addison-Wesley, 1978.

12. Konsynski, B. R. Advances in information systems development. *Journal of Management Information Systems*, 1, 3 (Winter 1984–85), 5–32.

13. Liang, T. P. A self-evolving user interface design for decision support systems. *Proceedings of the Seventeenth Hawaii International Conference on System Sciences*, Honolulu, 1984, 548–557.

14. Liang, T. P. A graph-based approach to model management. *Proceedings of the Seventh International Conference on Information Systems*, San Diego, 1986.

15. Mason, R. O., and Mitroff, I. I. A program for research on management information systems. *Management Science*, 19, 5 (1973), 475–487.

16. McLean, E. R. End-users as application developers. *MIS Quarterly*, 3, 4 (1979), 37–46.

17. Moore, J. H., and Chang, M. G. Meta-design considerations in building decision support systems. In Bennett, J. L., ed. *Building Decision Support Systems*. Addison-Wesley, Reading, Mass.: 1983, 173–204.

18. Rich, E. *Artificial Intelligence*. Englewood Cliffs, N. J.: Prentice-Hall, 1983.

19. Rockart, J., and Lauren, F. The management of end user computing. *Communications of the ACM*, 26, 10 (1983), 776–784.

20. Snyder, C. A., and Cox, J. F. A dynamic systems development life cycle approach: A project management information system. *Journal of Management Information Systems*, 2, 1 (Summer 1985), 61–76.

21. Sprague, R. H., Jr. A framework for the development of decision support systems. *MIS Quarterly*, 4, 4 (1980), 1–26.

22. Sprague, R. H., Jr., and Carlson, E. D. *Building Effective Decision Support Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.

23. Waterman, D. A. *A Guide to Expert Systems*. Reading, Mass.: Addison-Wesley, 1986.

24. Watson, H. J., et al. An investigation of DSS developmental methodology. *Proceedings of the Seventeenth Hawaii International Conference on System Sciences*, Honolulu, 1984, 515–519.