# Data-Independent Application Development with SAS/AF® Software FRAME Entry

John Leveille and Don Williams

## ABSTRACT

One of the great benefits of Object-Oriented Programming (OOP) is the ability to create applications that are independent of the data they manipulate. Recent releases of the SAS System have realized this capability through the SAS/AF Software FRAME Entry.

This paper outlines techniques for developing object-oriented applications by referencing an actual application recently developed at SAS Institute. This paper describes the implementation and benefits of an object-oriented Model-View-Controller (MVC) approach for displaying data. The benefits include presenting data from a single SAS data set in a list and form view simultaneously. Flexible field and data set modeling allows the physical structures of the application data to change without affecting the application. Variable types and lengths can change, new variables can be added, and variables can be removed from the SAS data set without the need to redesign, recode, or recompile. Data from multiple SAS data sets can be joined and displayed on the same form seamlessly. The design and implementation techniques discussed illustrate how you can create an application that accommodates change with reduced maintenance cost and effort. In addition to utilizing SAS/AF Software FRAME entries, the application presented here also uses Screen Control Language (SCL) and SAS/SHARE® Software.

## INTRODUCTION

The phone rings. A demanding voice on the other end says, "The Inventory Tracking application needs five new fields as soon as possible. And, while you're at it, two other fields need to be wider and another field needs six new valid values." As he hangs up the phone, a knot begins to form in the application programmer's belly. He groans and reaches for the bottle of Maalox on his desk as his quick tally of the number of source files affected by this change reaches 20. He would like to just say "yes," but he fears that the cost of changing the application will be prohibitive.

Scenarios like this illustrate the need to design applications that rely as little as possible on the structure of their data. Too often, seemingly minor modifications and enhancements cause ripples that spread throughout an application like ripples in a pond. Obviously, this application programmer could be more responsive to his clients if he had an application that could accommodate inevitable changes in its data structure. Throughout this paper, the term *data-independent* will be used to describe such applications.

More specifically, this paper describes:

- the object-oriented paradigm
- the SAS System feature set
- choosing an OOP strategy for implementing the DEFECTS 2.0 application
- implementing data-independence in DEFECTS 2.0.

Of course, no application can be completely independent of its data. This would probably render the application useless. There is, however, a belief that using techniques to construct a data-independent application decreases its maintenance costs, increases its longevity, and enhances its value. Using an object-oriented approach offers application programmers the opportunity to construct data-independent applications. Before discussing an example application constructed using this approach—the DEFECTS 2.application—a brief discussion of the object-oriented paradigm is required.

### The Object-Oriented Paradigm

Until recently, *structured programming* was the most prevalent philosophy used in constructing an application. This philosophy typically involves the use of a main routine and a series of supporting routines (i.e., functions or procedures). By having a library of routines, an application programmer could structure code in a modular fashion, thereby constructing an application from these building blocks. Languages such as PL/1 and C are highly conducive to creating applications using a structured programming strategy.

Structured programming is generally agreed to be an improvement over the unstructured techniques used when programming was in its infancy. Unstructured programming does not have modular concepts; it often leads to so-called spaghetti code. Without routines with clearly defined entry and exit points, logic can jump back and forth throughout the program. This jumping often results in a tangled flow of control that is generally difficult to follow, even for the original author of the program. This difficulty in understanding the possible control flow within an unstructured program has disastrous consequences for code changes. Frequently, the lack of understanding the impact of an error correction or maintenance request makes code changes prohibitive, or worse, results in code that executes with undesirable consequences.

With structured programming, the impact of changing code is better understood. The impact of an edit is easier to follow as a result of the identifiable routines that are involved. However, because of the reliance on parameter passing as the mechanism to communicate values to and from these routines, a change to one routine's parameter set can affect other routines throughout the application. As a result a ripple effect is created throughout the application, forcing multiple routines to be edited. Of course, the more routines that are edited, the greater the risk of creating new errors. By its nature, structured programming causes the application programmer to focus attention on these routines or functions. By focusing predominantly on the flow of execution, structured programs often fail to model or resemble the real problem domain they attempt to represent.

Recently, an alternative philosophy to structured programming has been gaining support. Its primary focus is on the objects involved in the problem domain. This object-oriented paradigm involves careful analysis of the objects in the problem domain (Object-Oriented Analysis (OOA)), design of the application based on these objects (Object-Oriented Design (OOD)), and programming the application (Object-Oriented Programming (OOP)) to implement the objects that have been abstracted from the problem domain. An object-oriented approach begins first with the question of "What?", instead of "How?". Exactly "what" an object is has been explained in various ways:

- an abstraction of a set of real-world things (Shlaer/Mellor 1988, p.14)
- an object has state, behavior, and identity (Booch 1991, p.77)
- an abstraction of something in a problem domain reflecting the capabilities of the system to keep information about it, interact with it, or both (Coad/Yourdon 1991, p.53)
- anything, real or abstract, about which we store data and those methods that manipulate the data (Martin/Odell 1991, p.16).

Objects are said to have characteristics, primarily:

- identity—a value that distinguishes one object from another
- properties—a collection of attributes and their values. These are generally termed *instance variables* (IVs).
- behaviors—a collection of operations or services that the object can perform on itself or other objects. Objects are requested to perform these behaviors through *messages*. (Semaphore 1993, p.9)

As real-world objects are grouped or differentiated by their characteristics, so too are objects within this paradigm. Classes of objects can be formed based on shared characteristics, and, if needed, these classes can be subclassed to group objects with specific properties or behaviors. Class hierarchies can be constructed, thus allowing objects within subclasses to inherit properties and behaviors from parent classes. This is a most valuable asset because it allows general (super) classes to be defined, and, if needed, more specialized (sub) classes to be defined.

By focusing on an object, this paradigm allows the programming solution (abstraction) to be *instantiated* in such a way that it closely resembles the problem domain. Supporters of the object-oriented paradigm assert this has many benefits, notably:

- reusability of classes (through collections of classes into class libraries) from one application to another
- improved ability to respond to code changes because the impact of a code change is easier to recognize (it affects specific class hierarchies), and, due to subclassing, impact of code change can either be as narrow or broad as desired
- well-suited to handle the programming challenges linked to Graphical User Interfaces (GUIs).

These perceived benefits are demonstrated by the number of languages and tools that are now available to support an object-oriented paradigm. Implementation languages and tools include: Smalltalk; Eiffel; Ada; CLOS; Turbo Pascal; Simula; Modula-3; Object Oberon; Enfin; Actor; Objective-C and SAS Institute's SAS/AF Software FRAME Entry and object-oriented extentions to SCL (Semaphore 1993, Appendix D, p.3). Differences among these languages and tools exceed the scope of this paper, as does a more complete discussion of the object-oriented paradigm.

**SAS System OOP Feature Set**

For the purpose of discussing OOP features within the SAS System, general experience with Base SAS® software and SAS/AF Software is assumed.

Specifically, this discussion assumes familiarity with:

- SAS data sets
- SAS catalogs
- Screen Control Language (SCL)
- SCL lists
- the application development facility (PROC BUILD)
- SAS/AF Software FRAME Entry.

A brief summary of the history of OOP support within the SAS System includes these landmarks:

- SAS System Releases 6.08 and 6.09
  FRAME Entry technology available for object-oriented windowing applications
  library of widget classes
  support for developing and storing class definitions
  classes are not true objects, only SCL list identifiers
- SAS System Release 6.10
  introduction of dynamic (linked) inheritance
  classes are true objects
- SAS System Release 6.11
  greatly expanded class library (Figure 1)
  capability to treat classes as objects at runtime
  improved class editor
  support for composite widgets
  drag and drop support
  expanded SCL features.
  (SAS Institute, 1995, p.43)

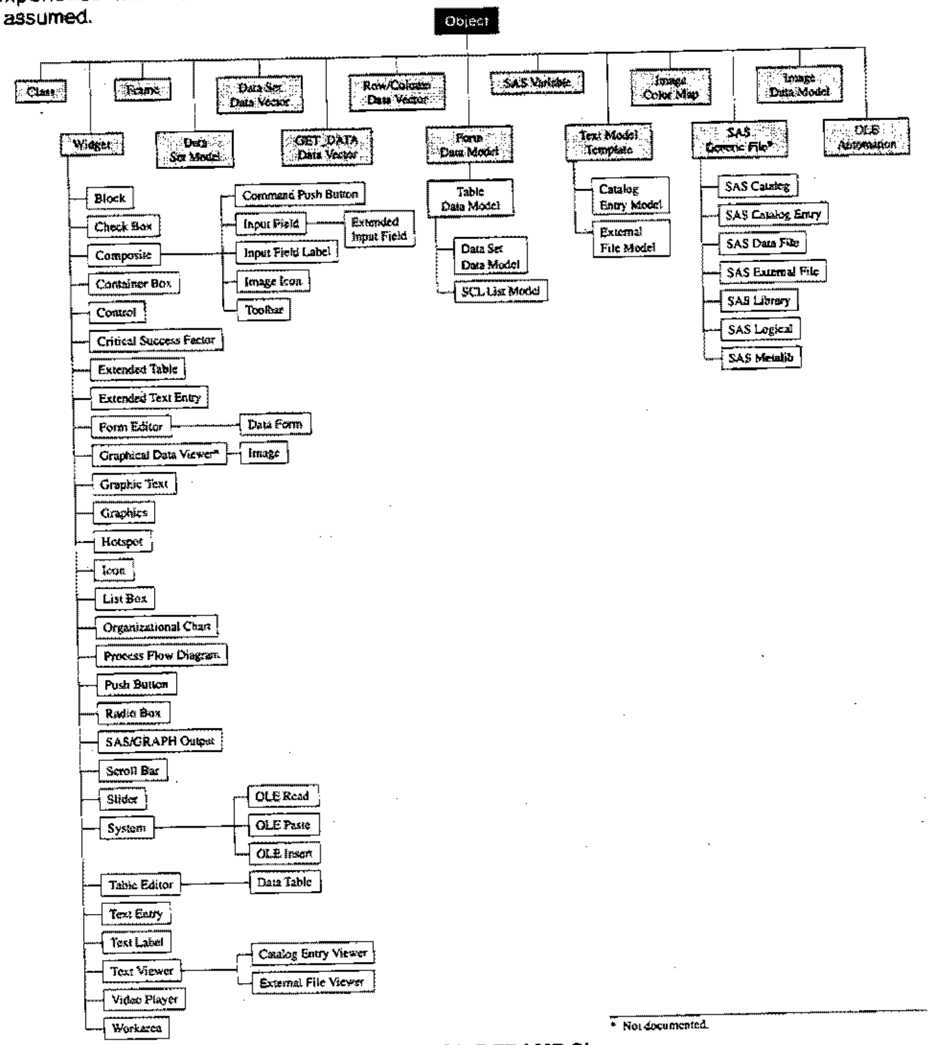


**Figure 1. SAS/AF FRAME Classes**

Objects and classes are implemented through extensions to SCL. In the SAS System:

- all classes (both supplied and user-defined) form a single-inheritance hierarchy.
- all classes are derived from a root class called the Object class, which has no parent class.
- the Object class supplies very basic methods for general object manipulation, including object creation (_INIT_) and destruction (_TERM_).
  (SAS Institute, 1995, p.43)

Classes can be defined at runtime with an SCL program or at development time with the SAS/AF Class Entry window. Classes can define instance variables, which can either be inherited from a parent class, or new.

These instance variables can be one of three types:

- character
- numeric
- SCL list, which can include character or numeric values or identifiers to other SCL lists.

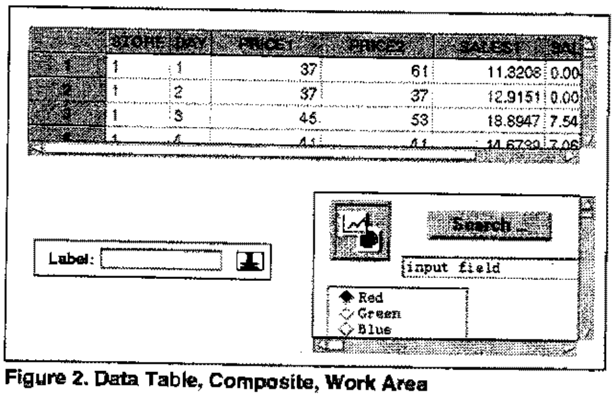
Classes can have methods. Method calls are the messaging protocol used by objects to communicate requests for service. Methods for a class can be:

- new to that class
- inherited from a parent class
- overriden from a parent class
- disabled.

SAS/AF software offers two major subclasses of the Object class that are particularly valuable when implementing applications with GUIs—the Frame class and the Widget class. Objects of the Frame class are used to manage application windows. The Frame class can be subclassed, and in Release 6.11 of the SAS System has been provided with over 80 available methods.

A widget is an object that is visible and can be instantiated on a frame. The SAS/AF Software Widget class cannot be directly instantiated because it is a *virtual class*. From the Widget class, other widgets are subclassed and those subclasses can be instantiated. Release 6.11 of the SAS System includes over 40 subclasses of the Widget class. The following three widget classes were valuable in the data-independent implementation of the DEFECTS application:

- **Data Table**
- **Composite**
- **Work Area.**



| | STORE | DAY | PRICE1 | PRICE2 | SALES1 | SAL |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 37 | 61 | 11.3208 | 0.00 |
| 2 | 1 | 2 | 37 | 37 | 12.9151 | 0.00 |
| 3 | 1 | 3 | 45 | 53 | 18.8947 | 7.54 |
| 4 | 1 | 4 | 41 | 41 | 14.6739 | 7.06 |

**Figure 2. Data Table, Composite, Work Area**

The Data Table is an widget that can display and manipulate a SAS data set given only its data set name. It functions in a data-independent manner because it has no permanent knowledge of the data set structure. Each time the Data Table is instantiated, it reads and displays the data set, modeling the current structure. The data set structure can change and the Data Table accommodates those changes. In addition, columns in the Data Table can be moved, resized, hidden, and unhidden without affecting the physical data set.

The Composite widget is a collection of widgets that act as a single widget. This widget is useful for collecting application functionality into components. Futhermore, these components can be utilized and managed as a whole instead of as many pieces.

The Work Area is a virtual desktop. On it, you can move, resize, add, and remove widgets interactively. The Work Area is data-independent because it has no knowledge of the data that can be placed upon it. The Work Area is not associated with a SAS data set, whereas an FSEDIT screen must be. The Work Area simply allows the placement of widgets and facilitates user interaction. If the widgets on the Work Area contain data, then the Work Area functions as a customizable viewer.

In short, OOP features within Release 6.11 of the SAS System are generous and enable representative OOP implementations. The capabilities of the 6.11 feature set make it possible to consider using the OOP paradigm to address the specific demands of the application being implemented. There are numerous other elements of the 6.11 OOP feature set that cannot be discussed in the physical constraints of this paper. Refer to *SAS/AF Software: FRAME Class Dictionary, Version 6, First Edition* for a more complete introduction to the 6.11 OOP feature set.

**Choosing an OOP Strategy for Implementing the DEFECTS 2.0 Application**

SAS Institute's DEFECTS tracking application, version 1.5, was nearing the end of its lifecycle. It was implemented using a structured programming strategy and developed for the MVS platform using Release 6.08 of the SAS System. It functioned primarily as a data entry and database application because it allowed SAS Institute staff to add, browse, update, query and report against defects recorded during development of Institute product offerings. DEFECTS 1.5 was implemented prior to the availability of SAS/AF Software FRAME entries and, consequently, utilized PROC FSEDIT screens, SAS/AF PROGRAM entries and SCL entries to deliver its windowing features. Due to the importance of the data being tracked, the application's primary SAS data set was highly validated, and various relationships were defined among critical data fields. As a result, a change to one field's value might impart a change to another field's value. The ripple effect previously described often occurred with maintenance to the DEFECTS application.

In addition to data fields, another component of DEFECTS 1.5 was its need to associate "funny" objects with the defect entry. These funny objects included the text log, test program, and history. The text log recorded the description of the defect and the developer's response to the defect entry. The test program was a text object much like the text log. Finally, due to the possibility that a particular defect might exist in multiple locations (in specific operating systems and in several SAS System releases), the DEFECTS 1.5 application needed the ability to record the disposition of a defect across multiple locations simultaneously. This need resulted in the use of a supporting history SAS data set. Observations from this supporting history data set allowed for the status of a defect to be recorded for each specified location. The history data set was associated (joined) to the primary defect observation through the use of a unique key value (DEFECTID).

In general, the DEFECTS 1.5 application functioned well during its lifespan. As it aged, limitations of the application became more pronounced. Some of these limitations included:

- changes often required editing of many catalog entries with the associated inherent risk of introducing undesired side effects
- user customizations were limited and difficult to retrofit into the application
- batch processing was unavailable for the general user

- maintenance batch processing did not utilize the same code as the interactive application
- certain display widgets and GUI features were absent and users were requesting them.

Many of these limitations revolve around a defect's data and the relationships among the various components of a defect. Consequently, an OOP strategy was choosen for the DEFECTS 2.0 implementation to achieve data-independence whenever possible. By implementing data-independent objects, the DEFECTS 2.0 application:

- accommodates a very robust hierarchical data structure
- allows users to highly customize the GUI with individual preferences
- effectively communicates the data structure to users, thereby facilitating proper interpretation, reporting, and analysis
- reduces maintenance costs in responding to new/unforeseen changes in the data structure.

**Implementing Data-Independence in DEFECTS 2.0**

One of the central components of the design of the DEFECTS 2.0 application is the necessity to model hierarchical data structures. The application requires a two-tier hierarchy.

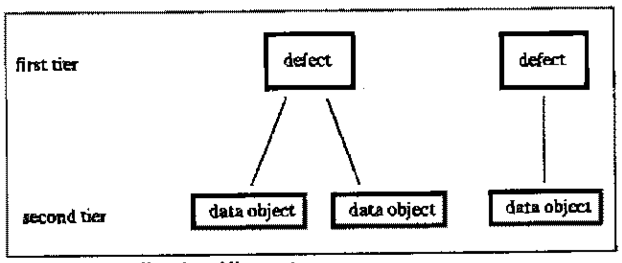


**Figure 3. Application Hierarchy**

At the first tier (the root) are the defect entries. Each defect entry is uniquely identified as an observation in a SAS data set and can own an assortment of other data objects. These second-tier data objects may include observations in other SAS data sets, text files, or graphic images or possibly any other type of data object that can be defined to have a relationship to the primary tier. There is no limit to the number or type of second-tier data objects that can be owned by a defect entry. Thus, this structure accommodates either a one-to-one or a one-to-many relationship. More important, it allows for the future addition of data objects that are not defined in the initial implementation.

Viewing a data structure like that of the DEFECTS 2.0 application is possible in a non-object-oriented design, but can often require hard-coded relationships among the data sets and the construction of large and complex programs to manage the entire application's data relationships. An object-oriented design overcomes many of the difficulties present in modeling this type of data structure.

In order to model a collection of data sets with a certain relationship, we started with an object class that could model a single SAS data set. SAS/AF Software provides the DATA_M class for this purpose. The Data Table widget described earlier is actually more than one object working in concert. There is a TABLE_E widget, which is the displayable data viewer, and a DATA_M object, which is the non-displayable data modeler. The DATA_M object can exist outside of a Data Table instance. A DATA_M object has the ability to read records from a SAS data set, organize the variables as columns in any order, hide and unhide columns, and subset the records based on some criteria. It is important to point out that the DATA_M object does not need the program to tell it what variables are in the SAS data set; it only needs the data set name. With this functionality, a collection of DATA_M objects could model the DEFECTS 2.0 hierarchical data structure. Our design was for one DATA_M object to model the main SAS data set at the first tier and for multiple DATA_M objects to model SAS data sets at the second tier.

In order to keep the application data-independent, we endowed the DATA_M objects with as little information about the data as possible. Thus far, each DATA_M object only knows the name of the SAS data set it models. For clarity, let us assign a name to these DATA_M objects. The first-tier DATA_M object is named **MAIN** and the second-tier DATA_M objects are named **TABLE**. We can define the hierarchical relationships in the application by assigning each TABLE object a key variable. The key variable is a variable common to the MAIN data set and the TABLE data set and can be used to subset the TABLE records. When a record is selected from MAIN, each TABLE subsets its records to match those sharing the same key variable value of MAIN. In object-oriented terminology, the knowledge of this relationship is *encapsulated* in the TABLE. There is no structured program external to these objects that must know the relationship. Furthermore, each relationship between MAIN and a TABLE can be unique if the application requires.

Up to this point, the data structure discussion has centered on TABLE objects modeling SAS data sets. Remember that other data objects can be easily incorporated into this type of application as needed. The DEFECTS 2.0 application has data objects other than SAS data sets, for example text files. Consequently, DEFECTS 2.0 incorporates a class that deals with text files. Given that it is unlikely that the application defines all the possible data objects it will need in the future, the application's data-independent design allows future data objects to be accommodated with minimal cost.

Before focusing more closely on the objects that enable data-independence in the DEFECTS 2.0 application, it is necessary to understand an object-oriented concept called Model-View-Controller (MVC). MVC is a way of creating an object that can handle a complex task in an orderly and segregated manner. An MVC implementation can be one object containing various layers in its functionality or more than one object cooperating to accomplish a task (the Data Table object is an example of the latter). The responsibility of the Modeler in the MVC concept is to model the data or environment. The responsibility of the Viewer is to display the output or results and to possibly receive user input. The responsibility of the Controller is to negotiate the interaction of the Viewer and Modeler. The Controller can enforce rules that may apply to the interaction and can translate events and messages.

The MVC concept has several advantages. The code that enables the functionality of the MVC as a whole is modularized. This makes the code easier to maintain, enhance, and transfer to other projects. If the Modeler and Viewer are separate objects, the Modeler can function in a batch mode without the Viewer. It is advantageous to run common code for interactive and batch versions of an application. Finally, the responsibility and functionality of each data object is distributed, which is sound object-oriented design.

Designing the user interface for the DEFECTS 2.0 application was one of the most difficult tasks of creating the application. However, as a result of the underlying object-oriented, data-independent design, there were many options available. The starting point for viewing the DEFECTS 2.0 data is the first-tier SAS data set—the MAIN data set defined previously. Historically, data sets have been viewed as a list (e.g., FSVIEW or SAS/AF extended tables), or one record at a time in a form view (e.g., an FSEDIT screen). The first type of view provides a table-like multiple record view, and the second type of view provides the ability to view a single record's data in a specific arrangement. Rather than accommodating only one of these view modes, we chose to offer users a list (Figure 4), a form (Figure 5), and a combined list/form view of the MAIN data set (Figure 6).

The user can switch instanteously between these view modes. The MVC concept was essential in implementing this type of versatility.

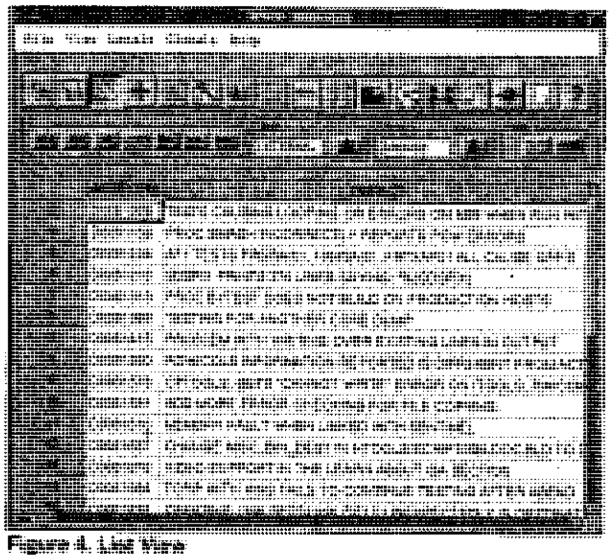


Figure 4. List View

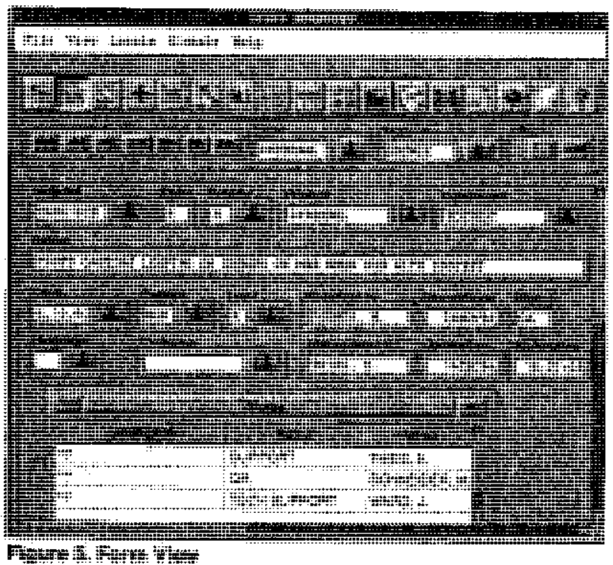


Figure 5. Form View

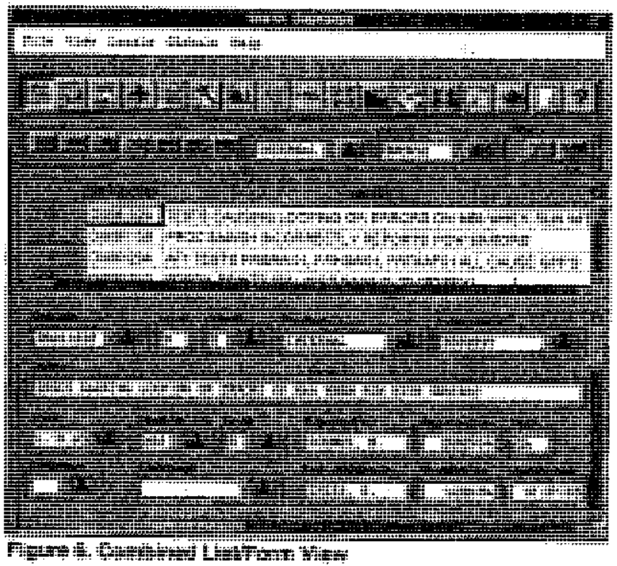


Figure 6. Combined List/Form View

In the DEFECTS 2.0 application, we chose to implement MVC using two objects instead of three objects. The Modeler object performs the duties of both the modeler and controller objects. The second object in our MVC implementation is the Viewer object. MAIN_M is the object that functions as the modeler for the main SAS data set. The MAIN_M class is a subclass of the SAS/AF DATA_M class. We extended the MAIN_M class so that it could have many viewer objects attached to it. The list view (Figure 4) utilizes SAS/AF TABLE_E class which is the same viewer used by the Data Table widget. The other viewers can attach themselves to the MAIN_M object and, combined, create the form view (Figure 5) of the MAIN data set. These other data viewer objects fall into three categories: Fields, Tables, and other data objects.

The form view for the DEFECTS 2.0 application uses FIELD objects (Figure 7) in the user interface to display and allow for updating of data.

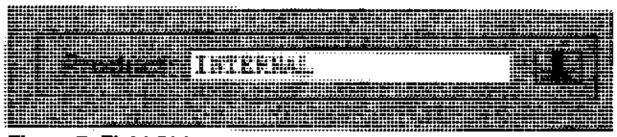

Product: INTERNAL

Figure 7. Field Object

The FIELD object could be a "dumb" viewer attached to the MAIN_M object. However, with closer examination, we see that the FIELD object should be subdivided into its own MVC relationship. The FIELD object must be more than a dumb viewer. It must validate input, relate to other FIELD and TABLE objects, and allow for users to cancel changes. Because of these design needs, the FIELD object is split into a modeler called FIELD_M, and a viewer called FIELD_V. The FIELD_M class is designed with a flexibility that plays a major role in making the DEFECTS 2.0 application data-independent. Each FIELD_M object has a set of attributes that describe the data that it models. Because it models a SAS data set variable, its attributes might be familiar:

- variable name
- type
- length
- format
- informat.

Here is the key to data-independence: the FIELD_M object ONLY stores the variable name permanently. The other variable attributes are loaded from the data set each time the application is run; more specifically, the variable attributes are loaded each time the object is instantiated. This means that a variable's type, length, format, and informat can all change and the application reflects those changes **without editing or recompiling any SCL code!** This technique for handling variable attributes can be referred to as *weak constraints*. The FIELD_M class allows for weak typing, weak storage constraints, and weak formatting. The FIELD_V class is a subclass of the SAS/AF Composite class. It is designed to attach to the FIELD_M class and respond to the current settings of its attributes. For example, the length of the input widget on the screen will change when it attaches to a FIELD_M object and this new length attribute is communicated. Some of the attributes of the FIELD_V object are received from the Modeler, as in the previous example with length, while other attributes have no bearing on the Modeler, such as color or field label font. Obviously, the Modeler functions the same whether the foreground of the field is blue or yellow.

A second type of data viewer object that can attach to the MAIN_M object is the TABLE object.

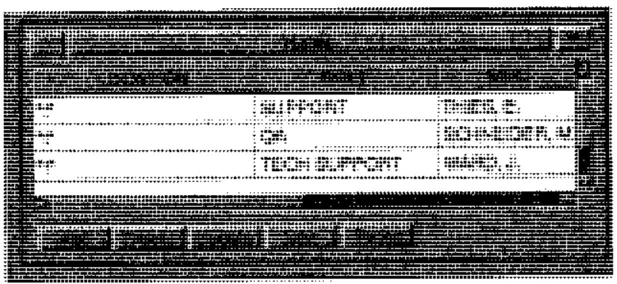


**Figure 8. Table Object**

The TABLE object was introduced in the discussion about the hierarchical structure in the DEFECTS 2.0 application. Similar to the FIELD object being split into FIELD_M and FIELD_V components to implement an MVC design, the TABLE object is split into TABLE_M and TABLE_V components. Like MAIN_M, the TABLE_M class is a subclass of the SAS/AF DATA_M class. The difference is that TABLE_M is on the second tier of the hierarchical data structure, and MAIN_M is on the first tier. Recall that the FIELD_M class allows the FIELD object to be data-independent. The same is true with the TABLE_M class because it enables the overall DEFECTS 2.0 application to be data-independent. Each TABLE_M object has a set of attributes that describe the data that it models:

- data set name
- key variable
- hidden columns list
- unhidden columns list
- column order and display widths list
- actions list.

Again, the key to data-independence is that the TABLE_M object only stores the data set name and key variable permanently. The DATA_M class handles the flexibility needed for dealing with changes in the data set. As stated before, it automatically accommodates the current data set structure each time the application runs (specifically, each time the object is instantiated). Variables can be dropped or added to the data set and the TABLE_M class does not fail. The attributes of variables in the data set can change and the TABLE_M class does not fail. **There is no need to recode or recompile if the data set changes!** The TABLE_V class is a subclass of the SAS/AF COMPOSITE class. It is designed to attach to the TABLE_M class and respond to the current settings of its attributes. Other attributes are reserved solely for the TABLE_V object, such as color or size. One of the components of the TABLE_V object is an instance of the SAS/AF TABLE_E class. The same viewer used to view the main data set in a list view is used to view the data in the TABLE object. Users will experience the same customizable table view at both tiers of the data hierarchy, and, as a result, the application benefits from object reusability.

Finally, there are data objects that are not fields in the main data set or other data sets in the hierarchical structure. These data objects are handled in roughly the same way as the TABLE objects are handled. The design of these data objects reflects the MVC concept. It is not necessary to discuss their design in detail, but it is important to note that object-oriented design and the MVC concept will allow for the future design of and incorporation of currently unexpected data objects.

Once the hierarchical data structure is properly modeled, the DEFECTS 2.0 implementation has to present the viewers (FIELDS and TABLES) to the user. As previously stated, each viewer class is designed to be flexible and customizable in its presentation based on its current attributes. It is not enough, however, to have each individual viewer customizable yet fixed in location on the application frame. The viewers must be customizable AND their arrangement on the frame must be customizable. Figure 5 displays the form view with FIELD and TABLE viewers. These viewers are on top of a Work Area widget. As described above, the Work Area widget enables our FIELD and TABLE viewers to act as a dynamic form view, allowing users to move, resize, add, and remove data viewers. **This form view of the data is totally customizable!** Users can view as few or as many of the available fields and tables as they like, in whatever colors and fonts they like, in whatever size they like. Suddenly, the application does not have the impossible task of designing a screen to meet all users' preferences. Each user has the ability to design a view of the data that meets his/her needs. Thus, the application programmer can concentrate on supporting functionality, instead of defending a particular screen design. An added benefit of using the Work Area is that it is not directly associated with one data set modeler. In fact, this design has the potential of displaying fields from more than one data set as if the fields were in the same data set. The user sees FIELDS on the form, but has no idea if the fields come from the same data set. The user sees one form, but could be viewing an interactive join of field data from two, three, or more DATA_M data set modelers. The utility of interactive, flexible, and invisible joins is tremendous. This type of design results in a application that accommodates changes in data and user preferences.

Now that we have encapsulated the relationships in the various object classes and sufficiently sheltered the user from the data structure, we must inform the user. It is important for the casual, infrequent user to not be burdened by having to know about the data structure, but there is a group of users that must have this information. Certain users need to run reports, and they need in-depth understanding of the data structure to write programs or develop tools to access the data. It is probably true that a data management application is only as good as the information that you can get out of it. In that vein, reporting is an important facet to an application such as DEFECTS 2.0. While we can provide users with a built-in report generator, there will always be a set of users that want to generate reports on their own—using data steps, Procs, or other SAS System tools. For these advanced users, and those who would be, the application must attempt to communicate data set names, variable names, formats, relationships, and so on. The question is: "Where should such communication take place?" If it is placed on the surface in the application, then information will be distracting and get in the way of the average user. If it is hidden in the application's help screens, then it is exactly that—hidden. Information such as data set names, variable names, formats, an so on should be placed where a user can easily access it. We chose to display this information in the object Attributes Window.

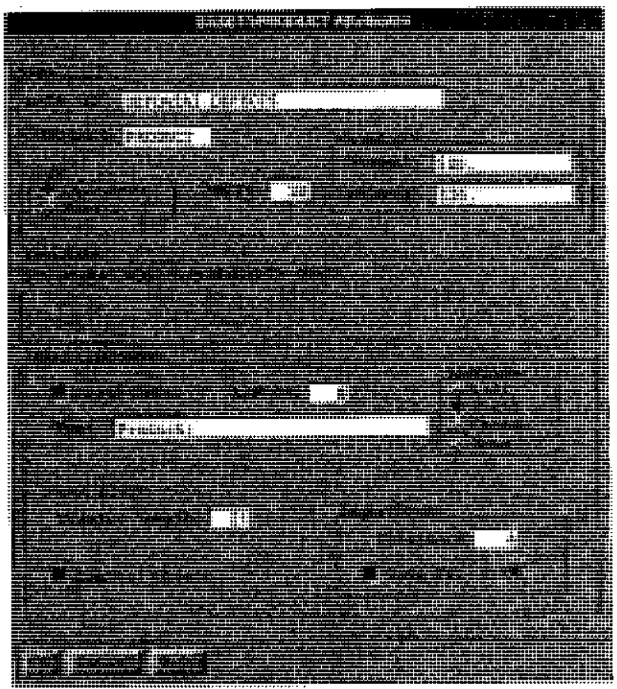


Figure 9. Attributes Window

Thus, the information is closely associated with the data it describes and just a mouse click away. At this point, the challenge is to provide specific information, yet retain the data-independent implementation. We accomplished this by having the FIELD or TABLE object dynamically fill the Attributes Window each time the user selects it. The code that enables the functionality of the Attributes Window only knows what attributes are possible and loads their values from the widget selected. The user views the information in the Attributes Window and the application remains data-independent. As an aside, it is interesting to note that the Attributes Window can be used when constructing the application. It can be coded to receive and save initial attribute values when the application programmer defines the FIELD or TABLE classes for the first time. Then the Attributes Window can change its functionality at run time to display the current attributes and protect certain ones so the user cannot change them. It is quite remarkable that the window designed for displaying an object's attributes can also be used to set its initial attributes.

## CONCLUSION

Finally, there is the need to prepare for the inevitable changes that will be required of the application. This need is at the core of the entire design for DEFECTS 2.0 and the basis for this paper. Remember the scenario described in the opening paragraph of this paper: the application needs new fields, other fields need to be modified, and the arrangement of some displayed information needs to be modified or reformatted. It is a fact of application programming that changes will need to be made, new functionality will need to be added, and many of these modifications will be unforeseen no matter how hard you try to anticipate them. There are no real obvious post-implementation responses to these needs. Rather, the application should be designed, at the out set, to meet these needs. Data-independent widgets and objects implemented in a data-independent design is effective preparation for change. Incorporation of an OOP MVC design into applications such as DEFECTS 2.0 should keep the Maalox where it belongs—on the store shelf, rather than on an application programmer's desk.

## REFERENCES

Booch, Grady. (1991), *Object-Oriented Design with Applications*, Redwood City, CA: Benjamin/Cummings.

Coad, Peter and Yourdon, Edward (1991), *Object-Oriented Design*, Englewood Cliffs, NJ: Prentice-Hall.

Martin, James and Odell, James (1991), *Object-Oriented Analysis*, Englewood Cliffs, NJ: Prentice-Hall.

SAS Institute Inc. (1993), *SAS/AF Software: FRAME Entry Usage and Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1994), *SAS Screen Control Language: Reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1995), *Object-Oriented Programming using SAS/AF Software Course Notes*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1995), *SAS/AF Software: FRAME Class Dictionary, Version 6, First Edition*, Preliminary Documentation, Cary, NC: SAS Institute Inc.

Semaphore (1993), *Object-Oriented Analysis and Design Course Notes*, North Andover, MA: Semaphore.

Shlaer, Sally and Mellor, Stephen J. (1988), *Object-Oriented Systems Analysis: Modeling the World in Data*, Englewood Cliffs, NJ: Prentice Hall.

John Leveille, SAS Institute Inc., 100 SAS Campus Drive, Cary, NC 27513, 677-8000, ext. 7025, e-mail sasjpl@unx.sas.com

Don Williams, SAS Institute, Inc., 100 SAS Campus Drive, Cary, NC 27513, 677-8000, ext. 7037, e-mail sasdsw@unx.sas.com