The goal of this paper is:

<table>
<thead>
<tr>
<th>goal</th>
</tr>
</thead>
</table>
describe information system that is ...  
EASY TO DEVELOP  
EASY TO MODIFY

The paper will be divided into two parts:

<table>
<thead>
<tr>
<th>overview</th>
</tr>
</thead>
</table>
PART 1:  
develop simple example  
PART 2:  
apply structured techniques

**PART 1: SIMPLE EXAMPLE**

Assume a scientist is collecting blood pressure data on rats:

<table>
<thead>
<tr>
<th>PART 1: example data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAS® VARIABLE NAME</strong></td>
</tr>
<tr>
<td>drug</td>
</tr>
<tr>
<td>rat</td>
</tr>
<tr>
<td>bp</td>
</tr>
</tbody>
</table>

The most primitive information system would input the data via the 'cards' statement into a temporary SAS® data set and then print the data in a report:

```
data labdata;  
input drug rat bp cards;  
proc print;```

(Note: Example SAS® code is for illustration only. The code is abbreviated and not meant to be complete. 'Run' statements and other ancillary code is intentionally not included.)

This is how many of us developed SAS® systems when the SAS® System ran only in batch mode. SAS® data sets were often temporary and data editing was accomplished in the data step or by changing the input data on the cards. This mode of operation didn't allow for convenient data handling which is important for any information system.

With the introduction of the interactive line mode it became possible for direct interactive editing of a SAS® data set with 'proc editor'. Data was typically input via an 'infile' statement into a permanent SAS® data set:

```
data my.labdata;  
infile myfile;  
input drug rat bp;  
proc editor;  
proc print;```

With the introduction of SAS/FSP®, full screen interactive input and editing of the data became possible with 'proc fsedit'. The system developer needed only to create an empty data set in the intial data step:

```
data my.labdata;  
length drug rat bp 8;  
stop;  
proc fsedit;  
proc print;  
proc print;```
In addition to interactive user control of the data we would like to have interactive user control of program direction, i.e. allow the user to select the order of procedure and data step execution rather than have a fixed linear sequence. The SAS® macro language offered the first opportunity to control program direction. Screen input is converted to a macro variable which then controls branching based on the value of the macro variable:

The macro language in combination with SAS/FSP® allowed the development of very sophisticated information systems. Unfortunately these systems could also be very difficult to develop and modify. Many of us can testify to debugging nightmares with the macro language. SAS/AF® facilitated the same branching ability but with a much easier menu generator:

Branching could now be achieved without programming: instead we tell SAS/AF® where to branch to. A SAS/AF® program is a collection of screens, which for this example include menu and program screens:

The menu and program screens are stored in a SAS® catalog with an associated screen directory:

If we put an 's' next to the main menu entry in the directory we select the menu screen:

The menu screen is created by the system developer for the user to see. Associated with the menu screen is a menu attribute screen. On the menu attribute screen we tell SAS/AF® which screen to go to when the user selects a menu option. The screens listed are referred to as the 'child' screens for this menu:

If we return to the catalog directory and select the data program screen we will get a screen of SAS® code that begins with dashed lines:
The dashed line divides the program screen into two parts: the area above the dashed line is optionally visible to the user and the area below the dashed line has the SAS® code we want submitted. The 'proc display' at the bottom of the program screen tells SAS/AF® to return to the menu. The fsedit program screen looks like this:

```
fsedit.program screen
proc fsedit data .. my.labdata;
proc display;
```

The print program screen looks like this:

```
print.program screen
proc print data=my.labdata;
proc display;
```

The scientist may want to print a report for only one drug at a time. The print program screen would be modified to allow the scientist to enter the drug number in the area above the dashed line. This drug number could then be used in an 'if statement in a data step to create a temporary data set to include data for the selected drug only:

```
print.program screen to print 1 drug
enter drug number: &drugnum --------------
(USER SEES 10 UNDERSORES STARTING AT &)
data tempdata;
set my.labdata;
if drug = &drugnum;
proc print data = tempdata;
proc display;
```

Suppose the scientist wants to print a report for only one drug at a time. The print program screen would be modified to allow the scientist to enter the drug number in the area above the dashed line. This drug number could then be used in an 'if statement in a data step to create a temporary data set to include data for the selected drug only:

```
print.program screen to print 1 drug
enter drug number: &drugnum --------------
(USER SEES 10 UNDERSORES STARTING AT &)
data tempdata;
set my.labdata;
if drug = &drugnum;
proc print data = tempdata;
proc display;
```

The '8' above the dashed line tells SAS/AF® that it is at the beginning of a user field. The '&drugnum' is not a macro variable and the user supplied value is available for the SAS® code below the dashed line on this screen only.

If we wanted to run several report programs on the same drug it would be inefficient to create a temporary data set each time. A better strategy would be to create the data set once and make it available to a separate report menu:

```
add options for 1 drug to menu
select option:
<< all data >>
a. create database
b. input data
<< for 1 drug >>
c. print data
d. plot data (proposed program)
e. ...
```

The main menu would look like this:

```
main menu
select option:
a. go to report menu
b. create database
c. input data
d. ...
```

Option 'a' will create the temporary data set and transfer control to the report menu. Suppose we call its program screen 'goto.program':

663
Notice that a temporary data set is created that includes data for one drug only. This data set will be available for all programs on the report menu:

To return to the main menu we press the 'end key'. The report menu attribute screen needs to include instructions on how to return to the main menu. The 'parent screen' is the screen that is invoked if the user presses the 'end key':

If we look at the catalog directory for our two menu system we see that it is getting more complicated:

If we came back to this system a year later it may not be obvious how the screens relate to each other without looking at the menu attribute screens to understand the branching. Imagine if this system had several menus and hundreds of screens. As an example of a complex system consider this diagram where p = program screen and m = menu screen:

![Diagram of a complex system with a menu screen and several program screens branching off it.]

Nothing prevents menu m3 from branching to program p3 thus complicating the structure even more. If we don't control the branching it is possible to develop systems that are very difficult to modify (or explain to others who may have to modify our system). At this point it would be helpful to introduce structured techniques to help us design our information systems:

**PART 2: STRUCTURED TECHNIQUES**

(Note: these structural techniques are not new and can be found in many system development texts. Furthermore the specific applications of these techniques described here are just an example of many possible implementations using SAS®. Your application may require another approach.)

We will begin by dividing our system into logical units which we will call modules. We will define a module to be a menu and all of its parent and child screens. We will further assume that all parent and child screens are program screens only. The program screens are associated with the 'end key' (parent screen) and menu options 'a', 'b', ..., 'z' (child screens):
An additional restriction is that the 'end key' program screen always returns control to the menu of the last module, i.e. the module that invoked the present module. Furthermore, the program screen associated with option 'a' is the only branching point to other modules. Program screens associated with options 'b' thru 'z' do the actual functional procedures for that module. A diagram for a two module system would look like this (M = menu):

For a three module system:

A more compact diagram that implicitly assumes the menu and attached program screens would look like this:

Now that we have defined a module we need to impose restrictions on the interrelationship of these modules. Some restrictions are implicit in the constraints we have already placed on the program screens for the 'end key' program and the option 'a' program. Modules are placed in a tree-like hierarchical structure with a single 'root' module. A module can be invoked only by a single 'father' module. A module always returns via the 'end key' to its invoking 'father' module. Thus a unique path is guaranteed between the root module and any other module. The entire structure will be referred to as a 'modular hierarchical model'. Complicated systems can now be developed where it is easy to relate the branching to the underlying menu and program screens. A hypothetical six module system is
Now that we have a model for defining and relating our menu and program screens we need a basis for actually designing our information system. We will use a 'data model' as the basis for system design. This is based on the principle that the 'data model' is relatively permanent whereas we may frequently wish to change the procedures that act on that data:

- use data model as basis of design
  - user views of the data (permanent data sets)
  - proc views of the data (temporary data sets)
  - independent canonical data model

Usually the user will envision the data in a certain way. This is typically the way the data would be described on a form designed by the user. SAS® procedures also expect to see or view the data in a certain way. Hopefully underlying these multiple user and proc views is an independent or canonical view of the data:

---

**example data**

<table>
<thead>
<tr>
<th>VARIABLES CANONICAL VIEW</th>
<th>VARIABLES ORIGINAL VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>drug</td>
<td>drug</td>
</tr>
<tr>
<td>rat</td>
<td>rat</td>
</tr>
<tr>
<td>parameter (bp, hr, etc.)</td>
<td>bp</td>
</tr>
<tr>
<td>response (value observed)</td>
<td></td>
</tr>
</tbody>
</table>

Our example data was originally described in a way that would be convenient for SAS® procs. The system developer would recognize that blood pressure was one of several parameters that the scientist may eventually want reports on. A canonical view of the data would accommodate this and define a new attribute (SAS® variable) called 'parameter'. On the other hand the user may envision the data like this:

---

The system developer could then provide an 'fsedit' screen designed to collect and store the data according to the user view of the data. The option 'a' branching program screen, 'goto, program', would then convert the data to canonical form before making it available to the report menu:

---

Suppose we have one set of reports that expect the data according to 'proc view 1' and another set of reports that expect the data according to 'proc view 2'. We could then design a separate module for each 'proc view'. The 'user view' would still be converted to the 'canonical view' which would in turn be converted to the appropriate 'proc view' for each module:
We need to add one additional structural technique in order to quickly develop and modify our information systems:

**use standard naming scheme for screens**

- to easily add and remove program and menu screens
- to identify where screen is in modular hierarchical model
- to allow development of shell or prototype of system

The first step in establishing a naming scheme is to devise a numbering algorithm for the modules:

![Diagram of module hierarchy]

The algorithm implied by the above diagram allows one to identify the father module of any given module. Furthermore the depth of a module in the hierarchy is obvious from the number. For multiple branches: 1 would refer to the first branch listed on the option ‘a’ program screen, 2 would refer to the second branch listed, etc.

Menu names are defined by prefixing an 'm' to the module number. The function of the menu can be supplied in the 'description' field allowed in the catalog directory:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>menu</td>
<td>main menu</td>
</tr>
<tr>
<td>m11</td>
<td>menu</td>
<td>report menu</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program screens follow the naming convention described on this example attribute screen for menu m11:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>p11a</td>
<td>program</td>
<td>option 'a'</td>
</tr>
<tr>
<td>p11b</td>
<td>program</td>
<td>option 'b'</td>
</tr>
<tr>
<td>p11c</td>
<td>program</td>
<td>option 'c'</td>
</tr>
<tr>
<td>p11z</td>
<td>program</td>
<td>option 'z'</td>
</tr>
</tbody>
</table>

name = (p)(module number)(option letter or _)

This naming scheme allows one to identify the program screen associated with each menu option. The 'end key' parent program screen p11_, transfers control back to menu m1. Using a program screen to transfer control (rather than just returning directly by assigning menu.m1 as the parent screen) allows certain programming details to be handled before returning to the invoking module:

```sas
proc datasets library=work;
delete tempdata;
title1;
options ...
proc display catalog=my.catalog.m1.menu;
```

The directory for a two menu prototype system would now look like this:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>_m1</td>
<td>menu</td>
<td>main menu</td>
</tr>
<tr>
<td>_m11</td>
<td>menu</td>
<td>report menu</td>
</tr>
<tr>
<td>_p1</td>
<td>program</td>
<td>option 'p1'</td>
</tr>
<tr>
<td>_p1a</td>
<td>program</td>
<td>option 'a'</td>
</tr>
<tr>
<td>_p1b</td>
<td>program</td>
<td>option 'b'</td>
</tr>
<tr>
<td>_p1c</td>
<td>program</td>
<td>option 'c'</td>
</tr>
<tr>
<td>_p1z</td>
<td>program</td>
<td>option 'z'</td>
</tr>
<tr>
<td>_start</td>
<td>program</td>
<td>set options, etc.</td>
</tr>
</tbody>
</table>

Program screen p11c is a proposed plot program. A prototype program screen can be set up before the actual SAS® code is written.
The system always enters the 'root' module via a 'start.program' screen which is used to initialize system options and macros:

```
start.program
options ...;
goptions ...;
(define macros)
proc display catalog=my.catalog.m1.menu;
```

The 'end key' program screen for menu m1 is program screen p1_.program which exits the SAS® System:

```
p1_.program
endsas;
```

In conclusion we have seen how certain structural techniques when used with SAS/AF® will allow us to easily develop and modify very sophisticated integrated information systems:

```
• use SAS/AF® for branching
• divide information system into modules
• put modules into hierarchical relationship
• design hierarchy with data modeling
• standardize naming scheme to prototype
```

SAS, SAS/AF and SAS/FSP

are registered trademarks of SAS Institute Inc., Cary, NC, USA