Refactoring Design Patterns into a SAS Application

Mark Tabladillo, Ph.D., Atlanta, GA

ABSTRACT

This presentation will describe the object-oriented substructure developed to manage, validate, and process survey data. Experience with classes or objects or design patterns is not necessary for this talk, but it will help to know how to code a class and instantiate an object with SCL.

INTRODUCTION

To assist states and countries in developing and maintaining their comprehensive tobacco prevention and control programs, the Centers for Disease Control (CDC) developed the Youth Tobacco Surveillance System (YTSS). The YTSS includes two independent surveys, one for countries and one for American states. A SAS/AF® application was developed to manage and process these surveys. During a four year period, over 600,000 surveys have been processed for 35 states and 100 international sites (from 60 countries).

This presentation will describe the object-oriented substructure developed to manage, validate, and process survey data. Over a three year period, the application grew from zero to four to now thirty classes (and it will likely have more with future refactoring). This presentation will describe the rationale for class construction, illustrate the benefits of design pattern structure, and generalize the lessons and applications to any object-oriented application development (even one which does not use SAS).

THE CONTEXT

In 1998, the Office on Smoking and Health (OSH) first administered the YTSS in three states. This initial group participated in the Youth Tobacco Survey (YTS), built on identical sampling methodology to the Youth Risk Behavior Surveillance System (YRBSS). In 1999, OSH also launched the Global Youth Tobacco Survey (GYTS), sponsored by the World Health Organization (WHO). Subsequently, the Global School Personnel Surveys (GSPS) was added.

Three U.S. states participated in 1998, 10 states participated in 1999, and about 30 states participated in the 2000 school year. OSH provided customized survey support and analysis for each location (state or country). Each location administers a different survey, requiring creating a customized two-stage cluster sampling design, data collection strategy, and analysis. The survey questions are typically customized, with many states and countries choosing to add additional questions to the standard core questionnaire or change the order of questions. Additionally, the states or countries are typically divided into regions, requiring a separate set of reports for each region, and then combined data analysis for the entire state. Many states choose to survey both middle school and high school, so a state with 5 regions would require 5 reports for middle and high school (each), and total summary report, for a total of 12 processes.

WHY ADD MORE CODE?

Most legacy SAS programmers have done fine for many years without explicitly declaring classes. SAS/AF specifically has allowed class (or object-oriented) development, though creating classes is not required for a SAS/AF application.

Classically, a base SAS software program (even one with macro language) could be a very long single block of code. Sometimes that code could be very long, though an organized programmer can easily apply many techniques to make the base SAS code run with as few iterations and variables as possible. Performance tuning and simplification can and should come from a thorough understanding of the source, intermediate, and output datasets. Macro language can be applied to reduce redundancy and interact with the operating system more effectively. In most cases, improving the efficiency of base SAS code involves reducing the number of lines of code.

Creating classes, by contrast, typically increases the amount of code, and increases the locations where code is located (see SAS Institute, 2001). Each time a class variable is declared, it's not enough to use a single DCL (or DECLARE) statement, the developer has to determine the variable scope (such as public or private) and optionally has many other options available (such as editable, description, and initialValue).

What do these options really matter when the whole exercise requires adding more code? At a deep gut level, a SAS programmer who has been trained to reduce code for years may therefore have resistance to adding more code. A SAS/AF frame need not have more code than the single SCL entry related to that frame, making it one very long piece of code.

Also, the historical roots of SAS were initially to reduce the amount of required programming to run statistical routines on large computers. Overall, adding more code feels like the wrong choice to make.

THE DESIGN PATTERN RATIONALE

The term *design pattern* has been used in many object-oriented texts (see Fowler 1999, Gamma et. al., 1995, Page-Jones, 2000, SAS Institute, 2001, Shalloway and Trott, 2002). The term comes from architect Christopher Alexander, who defined the term *pattern* to mean a solution to a problem in context. Thus, given a certain set of conditions (context) the solution would be the same.

For software, applying design patterns requires asking whether there are certain types of problems which occurred with regular frequency, which could be solved in the same way routinely. Perhaps the most cited text is *Design Patterns* (Gamma et. al., 1995), which contains a list of 23 distinct design patterns. Because of this book's popularity, using its terminology has come to be an easy way to learn and apply design patterns. However, the book (even by its own admission) is not the exhaustive or final word on all possible types of design patterns and other authors have introduced modified or entirely new patterns (see Fowler 1999, Page-Jones, 2000, Shalloway and Trott, 2002).

For example, one design pattern used throughout the SAS

literature is the model/view paradigm or the Observer pattern (Gamma et. al., 1995, page 293). In this pattern, the idea is that a single dataset or table of results could be variously viewed in a number of different ways, perhaps as a table of numbers, as a pie chart, and as a bar graph. The three viewers (two graphical, and one table) are linked to a single dataset (the model), and when the model changes, the viewers change too. This pattern could also be implemented in Microsoft Excel, where a single range of numbers can be linked to multiple graphs on the same worksheet. The Observer pattern solves a recurring type of problem, namely how to allow one source object to push content to linked observers.

Another example is the SAS/AF frame itself. The frame is an example of the Mediator pattern (Gamma, et. al., 1995) because it allows communication between two or more graphical objects on the frame. The frame mediates communication among the objects, which themselves remain loosely coupled together. Thus, instead of having SCL code for each and every component, the single frame's SCL code contains (or mediates) communication among the components, which Gamma et. al. refer to as *colleague objects*. For those who have made a SAS/AF frame know how these colleague objects will route requests directly to the mediator (the frame), instead of to each other.

Yet, the frame is not strictly limited to the Mediator pattern only. Inherently, SCL code is not placed inside the class structure, and therefore both the standard frame and SCL programs default to the Singleton design pattern (Gamma, et. al., 1995), which limits the number of instantiations to one. These two examples illustrate several important points:

- Any specific object (like a frame) can and often does rely on several design patterns simultaneously
- Because not everything in SAS requires formal class instantiation does not mean that design patterns are not present
- Discovering design patterns requires continuously thinking about how the software's visual and nonvisual objects act and communicate together, even if those relationships are not explicitly coded.

HOW TO START WITH DESIGN PATTERNS

Shalloway and Trott (2002, page 71) propose that learning design patterns while learning object-oriented programming helps improve understanding of concepts and design. This paper and the author's experience support that premise, and while design patterns are sometimes hard to initially grasp, their rich complexity helps tackle large unstructured programming challenges.

Yet, as a novice class developer, it's hard and impractical to start with the printed design pattern summary on the inside flaps of *Design Patterns* (Gamma, et. al., 1995), and immediately go to some type of class structure and actual code.

Shalloway and Trott (2002) specifically help in providing extensive advice in starting to using design patterns. One basic message is to generally think about patterns as being overlapping constructs which may be simultaneously applied, as is the case with the standard SAS/AF frame. One specific class or object may draw on concepts from several designs simultaneously.

A second message they promote is continuously improving the internal structure of the class structure, or refactoring. Fowler (1999) defines refactoring as follows:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written. (Fowler, 1999, page xvi).

In the case of developing the SAS/AF application for the Office on Smoking and Health (OSH), there were no classes to start with. Granted, there was an initial SAS/AF frame, which has its own inherent design patterns. However, more important than categorizing the inherent object-orientation of SAS/AF, the refactoring process was one of discovering the design patterns important to what the application was supposed to do. For this paper, *refactoring* refers to the process of discovering design patterns within the inherent application design, and applying the patterns through overt class creation.

It was over a period of several years that several classes became inherently important, and the task of creating and recreating smaller subclasses provided great efficiency for making modifications, as well as reduced the possibility of introducing bugs into the code when making changes.

WHEN TO FORGET DESIGN PATTERNS

This paper will support the use of class structure in SAS for specifically two cases.

First, class structure is worth the effort for a long-term or complex application which cannot be coded quickly. Experienced SAS programmers will know that many types of requests can simply be coded with a few lines, and these cases would not be great for either creating classes or necessarily discovering design patterns. For example, the base SAS procedures automatically encapsulate (or hide) implementation. Few programmers would probably know how PROC FREQ actually accesses a dataset and makes tables. Inherent encapsulation makes it unnecessary to create a class structure on top of another class structure.

The answer of how large is large enough to justify class creation is a function of developer's experience. A more experienced developer will naturally see class structure at a smaller level, and may have a lower threshold of when to code classes. However, a less experienced developer may not code any class constructs at all, and only for a very large project would consider overcoming the mental challenge of applying design patterns and refactoring. Again, even in a complex SAS/AF application, there is no requirement to create customized visual components or nonvisual classes.

Second, class structure has a benefit when a previous project already exists which has reusable code. In this case, the SAS/AF application developed had no previous reusable SAS code, as has been documented in Tabladillo (2002). However, this project did lend itself to calling the Microsoft Windows API,

and therefore calling that interface necessarily presents class structure. More details about the Windows API are in Tabladillo (2002).

WHAT IS A CLASS?

Many authors have made attempts to create metaphors for a software class. SAS software specifically was historically written without the need to expressly define a class, and therefore the SAS community varies on its understanding of classes. Java, by contrast, relies on class definitions as its bread and butter, and therefore the discussion seems relatively moot to those developers.

The SAS literature describes a class as a set of attributes (data or variables) and the operations (methods) you can perform on it (SAS Institute, 2000c). The set of data is implemented as a combination of character and numeric variables and SCL lists. There are class definitions too, but behind the scenes, class definitions are implemented as SCL lists too. Methods (or functions or operations or procedures) are well known to SAS programmers, and one example from data step programming is the SUM function, which calculates the sum of numbers. Several inputs can be used for the function, and one return value results. The standard data step is a complex combination of functions, with potentially multiple inputs and outputs.

Shalloway and Trott (2002) have a useful metaphor for understanding classes, and they introduce the scenario of having a school (let's consider it a university) with students. If this metaphorical university were to have no class structure, then the school would have to remember everything about the students (the data) and what they were supposed to do (the operations). When it came time for students to go to classes, the university would have to post a master list on the wall of each student's schedule. Again, because these students have no memory, that schedule would have to be posted throughout the campus, and perhaps the school administrators would physically have to usher students from class to class.

However, these metaphorical students do have the capacity to remember and the ability to operate. Ideally, the university would want its students to remember their own identity and develop their own list of actions and operations to perform, and move from class to class on their own given a class schedule and map. All the students would have the same map, but different students have different schedules. The independent ability of students to think and operate on their own is the same type of discovery when learning that class structure is available for software development. The university delegates responsibility to remember information and complete tasks, therefore lowering the required level of university resources.

The college example is additionally appropriate since it's supposed to be the role of education to increase, enhance and improve not only the students' memory but also their range and depth of functional capacity. All colleges will teach how to think better and some will also provide training for the body through sports. Education is a metaphor for refactoring, and it becomes apparent that sometimes it does benefit to study a software design and become more educated on the functionality. On the other hand, there is also the idea of TMI (too much information), so in a practical sense it's not always worth knowing something even if it's knowable.

DEFINING THE CLASSES

Class metaphors help understand what classes do, but they do not directly provide insight into discovering how a real application would work. The education metaphor is interesting, but its direct application would perhaps be most appropriate for an application that developed artificial intelligence. The application developed for CDC has a smaller yet still complex objective, namely the processing and statistical reporting of tobacco surveys. While the university metaphor may not apply, other design patterns do provide a way to conceptualize the application.

For this application, there was no inherent need to create customized visual components. This type of thing could be done by making a customized resource entry, instead of using the standard resources from sashelp.fsp.afcomponents.resource (version 8 components) or sashelp.fsp.build.resoucre (version 6 or legacy objects). That option is mentioned here for general solutions, and there is good instruction in SAS Institute (2000b).

The next few sections will describe the historical process used to go develop two version 6 SAS/AF applications with no defined classes to the current single SAS/AF application with 30 classes.

STARTING WITH SAS 6.12

Tabladillo (2002) documents how the application got started, and in brief, the choice was made to convert a collection of six base SAS programs into a SAS/AF application. For inspiration, there was a similar SAS/AF application used for another surveillance system, and a second application which had been started for the YTSS, but not completed.

In 1999, only SAS 6.12 was available, and though that version did support class structure in Windows, classes were ignored in the early development, both because the application's intended functionality requirements were not well understood, and because this developer had zero experience in making classes. Neither of the model SAS/AF applications had classes, and in the end, much of the functionality ended up being different (though similar).

Even though it was not understood at the time, putting a SAS/AF frame in front of six base SAS programs immediately meant applying the Mediator design pattern for visual objects. Additionally, Tabladillo (2002) documents the process of making a series of datasets to control the creation of both SCL and base SAS code, which is an application of what Shalloway and Trott (2002) call the Analysis Matrix, where standardized and customized information is encapsulated inside a SAS dataset. Also, since existing base SAS code was being put behind a frame, it could have been considered an application of the Facade pattern (Gamma, et. al., 1995). Since six base SAS programs were eventually collapsed into five distinct processes, that could have been seen as an application of the Strategy pattern (Gamma, et. al., 1995). The point is again, that even though SAS does not require the explicit class declaration (as Java does), the concepts are already inherent in the most basic SAS/AF application. It would only be after many months of applications that these patterns would become more evident, and therefore point the way to simplify and streamline the application's complexity.

When rewriting or modifying existing code, a project of even modest size may likely have some inherent design patterns hidden in the application's function. Refactoring is the way to explicitly code these discoveries into a defined class structure.

Initially, though, the choice was made to develop two separate SAS/AF applications in version 6.12. The first application would be for the Youth Tobacco Survey (YTS) 1999 and the other for the Global Youth Tobacco Survey (GYTS) 1999. Even in retrospect, this decision was good, because it was not known at the time whether or not the two surveys would proceed under identical or similar processing requirements. Not only were the software applications new, but also the surveys were new too, with YTS debuting in 1998 and the GYTS starting in 1999. The OSH team, while inspired greatly by a similar program called the Youth Risk Behavior Surveillance System (YRBSS), had created unique statistical code and algorithms. Additionally, since the OSH team had the benefit of the other team's experience, there was much insight into the types of problems to expect and therefore many data cross-checking features and exception reports were added early in the process. Adding this type of functionality was the main development task during the application's early stages, and with so much functionality being added, the time was not right for class structure development.

Additionally, it was already known that SAS would soon be doing a major release to version 8, and while the product did not yet come, the development proceeded with the assumption that the two applications would need to undergo major conversion.

CONVERTING TO SAS 8

SAS version 8 for Microsoft Windows introduced many new elements, including some new features for SAS/AF. Specifically, this major release supported dot notation and longer variable names and longer possible sizes for character variables, as documented in SAS Institute (2000c).

For converting the frame, there would now be two categories of visual components, the list from version 6 and the newer version 8 counterparts. In some cases, like the data table or the tab object, there were only legacy or version 6 components available, so those components were left at that version. In other cases, version 8 had a newer component, and the main advantage included having a standardized properties interface, similar to what Visual Basic developers have. By contrast, version 6 had often unique and customized interfaces to set properties for visual objects. The newer version 8 properties list provides all the possibilities on one screen, and allows these properties to be sorted by name or category.

For converting the code, the variables now had to be declared using the DCL (or DECLARE) statement. Additionally, some variables benefited from the long variable names. However, most of the variable names were left alone, and are a historical reminder of the application's version 6 origins. When possible, the dot notation was used to replace the version 6 equivalent.

INTRODUCING FOUR CLASSES

When the application was being converted to version 8, the group still continued to use the version 6 programs simultaneously, and they were used as the benchmark standard to judge the new application. With two solid benchmarks in place, it was possible then to not only convert versions, but also roll the two separate applications into one. The new single application could be developed and compared to the other two production applications, which were both in stable shape by the time SAS version 8 software was installed.

Around the same time, the first four classes were introduced to the project, and these four classes were named Output, Win32, Region, and SurveyAnalyzer.

The Output class had four functions to change the orientation and print size of the output. The four choices supported became the names of the functions of that class: port8, port10, land8 and land10. The five processes which ran required changing the printer, and by having this simple functionality in a class, it could be called from any SCL routine or class. This early class would first set the ORIENTATION and SYSPRINTFONT using a SUBMIT routine, and then set the linesize and pagesize.

Since then, the Output class has been changed to setting the margins instead of the linesize and pagesize because we learned that Windows 2000 has slightly different fonts than Windows 95, and therefore we allow Windows to tell us what the linesize and pagesize should be given a set of specific margins. Also, the Output class has an option for not only orientation and font size but also font. These changes are relatively minor, but because the functionality was encapsulated inside a class, it required only changing that class and not changing all the many times this class is called.

The Win32 class came to be the repository of accessing Microsoft Windows. Especially if you don't like Microsoft Windows, you might choose to do the same thing in your SAS/AF application, and that would keep you from looking at Windows functionality every day. Alternatively, if you are considering using portions of your code on other operating systems, it's a good idea to consider putting all the operating-specific commands inside one class. Returning to an earlier premise of this paper, separating out operating system commands will not likely reduce the amount of overall code, but it does provide advantages including making the application more maintainable (if you should happen to change your specific operating systems version) and reusable (if you should want to go to another platform altogether).

The types of things handled in this early Win32 class include the following:

- Reading a Windows 32 Error Table – a SAS Dataset was created with the standard descriptions for the numeric Windows 32 errors, and this dataset is read into an SCL list
- Detecting the specific Windows version – a call to the SYSSCPL macro system variable.
- Printing the Windows 32 Error to the Log – how the Windows 32 errors are used
- Creating the SASCBTBL text file from a SAS Catalog source program – the SCL preview command reads and writes the file for accessing the Windows 32 API
- Getting the Windows System Directory – Knowing the system directory is important for potentially having to copy and register the single Visual Basic OCX control (see Tabladillo 2002 for more details on the control)
- Creating, copying, or moving files or directories in Windows

- Setting Environmental Variable – This feature is unique to Windows and is required to use SAS callable SUDAAN (which is add-on software used to calculate standard error estimates for complex survey designs)

Again, as Windows changes or as Windows-related needs change, it is only important to affect this one specific class instead of having to rewrite potentially many areas of the code. Newer changes to the Win32 class include comparing versions of DLLs (which is a hard problem to solve), and getting the user and computer name for the log dataset.

The region class is the major application workhorse, and provides the application with a region's identity. As described more fully in Tabladillo (2002), this application is generic to different countries (for the GYTS) or different American states (for the YTS), and within each year, a specific country (for example) might choose to stratify their sample into several regions. Some cases require regions to be as many as 25 but typically the range is from 3 to 7. The application can currently handle up to 999 regions.

For running the five processes, the user needs to choose a specific survey, a specific year, and a specific region to work on. For example, the choice might be YTS 2000, Kansas region one. Having made these choices implies a certain set of frame variables which all have default values, but may have user-specified values for that specific region. For the first version 8 application, the user could set 12 pieces of information on the screen, pick one of the five processes, and hit the command button labeled "RUN". The program would then generate customized SCL and base SAS code from a combination of the information from the screen and information from standardized and customized datasets for that specific region (see Tabladillo 2002 for more details on how these information sources work together).

A single dataset representing the specific survey and year, in this example YTS 2000, stores the information from the screen. How the information gets from the screen to base SAS, however, is handled by classes.

The early Region class simply stored the information from the screen, and provided data integrity checks using the SET CAM option. This feature would run a specific protected method when the variable was set to a new value. The following table contains a five methods called by SET CAM.

```
Sex:protected method sex:char;
      sex = trim(sex);
      if sex = ' ' then sex = 'CR2';
endmethod;

Grade:protected method grade:char;
      grade = trim(grade);
      if grade = ' ' then grade = 'CR3';
endmethod;

Impsel:protected method impsel:num;
      if impsel not in (1,2,3) then impsel =
1;
endmethod;

MinPct:protected method minpct:num;
      if minpct<0 or minpct>100 then minpct
= 60;
endmethod;

MaxClass:protected method maxclass:num;
      if maxclass<1 or maxclass>500 then
maxclass = 20;
endmethod;
```

Again, what is interesting is that this conversion came from version 6 so the variable names are short. However, the main focus is to see that the class would only allow for values in a certain range to be accepted into the program. In the case of checking for blanks, the class advantage is having the ability to set a default value, and in this case, the default for sex is CR2 and the default for grade is CR3.

This early region class also provided a series of directory names which were specific to that region. A specific example is the variable dir_output which is the location to save the output. Within each regional subdirectory, five standardized subdirectories store the region-specific files: 1) "Final" has the final output results, 2) "Input" has the input text files, 3) "Log" stores program logs, 4) "Output" stores program output, and 5) "SASData" stores the SAS survey data and SAS control datasets (and perhaps the control data in another format like Microsoft Excel). Each file is named according to a standardized naming convention, which necessarily requires information from the control datasets, and sometimes the date is included. The region class is the natural location to determine this type of information, because it is specific to that region. The location of subdirectories, then, is a derivative concept which is part of that region's identity, and therefore can and should be part of that class instead of being connected to a frame.

Since then, a number of additions have been made to the Region class, the most important of which has been the setting and clearing of libnames. Previously, libnames were always set or reset from the frame SCL, but since the directory structure is region-specific, setting libnames should also be there too.

The fourth early class was the SurveyAnalyzer class, which was intended to implement the Strategy design pattern. Sometimes it makes sense to keep the identity and the

functionality together, and it would have been possible to put the five processes into functions of the Region class. Gamma et. al. (1995) make the case and illustrate the tradeoffs for using the strategy design pattern.

In retrospect, some of the variables declared in the early SurveyAnalyzer class were later moved to their arguably more proper home inside the Region class. However, encapsulating these variables from the single frame (this application runs on one frame with a tab layout object), allows these variables to be considered as a unit, and being in the SurveyAnalyzer class was better than being with the frame.

The idea behind the Strategy design pattern is to present the same list of variables to the five processes. Not all the variables would necessarily be used, but because all five processes were run from the same tab, all the frame variables were simply taken from the screen, put into the region class (and potentially changed) and then presented to the SurveyAnalyzer class to create customized base SAS code.

Creating this class forces the question of what is absolutely necessary to specify on the screen, and what could be better put behind the scenes. Streamlined application development involves creating an application with the minimum number of parts presented to the user, thereby reducing the amount of training required but also reducing the chances of errors.

The early SurveyAnalyzer class provided some functions which created derived information such as the names of region-specific datasets. This was initially included in this class because it was related to the submission of code. Later, this dataset information was put into the region Class because overall it's more part of the regional identity.

Another early function would store the output and the log to a specific location. Previously, this storage was done by the frame SCL, but it is common function related to all the five processes, and therefore properly is better placed inside the class. Adding functionality before or after something else is considered an application of the Decorator pattern (Gamma et. al., 1995).

Since then, the SurveyAnalyzer class has been divided into the stateAnalyzer class and its subclass called regionAnalyzer. The stateAnalyzer class allows for multiple region analysis, and the regionAnalyzer subclass provides the typical regional level processing. Conceptually, either could have been the parent or child, and that decision is still changeable.

These four early classes represent the majority of what the application would conceptually add. The next few sections will describe the other classes, and illustrate what advantages they brought to the application.

DATASET ATTRIBUTE CLASSES

In terms of numbers of classes, 20 of 30 classes in the current application are related to dataset attributes. The single parent class is called DATASET_ATTR and given a dataset ID (or number), it will determine all the possible dataset characteristics from the SCL ATTRC and ATTRN commands, storing the results in class variables. The two methods will close or delete a dataset.

The subclasses all inherit the parent's functionality using the optional EXTENDS command on the CLASS statement. Each of the 19 subclasses refers to a different type of dataset. For example, the class DATASET_ATTR_LAYOUT refers to the region-specific questionnaire, and DATASET_ATTR_DATA refers to the initial survey data. Inside these subclasses, the variable numbers are determined for the standardized list of expected variable names. Some variables are required to run processes, and other variables are optional (if included, they will trigger certain processes). Thus, these subclasses mostly pass the results from the SCL VARNUM command to class variables, and keep the result.

Putting the layout, for example, in a class structure allows the developer to define two objects for the same type of dataset (a layout) and possibly do something with these two layouts together. One possibility would be to compare content.

One subclass of general use is called DATASET_ATTR_PROCFREQ and was created specifically to read the standard dataset which is output from the base SAS proc freq command. Both the count and the percent are kept as class variables when the class is presented with a dataset ID.

Though there are 19 different types of datasets (the generic type makes 20), future ones could easily be added by subclassing the parent class, and then customizing the subclass to reflect the structure of that new class.

ENUMERATING THE HIERARCHY

The early application had a Region class, but the newer structure enumerates the hierarchy into specific classes, which opens the possibility for the application to better organize the program state. The following table provides a list of the current classes which define the hierarchy, with the first being the parent, and each class below being a successive subclass:

Class	Scope
Hierarchy	Generic hierarchy class, which is used to implement error handling
SurveyYear	Survey type and year (e.g. YTS 2000)
State	Country or American State
Region	Region within country or American state

Each of these classes stores information about identity, and may provide information on directories and files at that level. The functions are specific to that level, and may include setting or clearing libnames (for example).

In general, when the data follows a hierarchical path, it may be beneficial to map that hierarchy into its equivalent class structure.

THE LAST TWO CLASSES

There are 20 dataset attribute classes and four in the hierarchy. The Output and Win32 classes have already been described, and also the regionAnalyzer and stateAnalyzer classes have also been outlined.

The second to the last class is called AnnualCode and is the repository for methods and attribute definitions which are tied to a particular survey and year, and therefore need to be modified each year. Perhaps in the future, this class may be renamed surveyYearAnalyzer (in line with the other Analyzer classes), or its content may be absorbed into the regular surveyYear class.

The last class is a utility class called sentenceParser, and what it does is given a length and a long string, it will chop up that string into smaller pieces, given the assumption that the string is a sentence. Thus, the method will not cut a word in half, but the method remains relatively simple because it does not hyphenate, but simply looks for blanks. The result is delivered in an SCL list. The class is used several times to determine multiple line titles and footnotes. The class structure builds on the Interpreter design pattern (Gamma, et. al., 1995).

APPLYING REFACTORING

Perhaps it seems unfair that what this paper presents is basically the starting, intermediate, and ending points from refactoring. Granted, some future refactoring will likely take place, and it will involve thinking about what the application does, and how to best express that functionality behind the scenes.

Fowler (1999) provides many specific and concrete examples of the types of things which can be done to refactor. However, in actual practice it's important to continually read different design patterns and go over refactoring techniques from time to time so that they become an inherent part of how code is evaluated strictly from human memory.

In other words, it's unlikely that a refactoring sequence could be choreographed, like a dance is choreographed, and therefore the expectation should be to understand the ideas behind refactoring. Likewise, for social dancing, it's more important to understand the basic rhythms and techniques behind a dance rather than memorizing all the possible patterns. Fowler (1999) specifically spends some time on what the basic fundamentals are behind refactoring, and Shalloway and Trott (2002) provide additional instruction in design pattern fundamentals.

Several examples were illustrated in this paper. One type of example is deciding where a specific variable should be defined. The earlier example was one of moving the dataset names from the SurveyAnalyzer class (which is more of an action class) to the Region class (which is more of an identity class). Granted, the difference is entirely subjective, and truly only is defendable given an overall analysis of the entire application.

The major refactoring in this application involved moving variables and functions away from the single frame and into nonvisual classes. The guiding general principle is that the frame provides the work of the Mediator design pattern, and tasks and variables related to that mediation function should be with the frame. Other tasks and functions not directly related to the frame could be moved to a specific class. The classes defined in this application all provide examples of data and functions not related to the input or display of information on a frame.

Because SAS/AF development is based on frame creation, the same idea would apply to all SAS/AF applications in general. More generically, the ideas also apply to web-based applications where the interpreted screen code is best left to a mediation function, and other functions could be called in from code encapsulated in other files. Once a specific frame structure is chosen, then the standard job of increasing performance by reducing lines of code can be applied. Specifically, if a block of code is repeated several times in the frame SCL, adding the class may actually reduce the total lines of code by replacing blocks with single function calls.

Adding classes is a type of expansion, and evaluating and enhancing performance by reducing lines is a type of contraction. Fowler (1999) mentions some cases where the developer may choose, as part of contraction, to collapse classes together.

GENERALIZED LESSONS

1) ***Consider adding class structure even if it means more code***. The different examples illustrated show that conceptually certain attributes and methods have an inherent cohesion (like the Win32 class), and therefore make sense to organize together.
2) ***Recognize that even base SAS has inherent design pattern structure***. From the beginning, the intention of SAS was to hide class instantiation, and that legacy means that sometimes the inherent overlapping object-orientation may not be immediately apparent.
3) ***Continuously learn and apply design patterns***. Even a new class developer can benefit from attempting to know and understand the complicated design pattern textbooks. As time passes, even more helpful texts and examples become available, and that helps reduce the learning curve.
4) ***Expect to continuously apply refactoring to large or complex applications***. There are always ways to reorganize code to make it more maintainable, and therefore reduce the chance of introducing bugs when it is modified.

CONCLUSION

This paper has discussed the build-time process of refactoring design patterns into a SAS application which had no explicit class declarations. Along with a version transfer, the resulting application went through a class growth, during which attributes (variables) and methods (functions or procedures) were encapsulated away from the standard SAS/AF frame into a more individually cohesive class structure which could be modified or generalized. The techniques applied here are most appropriate for formal SAS application development, though as developers gain experience with identifying design patterns, refactoring could be quickly applied also to projects of a more modest scope.

Further information on the dataset structure supporting this application is presented in Tabladillo (2002).

GLOSSARY

Abstract Class – a type of class created to describe an interface for inheritance purposes (SAS/AF has a separate interface statement), and therefore not intended to be

directly instantiated.

Attribute – In SAS/AF, the variables declared inside a class. The attribute types can be character, numeric, or SCL list.

Class – a class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform. Alternatively, a class is a build-time repository for the declared attributes (variables), and the methods (functions or procedures).

Cohesion – a description of how closely the operations inside a method, or methods within a class, are related.

Constructor – the method automatically invoked to initialize new instances. In SAS/AF, the optional constructor is a method with the same name as the class.

Coupling – the degree to which software attributes (variables), methods, classes, or components depend on each other.

Design Pattern – an object-oriented solution to a commonly recurring software problem. The solution is a general arrangement of classes and objects which solve the problem, though the answer is customized for a particular context.

Destructor – an method that is automatically invoked to finalize an object that is about to be deleted. In SAS/AF, the destructor is the _TERM method, which can be overridden to provide specific functionality.

Encapsulation – any kind of hiding, whether hiding variables or hiding methods, inside a class structure.

Inheritance – the process of acquiring the characteristics of another piece of software code. Subclassing involves inheriting both an interface and an implementation.

Interface – the set of all signatures for all methods of a specific class.

Instance – a particular example of a class; an instance is always an object.

Instantiation – the process of creating an instance of a class. In SAS/AF instantiation can be performed with a declaration or with the "_new_" operator.

Method – the combination of a signature and the procedures applied to the input parameters to determine a return value.

Object – a run-time entity which packages both a class structure and a specific state.

Operation – see method.

Overloading – redefining a method based on a new and different signature.

Overriding – redefining a method for a specific signature.

Polymorphism – the ability to substitute objects of matching interface for one another at run-time.

Signature – an method's signature defines its name, parameters, parameter types, and return value and type.

Subclass – a class which inherits the entire attribute and method structure from its parent. In SAS/AF, subclasses are defined by the EXTENDS option on the CLASS statement.

Variable – see attribute.

REFERENCES

Fowler, Martin (1999), *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison Wesley Longman, Inc.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison Wesley Longman, Inc.

Page-Jones, M. (2000), *Fundamentals of Object-Oriented Design in UML*, Reading, MA: Addison Wesley Longman, Inc.

SAS Institute Inc. (2000a), *SAS/AF Software: Application Development I Course Notes*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2000b), *SAS/AF Software: Application Development II Course Notes*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2001), *SAS/AF Software: Component Development Course Notes*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2000c), *SAS/AF Software Procedure Guide, Version 8*, Cary, NC: SAS Institute, Inc.

Shalloway, A., and Trott, J. (2002), *Design Patterns Explained: a New Perspective on Object-Oriented Design*, Boston, MA: Addison-Wesley, Inc.

Tabladillo, M. (2002), "Developing a control methodology for customized data management and processing", *SESUG Proceedings 2002*.

ACKNOWLEDGMENTS

Clifton Loo, Ph.D. provided important editing and feedback. Also, thanks to all the great public health professionals at the Office on Smoking and Health, Center for Chronic Disease.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Mark Tabladillo, Ph.D.
Email: marktab@marktab.com
Web: http://www.marktab.com/