ABSTRACT

The Management Information Systems (MIS) department at SAS Institute Inc. uses SAS® software to develop all in-house applications. In addition, we also use SAS software to develop portable generic tools to avoid replication and promote code reuse. This paper discusses a variety of tools developed in MIS as well as describing their purpose and coding techniques.

Some of the tools discussed are:

1) mail interface
2) printing interface
3) small host specific layer of tool implementation
4) data entry validation
5) dynamic free form text entry/unlimited comments
6) metabase for application resource management
7) automatic search path setup
8) catalog entry check in/check out
9) application mode control at run time (production/development)

INTRODUCTION

As the technology available to a SAS applications developer increases, it is easy to lose track of the seemingly endless ways to program various tasks. In addition, our client/server world makes knowledge of other operating systems imperative. It seems there must be a way to bring some of the information overload under control, especially since these newest advances are supposed to make life easier, not more complicated.

MIS has had these problems, too, just like everyone else. Common tasks such as e-mail, printing, and library allocation are all done differently on different operating systems. Application development and maintenance has its own unique problems with multiple developers and testing changes.

In response to these situations, MIS has developed several useful tools that are portable and flexible enough to be used by many different applications. SAS software is well suited for this purpose. Through the exploitation of Object-Oriented Programming (OOP), generic tools can be written using SAS Screen Control Language. This could be done without OOP technology; however, it's extremely useful for reusable encapsulated code and highly recommended. Consequently, we have saved a significant amount of time in our development process. OOP also promotes consistency between applications in both programming techniques and user interfaces.

You will find useful information and concepts in this paper to help you design your own portable, reusable, and flexible tools.

E-MAIL INTERFACE (and small host-specific layer)

One of the most common and useful tools is the e-mail interface. It is a superb example of portable code, and like the other tools discussed in this paper, the mail interface is implemented as a CLASS.

The E-MAIL class is very useful for sending mail from any application on any platform. Since the mail packages installed on different platforms are unique in their command structure, the E-MAIL class must be able to execute the correct command for the applicable platform. However, the class interface as well as the user interface is consistent across platforms. So if you have code that makes use of the e-mail tool on one platform, it will work on all platforms that have e-mail. There are different methods to send and receive e-mail.

268
The few interactions the class actually has with the host mailer must be coded in host-specific SCL methods. The following is an example of host-specific SCL source code for the SEND method of the E-MAIL class for two different platforms.

**UNIX:**
```scl
send: method optional = in out retcode 8;
/* Gather input information */
hostf = getnitemc(in, 'HOSTFILE')
to = getnitemc(in, 'TO');
subj = insertc(subl, 'Subject: ');
ger = insertc(subl, in, 'SUBJECT');
/*cc, bcc, replyto code removed*/
rcl = savefile('FILE', missys.mailsub', subl);
rc = delist(subl);
/* Execute the system command */
rc = system(dcat -/m missys.mailsub dl hostf mail to);
endmethod;
```

**MVS:**
```scl
Send: method optional = in out retcode 8;
/* Gather input information */
hostf = getnitemc(in, 'HOSTFILE')
to = getnitemc(in, 'TO');
subj = getnitemc(in, 'SUBJECT');
/*cc, bcc, replyto code removed*/
rcl = savefile('FILE', missys.mailsub', subl);
rc = delist(subl);
/* Execute the system command */
rc = system('MAIL SEND to ~ SUBJEDCT(' subj TIME cc bcc reply to ' FROM') hostf mail to);
endmethod;
```

The following is an example of portable SCL source code for using the SEND method of the E-MAIL class for any platform.

**Portable:**
```scl
/* Instantiate the mail object */
mailobj = instance(loadclass('MAIL'));
/* Provide some input */
parameters that tell the * / mailin = makelist();
rc = setnitemc(mailin, hostfile, 'HOSTFILE');
rc = setnitemc(mailin, subject, 'SUBJECT');
rc = setnitemc(mailin, address, 'TO');
/* Tell the mail object to send */
/* the mail through the host method */
call send(mailobj, 'SEND', mailin);
/* Clean up */
call send(mailobj, '_TERM_');
rc = delist(mailin);
```

**PRINTING INTERFACE (and small host-specific layer)**

Getting a file to a printer is always a chore to learn on a new operating system, especially when you want to send data directly to the printer, rather than storing data in an external file to print with system commands. The following sample SCL illustrates how to use the same interface on several operating systems:

**UNIX:**
```scl
/* open a fileref to a printer */
opprint: method optional = in out retcode 8;
/* get any supplied options */
options = getnitemc(in, 'OPTIONS', 1, 1, 0);
/* convert the list of options */
/* into a valid string of options */
opts = ' ';
if listlen(options) > 0 then do;
do i = 1 to listlen(options);
 opts = opts || ' ' ||
getitemc(options, i);
end:
end;
```

**MVS:**
```scl
opprint: method optional = in out retcode 8;
options = getnitemc(in, 'OPTIONS', 1, 1, 0);
/* Get options */
/* Use the REMOTE exec to set */
/* device (Prt) and options */
if listlen(options) > 0 then do;
do i = 1 to listlen(options);
 opts = opts || ' ' ||
getitemc(options, i) || ' ';
end:
end;
```

**CMS:**
```scl
opprint: method optional = in out retcode 8;
options = getnitemc(in, 'OPTIONS', 1, 1, 0);
opts = ' ';
/* Get options */
/* Use the REMOTE exec to set */
/* device (Prt) and options */
if listlen(options) >as 0 then do;
do i = 1 to listlen(options);
 opts = opts || ' ' ||
getitemc(options, i) || ' ';
end:
end;
printer = getnitemc(in, 'PRINTER', 1, 1, 1);
fileref = getnitemc(in, 'FILEREP', 1, 1, 1);
if printer ne ' ' then do:
rc = fiename(fileref, 'nlp -d' ||
 printer || ' -o ' || opts ||'
', 'PIPE');
e = 1;
end:
end;
```

**CMS:**
```scl
opprint: method optional = in out retcode 8;
options = getnitemc(in, 'OPTIONS', 1, 1, 0);
opts = ' ';
/* Get options */
/* Use the REMOTE exec to set */
/* device (Prt) and options */
if listlen(options) > 0 then do;
do i = 1 to listlen(options);
 opts = opts || ' ' ||
getitemc(options, i) || ' ';
end:
end;
printer = getnitemc(in, 'PRINTER', 1, 1, 1);
fileref = getnitemc(in, 'FILEREP', 1, 1, 1);
if printer ne ' ' then do:
rc = system('CP SPOOL PRINTER FORM STD NAME
```
The following is an example of portable SCL source code for using the OPNPRINT method of HOST.SCL for any platform.

**Portable:**

```scl
in = makelist();
rc = setnitemc(in, prtdest, 'PRINTER');
rc = setnitemc(in, 'OOTFILE', 'FILEREF');
rc = setnitemc(in, options, 'OPTIONS');
/* let the host routines open */
/* that fileref to that printer */
/* with those options */
call method('host.scl', 'OPNPRINT',
  in, 0, retcode);
/* delete all temporary lists */
rc = dellist(options);
rc = dellist(in);
msg = 'Printer was not successfully accessed.';
if retcode then do;
  /* try to open the printer */
  prsset = open(prtfile);
call set(prsset);
printer = fopen('OUTFILE', 'O', 0, 'P');
  tof = '1';
/* Write lines to OUTFILE to send */
/* them directly to the printer. */
  rc = fput(printer, 'This is test text');
  rc = fput(printer, tof);
  rc = fclose(printer);
end:
return;
```

### DATA ENTRY VALIDATION

The DATASET class is a portable tool that performs data set lookup for data validation for a value or a list of values. You can specify a WHERE clause to apply to the lookup data set, or send in a list of variables and variables to build a WHERE clause. Wildcards are supported and there is an option to ignore case. When the data set lookup tool builds a WHERE clause, the values are joined together by an AND operator by default, but you can specify that the values be joined with an OR operator instead. If there is more than one match, a list of matches is given for you to select from. This is useful for finding a person's name using a wildcard. You can select the exact name without having to know the exact spelling. For example, a user may want a selection list of all people whose last name begins with 'Johns'.

The DATASET class lets you
- specify which variables from the data set observation to return,
- specify which variables to show for any selection list if there is more than one match to the WHERE clause,
- specify an alternate set of values to look up in case the original specifications are not found.

With so many options, the input parameters can get fairly complex. To keep the input parameters flexible and somewhat simple, the parameters are sent in as a list. The input list to the lookup method of the data set class has the following parameters. Each parameter is a NAMED item in the input list, and the corresponding value is the parameter value.

**DSNAME** - The name of the data set to use for data entry validation and lookup.

**BATCH** - Indicates whether this is a batch or interactive invocation. This allows nonvisual objects to also run in batch mode with the same data validation tools. If a batch run is indicated, the DATASET class does not pop up any selection lists but merely returns a blank for the value if it did not validate. If it did validate, the value is returned.

**VALUES** - Each named item is the variable name, and the value is the value you want to find. Example setup of this list:

```scl
rc = setnitemc(val_list, 'johns*', 'NAME');
rc = setnitemc(val_list, 'wes*', 'NAME');
rc = setnitemc(val_list, 'son', 'NAME');
```

**SHOW** - List of variables to show for a selection list window. Example setup:

```scl
rc = insertc(show, 'NAME');
rc = insertc(show, 'OFFICE');
```

**RETURN** - List of variables to return to calling program when a match is found. Example setup:

```scl
rc = insertc(return, 'PHONE');
rc = insertc(return, 'IDNUM');
```

**NEEDEDONE** - specifies that the values will be joined by an OR operator. Based on the above VALUES list, the WHERE clause built
DYNAMIC FREE FORM TEXT ENTRY/UNLIMITED COMMENTS

Many applications need to have comments or annotations associated with observations in a data set. The quickest way to do this is to put several character variables in the data set, such as comment1-comment5, all of length 80. This works fine if you only need 5 or less comment lines. But what if you need more? Adding another 2 to 3 comment variables can take up a lot of extra space, and they may not even be used most of the time, which is a waste of disk space. Also, you cannot tell who added the comments or when.

The COMMENTS class is a portable tool that is used to enable a user to enter an unlimited amount of free-form text via the SAS preview window. The user's comments are then captured and saved in a SAS data set. This provides full editor capabilities for the user. This is extremely useful for applications that need some kind of logging capabilities. For example, a user may need to document some research or a phone conversation. Each comment is time/date/user stamped so it is easy to see who entered each comment and when it was entered. Old comments can be edited and changed while an infinite number of new comments can also be added. Each group of comments is associated with a particular application entity (could be an observation, object, subset, and so on). This class is completely portable.

Sample SCL Code:

```sas
/* Instantiate the comments */
/* object */
tin = makelist();
tout = makelist();
rc = setnitemc(tin, 'SYSCMNTX', 'NAME');
rc = setnitemc(tin, 'SYSCMNTS', 'TEXT');
rc = setnitemc(tin, 'WRITE=ABC', 'OPTIONS');
rc = setnitemc(tin, 'U', 'MODE');
comments = instance(loadclass('COMMENTS'), tin);

/* Load all comments for this key */
call send(comments, 'LOAD', tin);

/* Now edit the loaded comments */
call send(comments, 'EDIT', tin, tout);

/* Clean up */
call send(comments, '_TERM_', tin);
rc = dellist(tin);
rc = dellist(tout);
```
METABASE FOR APPLICATION RESOURCE MANAGEMENT

An easy way to keep track of all applications on a platform is to have some sort of centralized application launching program. In MIS we have such a utility, called MISSYS, and it lives on several different operating systems. The libraries associated with MISSYS house the portable tools and a metabase that describes the structure of each application and handles program maintenance. Each operating system has its own host-specific copy. The metabase consists of several data sets:

• A libref allocation data set which holds information necessary to issue librefs. The data is keyed by application name.

• A catalog search path data set keeps the catalog names for each libref, subset by application name. When MISSYS issues a libname, it also adds all the catalogs in the libref to the AF search path. This data set prevents that, and only uses the catalogs you specify. This is useful when you have a large format catalog that will only waste time being in the search path.

• A system defaults data set has application specific information, such as a brief description of the application and a default catalog entry that is the starting point for an application.

• A check in/out data set. When enhancing existing catalog entries, a developer must lock an entry to keep others from modifying the same code.

• Other commonly used data sets are valid printers, valid employees, and many other validation and lookup data sets.

There are four major advantages gained by using a metabase for application resource management:

1) Easy Maintenance: The applications developer never has to specify a libref or a hostfile name in SCL code. Librefs are only specified once in a metabase belonging to MISSYS. When hostfile names change or server locations vary, changes are only made in one place.

2) Development Environment: Users may be running the production version while applications developers are maintaining or enhancing the system. The development environment also offers a way to check catalog entries in or out so that if more than one applications developer is working on the same application, conflicts from code changes are less likely to occur. MISSYS takes care of setting up the search path so developers can very easily test their code changes while leaving users unaffected.

3) Portability: The only place host files are referenced are in the metabase. This makes the system portable. Thus, SASCONNECT® or other methods may be used to port the system to another platform. Once on the new platform, only the metabase needs updating before the application is ready to run. There is no need to recompile.

4) Automatic Resource Allocation: Resources are automatically allocated when running an application under MISSYS.

Following is a description of the structure of the main data set in the metabase, the library/application cross-reference:

SYSTEM specifies the name of the application or “accessed” system. (For example, DEMO, PAYROLL, and so on)

TYPE specifies which libraries are to be allocated when an application runs in production mode and which additional libraries should be allocated for development and test modes. For production libraries, the value of TYPE is PRODUCTION. For development libraries, the value of TYPE is anything else. For example, if you have two developers working on an application, each would want their own development librefs allocated but not the other developer’s librefs. The first developer may specify his development area with a type of TLAPEN, while the second developer may use a type of JZTPEN.
NAME specifies the named component of the application (for example FSEDIT SCREENS, AF CATALOGS, FORMATS, DATA).

LIBREF specifies the libref you want allocated with the hostfile. If you take full advantage of MISSYS, this will be the only place you need to specify the libref. (for example LIBRARY, DEMO, DEMODATA)

SERVER specifies the SAS/SHARE® software server alias or the actual server name. If it is left blank, no server will be used for the library allocation.

HOSTFILE specifies the actual hostfile of the libref.

OPTIONS specifies any allocation options to use when allocating the hostfile. This parameter is optional and is only filled in if options are needed. (for example, ACCESS=READONLY, RMTVIEW=NO)

ENGINE specifies the name of a SAS access engine if needed. This is optional and only need be filled in if an engine other than the default engine is needed.

An SCL program handles all library allocations using the above data structure. At the same time, the AF search path is constructed. The AF search path allows programs to always use two-level names when calling other SCL programs. You never have to code in a specific libref name in SCL code. This enables you to test changes to applications without altering the SCL code to point to a test library. Once a modification is tested, it can be moved directly into production without recompiling or changing back any libref names in the entry.

Following are the events in the library allocation program:
1) Find the first observation that matches the given application name.

2) Allocate the library based on libref, Server Name, Engine, Libref Options, and host file.

3) Using the TYPE information, determine where in the AF Catalog search path to put any catalogs in this library. A special list keeps track of what is production and what is not. All production catalogs are at the bottom of the list, and all development catalogs are added to the top. After all catalogs are put into the list, reconstruct the AF search path based on the updated list.

4) Add the following information to a local environment list. This list is an implementation of a double-linked list of all allocated libraries. Think of each libref as being a node in this list:
   - Libref Name
   - Type Level - all librefs of the same type have the same type level value.
   - Pointer to the previous libref of the same TYPE (PRODUCTION, TLAPEN, and so on)
   - Pointer to the next libref of the same TYPE to be allocated. This is initially null, and will be updated when another libref of the same TYPE is allocated.
   - Pointer to previous libref of the same Component Name (DATA, AF CATALOGS, and so on).
   - Pointer to the next libref of the same Component Name. This is initially null.

   In the above step, a pointer is just the list id of the relevant list. The setiteml and getiteml SCL functions set and retrieve the list ids.

5) Update the null pointers in the previously allocated libref for both the TYPE and Component Name.

   This list is important for check in/check out of production catalog entries. It will help route the right entries to the right catalogs.

   The DATASET class also uses this list for the data set search path mentioned on page 4. To find a one-level data set name, the DATASET tool looks through all Components named DATA, starting with the development or test librefs. If a libref has a matching data set (using the exists SCL function), then the search is over. Otherwise, it follows the pointer to the next Component named DATA, until all of them are searched.

   **CATALOG ENTRY CHECK IN / CHECK OUT**

   The two primary reasons for a check in / check out procedure for catalog entries are maintenance and testing. During the development of large applications, more than one developer may be involved with the project. In order to keep multiple developers from changing the same code, a locking mechanism is necessary. With MISSYS, the developer checks out the entry needed, makes changes to the checked-out copy in his/her playpen, then compiles and tests the entries without affecting any other developers or users. If other developers try to work with the same entry from the same application, MISSYS tells them the entry is already locked and gives the time/date/who has it locked information. This replaces coding coordination efforts that are tedious and error prone. Display 2 shows a picture of this screen.

   Definitions for user/auto filled fields:

   **SYSTEM** specifies the name of the application.

   **ENTRY** specifies the catalog entry that starts the application and is specified in the metabase. If not specified in the system defaults data set, the default is MAIN.SCL. It is better to use a more meaningful entry name since you can have several applications accessed at once, and MAIN.SCL could be the starting point for several of them.

   **PARM** specifies any parameters to send to the entry point code. This is useful for sending initial parameters to an application. You cannot use SYSPARM to do this because MISSYS uses SYSPARM.
ALLOC TYPE specifies the TYPE of the library you are describing to MISSYS. This is equivalent to the TYPE specified for this library in the metabase. Under Production Location this is PRODUCTION. The Development Location equals the TYPE in the metabase used to describe the developer's development library (for example PRODUCTION, TLAPEN).

ALLOC NAME specifies the current named component of the application and corresponds to the NAME specified for this library in the metabase.

LIBREF NAME specifies the libref that has been used to allocate the library you are using. It is the libref specified in the metabase for the current TYPE and Named Component. This will be automatically filled by MISSYS and is protected.

CATALOG specifies the name of the catalog you want to build. You can specify any catalog that is allocated.

NAME specifies the current entry name. Other actions described below will work on the current entry. This is the name given in the specified catalog without the type indicator. MAIN.SCL would be specified here as MAIN.

TYPE specifies the type of the entry you specified in NAME. MAIN.SCL would be specified here as SCL.

NEW NAME specifies the new name of the entry you want to rename. This is the name given in the specified catalog without the type indicator.

The following describes various operations (things a developer can do to a system) in the
MISSYS development window:

GET checks out the production entry specified by CATALOG.NAME.TYPE into the development catalog. When an entry is checked out, no one else can edit it.

REPLACE checks in the development entry specified by CATALOG.NAME.TYPE to production. When the entry is checked back in, other developers can edit the entry.

UNLOCK cancels the check out of the specified entry. The development copy is deleted.

ADD moves a new entry into production. No locking/unlocking is done since the element is new.

DELETE removes an entry from production.

RENAME renames the element specified by CATALOG.NAME.TYPE entry in the production catalog. The new element name is the value in the New Name field.

UPDATE copies the element specified by CATALOG.NAME.TYPE from the development catalog to the production catalog. The development copy is kept and the lock is not removed. This comes in handy when a fix needs to be pushed into production immediately but there is more work to be done on the same element.

BUILD issues the build command for the given system and development catalog.

EXECUTE runs the system specified in the SYSTEM field at the entry specified in the ENTRY field.

APPLICATION MODE CONTROL AT RUN TIME

Different parameters determine whether the application runs in test, production, or development modes. SCL can also be used to create a very useful development environment for single or multiple programmer application development. This creates an automated and uniform way of running applications in production, stage, development, or any other mode. This enables developers to test multiple changes without impacting the users in any way, and also provides a central depository for tool and data access.

By specifying a variety of options upon invoking MISSYS, the appropriate kinds of libraries are allocated and the search path is set up accordingly. The following is a description of some of the MISSYS options:

AFOPT 'AF OPTIONS' specifies options to pass into the AF command that will start your application. The AF options must be passed in as a quoted string.

ADDITIONAL SAS OR SYSTEM OPTIONS specify SAS or operating system options. After all the regular MISSYS options are specified, you can specify whatever SAS options you may need. This option must be the last one if it is used.

BAT runs MISSYS in non-interactive mode, and sets options to allow running in batch. Classes are smart enough to detect batch mode and not pop up any windows. For example, the Dataset class mentioned earlier, would not pop up any selection lists for data validation. It would return a blank value if the data did not validate.

ENTRY executes the specified entry first after allocating all appropriate libraries. If no entry is specified, a default of MAIN.SCL is used unless the application has a default entry point defined in the system defaults data set.

HOSTGEN specifies a host file to use to override the default MISSYS library, and use the specified file path instead.

HOSTBOOT specifies a host file to use to override the SAS autoexec for MISSYS.

INTSTART specifies a host file to use to override the default metabase.

PARM passes the value of a parameter to the application's entry point. The default is to not pass any parameter.

PEN allocates a SAS library or suite of libraries to be used as a development area. Catalogs
in the development area are inserted at the top of the catalog search path, which enables changes to be tested before putting them into production.

**SYSTEM** specifies the application to run. If no name is given, then **MISSYS** runs in development mode, enabling developers to make and test changes.

**UNIX Examples:**

To invoke the **DEMO** application, enter

```
missys -system demo
```

Invoke **DEMO** with debug mode on, a title of 'Demo', and use Icon 23, enter

```
missys -system demo -afopt 'debug=yes title=Demo icon=23'
```

To do development work on any application, enter **missys**.

To invoke **MISSYS** in develop mode, but with a big font: **missys** `-xrm SAS.DMSFont: 12x24 -xrm SAS.DMSboldFont:12x24`

**CONCLUSION**

This paper discussed several tools that are portable and generic enough to be used by diverse applications. The keys to creating portable tools are to keep programming and user interfaces consistent across platforms, and to keep host-specific code isolated. Using a metabase and the Object-Oriented paradigm makes this task easy to manage.

Generic portable tools save a lot of development time and effort on any programming project. Although there is an initial investment of tool development time, the long-term returns are well worth the effort. Reusable code prevents the developers from having to reinvent the wheel for each application that is produced. Generic tools also help with standardizing applications at a particular company; users become accustomed to the look and feel of the tool interfaces and do not have to relearn different interfaces for different applications.

There is nothing revolutionary in these tools. They are the same types of tools that every experienced application developer has written or used. However, **SAS Screen Control Language** and **Object-Oriented Programming** make it easy to implement useful and reusable tools that work.

When designing and developing your own portable tools to solve complex problems, step back and view the problem from different perspectives, and have confidence that **SAS software** can provide a time saving, portable solution.

**SAS, SAS/CONNECT, and SAS/SHARE software** are registered trademarks or trademarks of **SAS Institute Inc.** in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.