Paper 31-28

Application Refactoring with Design Patterns

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 presentation, but it will help to know how to code a class and instantiate an object with SCL (SAS Component Language).

INTRODUCTION

This presentation will describe the object-oriented substructure developed to manage, validate, and process survey data. Over a four year period, the application grew from zero to four to now thirty-three classes. This presentation will describe the rationale for constructing design patterns, 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® software).

BACKGROUND

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 similar but independent surveys, one for countries and one for American states. A SAS/AF® application was developed to manage and process these surveys. During a five year period, over 1,000,000 surveys have been processed for 35 states and 100 international sites (from 60 countries).

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 Survey (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 customized survey, requiring a unique two-stage cluster sampling design, data collection strategy, and analysis. Many states and countries choose 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. For the YTS, 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 units.

DESIGN PATTERNS INHERENT TO SAS

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). Attributes are implemented as a combination of character and numeric variables and SCL lists; class definitions are implemented as SCL lists. Methods (or functions or operations or procedures) can have several inputs and one explicit return value. Implicit return values (or outputs) can be encapsulated in class attributes or external datasets.

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 was coined by 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.

Because of its ubiquitousness, the term *design patterns* has come to describe any intentional architecture applied through class structure. The SAS System and all software applications which have intentional architecture have inherent design patterns, even if the software does not allow class creation or instantiation.

For any 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 immediate and sustained popularity, its terminology has come to be a recognized way to describe class architecture. However, the book (by its own admission) is not the exhaustive or final word on all possible types of design patterns and other authors have introduced complementary modified or entirely new patterns (see Fowler 1999, Page-Jones, 2000, Shalloway and Trott, 2002).

The SAS System has an intentional architecture, and therefore has inherent design patterns. For example, one design pattern described throughout the SAS documentation 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. For example, three viewers (two graphical and one table) could be 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 dynamically linked to multiple graphs; when the numbers change, so do the linked graphs. The Observer pattern solves a recurring type of problem, namely how to allow one source object to push content to linked observers.

The SAS/AF frame demonstrates another example of inherent design patterns. The frame uses 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 remain loosely coupled together. Instead of having SCL code for every component, the single frame's SCL code contains (or mediates) communication among the components, which Gamma et. al. refer to as *colleague objects*. Colleague objects route requests directly to the mediator (the frame), instead of to each other.

Yet, the frame is not strictly a Mediator only. Because frame SCL code is not placed inside the class structure, the standard frame defaults to the Singleton design pattern (Gamma, et. al., 1995), which limits the number of instantiations to one (one-to-one class-to-object). Thus, someone unfamiliar to SAS might better understand a frame as a Visual Singleton Mediator.

These examples illustrate several important points:

- Because not everything in SAS requires formal class instantiation does not mean that design patterns are not present
- Any specific object (like a frame) can and often does rely on several design patterns simultaneously
- Discovering design patterns requires continuously thinking about how the software's visual and nonvisual

> objects act and communicate together, even if design patterns are not explicitly declared

HOW TO START WITH DESIGN PATTERNS

In the last section, we saw that you don't have to know anything about class or design pattern vocabulary to use the constructs, but the premise of this paper is that using the vocabulary gives you deeper insight into application development as well as makes possible drawing common solutions from other developers.

Shalloway and Trott (2002, page 71) propose that learning design patterns while learning object-oriented programming helps improve understanding of concepts and design. At the beginning, it might be hard and impractical to start with a classic conceptual text like *Design Patterns* (Gamma, et. al., 1995), and immediately develop a workable class structure.

Shalloway and Trott (2002) specifically help in providing extensive advice in starting to use design patterns, and therefore their text serves as a better starting point. One message they teach 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 (Singleton Mediator). 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).

While developing this application, there were no classes to start with. Granted, there was an initial SAS/AF frame and the SAS environment, each of which has its own inherent design patterns. However, more important than categorizing all the inherent object-orientation of SAS/AF, the refactoring process was one of discovering the design patterns central to what the application was supposed to know (data) and do (functions).

In this paper, *refactoring* refers to the process of discovering design patterns within the inherent application design, and applying new patterns through overt class changes or creation. In other words, refactoring requires changing class structure. It was over a period of several years that several classes became inherently important. The task of creating and recreating classes provided great efficiency for making application modifications, as well as reduced the possibility of introducing new bugs when making changes.

WHEN TO CREATE DESIGN PATTERNS

Because design patterns are inherent to the SAS System, there is no question about whether or not to use them. Rather, the question is when to intentionally integrate design patterns into an application.

First, design patterns have an obvious benefit when a previous project has reusable code. For example, this application did lend itself to calling the Microsoft Windows API, and therefore calling that interface necessarily presents class structure. In this application, Microsoft Windows functions were logically encapsulated in one class. Also, the application encapsulated six base SAS programs inside a class.

Second, design patterns are useful for organizing large blocks of code. In this application, class development encapsulated frame SCL code. However, because SAS is typically efficient through its base SAS language, many programs do not lend themselves toward efficiencies in creating classes or necessarily discovering design patterns. By design, base SAS procedures already intentionally encapsulate (or hide) implementation. As software development increasingly relies on standardized components, there may be little need to go beyond the Façade pattern.

Third, design patterns are worth the effort for a long-term or complex application which cannot be coded quickly. The question of how long or how complex are both answered by the developer's experience. A developer more knowledgeable about design patterns will naturally see class structure at a smaller level, and may therefore have a lower threshold of when to code classes. However, a less experienced developer may not code any class constructs at all at the same threshold. In any SAS/AF application, there is no inherent requirement to declare customized visual components or nonvisual classes.

OVERALL DEVELOPMENT STRATEGY

Creating customized visual classes is an important activity when a customized and changing visual presentation is central to the application's functionality. For this application, there was no inherent need to create customized visual components (possible by making a customized resource entry).

By contrast, survey processing and reporting is less of an interactive user experience and more of a processing activity. Therefore, this application was built centrally on non-visual classes. This choice does not mean that the user interface is unimportant, but that the application was less about user interaction and more about data monitoring. An application based on user input and interaction would, by contrast, be more concerned about applying design patterns to visual classes.

The next few sections will describe the historical process used to develop two version 6 SAS/AF applications with no defined classes to the current single SAS/AF application with 33 classes.

INITIAL DEVELOPMENT IN SAS 6.12

Tabladillo (2003) 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 frame-based application. In 1999, only SAS 6.12 was available in production, and though that version did support class structure in Windows, the early development did not declare classes both because the application's intended functionality requirements were not well understood. Neither of the two 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 (2003) documents the initial process of encapsulating data for use in SCL and base SAS code, with a strategic extension of Shalloway and Trott's (2002) Analysis Matrix. 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). Six base SAS programs were eventually collapsed into five distinct processes (a *process* is defined as SCL code with submitted base SAS code), an action that could have been considered an application of the Strategy pattern (Gamma, et. al., 1995). Even though SAS does not require the explicit class declaration, the concepts are already inherent in the most basic SAS/AF application.

When rewriting or modifying existing code, a project of even modest size will 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 surveillance 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 and subtracted, 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.

APPLICATION CONVERGENCE IN SAS 8.0

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, some 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, and consistent dot notation language. By contrast, version 6 had often unique and customized interfaces to set properties for visual objects, and in most cases, unique ways to access the object in SCL. The newer version 8 properties list provides all the possibilities on one screen.

For converting the code, all variables now had to be declared using the DCL (or DECLARE) statement. Additionally, some variables benefited from the longer names. However, many 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 command.

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.

At this 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. 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; additionally, the class captured return errors.

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 (even though they are all True Type with the same name and size), 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.

This class resembles the Decorator design pattern since it couples error codes with print option setting, though it is not strictly a Decorator because the original wrapped functions are not classes themselves. It would be great if all SAS functions and procedures were inheritable classes, but then this application would not be a Decorator, which requires runtime (not inherited) coupling.

The Win32 class came to be the repository of encapsulating certain Microsoft Windows functions. 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. 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 system macro 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
- Working with 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 2003)
- Creating, copying, or moving files or directories in Windows – though these functions are mostly in SCL, the advantage of using native Windows includes a larger range of specific errors in case of failure
- 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 (perhaps a future version would be "Win64"). 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. This class is a Façade design pattern, since it exposes only certain interfaces from Windows altogether, and allows the application to be less disturbed when the operating system changes. The implementation is almost an Object Adapter, except that the Windows instantiation is implicitly called, not explicitly coded.

The region class is the major application workhorse, and provides the application with a region's identity. As described more fully in Tabladillo (2003), 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.

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 2002, 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 process 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 2003 for more details on how these information sources work together).

A single dataset representing the specific survey and year, in this example YTS 2002, 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 sample 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;
```

The class allows 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. Some region-specific cross-validation checks are also done.

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 identifying region-specific libnames should also be there too.

The fourth early class was the **SurveyAnalyzer** class, which, considered with the region class, is similar to the Bridge pattern (the different analyses are in distinct SCL files instead of subclasses). 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.

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 SCL.

The idea behind the Bridge 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 customized regional identity rather than user-chosen functionality.

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.

These four early classes represent the majority of what the application would conceptually add. The next few sections will describe other non-visual classes.

DATASET ATTRIBUTE CLASSES

In terms of numbers of classes, 22 of 33 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 object variables. The five methods are 1) create a new dataset, 2) open an existing dataset, 3) update an existing dataset, 4) close a dataset, and 5) delete a dataset.

The subclasses 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 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). These subclasses retain the results from the VARNUM command in object variables.

For example, putting the layout in its own class allows the developer to define two objects for the same type of dataset (a layout) and possibly do something with these two layouts together. For example, the application compares the standardized master with the region-specific (typically customized) layout for inconsistencies during customization.

One subclass of general use is called DATASET_ATTR_PROCFREQ and was created specifically to read the standard output dataset generated by proc freq. Both the count and the percent are kept as object variables when the object is presented with a valid dataset ID.

Though there are 21 different declared types of datasets (adding the generic type makes 22 classes), future subclasses could easily be added by subclassing the parent class, and then customizing the subclass to reflect the intended structure of the dataset. The entire structure most closely matches the Strategy design pattern, with the common overridden method being the constructor.

REFACTORING INTO HIERARCHIES

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 Name	Scope
Hierarchy	Generic hierarchy class, which is used to implement error handling
SurveyYear	Survey type and year (e.g. YTS 2002)
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 each level, and may include setting or clearing libnames (for example). Both the state and region objects contain a Memento design pattern which allows the object to know the initial state (or initial condition) during multiple region or multiple state analysis.

Also, the SurveyAnalyzer class has hierarchically expanded into three classes, in inheritance order, **surveyYearAnalyzer**, **stateAnalyzer**, and **regionAnalyzer**. The surveyYearAnalyzer analyzes an entire survey (multiple states), the stateAnalyzer class analyzes and entire state (multiple regions), and the regionAnalyzer class analyzes a specific region. Generic variables and functions (of use to all classes) were pushed up the hierarchy, while subclasses added variables and overridden functions specific to that level and below.

THE LAST TWO UTILITY CLASSES

There are 22 dataset attribute classes and 4 in the hierarchy. The 2 Output and Win32 classes have already been described, and the 3 Analyzer 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. This class is modified with each new survey addition. Perhaps in the future, this class may be absorbed into the regular surveyYear and/or surveyYearAnalyzer class. This class is a primitive precursor to the Strategy design pattern since all the choices are in the same object.

The last class is a utility class called **sentenceParser**, which, given a numeric length and a long string, will chop up that string into lines the size of the given length (or smaller). The method will not cut a word in half, but simply assumes that blanks represent delimiters, and preserves the results in an SCL list. The class is used several times to determine multiple line titles and footnotes. The class concept comes from the Interpreter design pattern, though is a primitive form because the results (lines of output) are not individually stored in unique subclasses. The limited but useful functionality and stateless behavior (despite results storage) make it also a Concrete Flyweight.

APPLYING REFACTORING

Some future refactoring will likely take place, and it will involve thinking about what the application knows and 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 study new and old 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.

Several refactoring 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/function class) to the Region class (which is more of an identity/data class). In some cases, 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 frame (visual singleton mediator) 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 directly 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) ***Recognize that all intentionally structured languages have 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.

2) ***Consider adding design patterns 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.
3) ***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.
4) ***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.

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 encapsulated data structure supporting this application is presented in Tabladillo (2003).

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. (2003), "The One-Time Methodology: Encapsulating Application Data", *SUGI Proceedings 2003*.

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.

TRADEMARK CITATION

CONTACT INFORMATION

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

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