Paper 186-2009

Better, Faster, and Cheaper SAS® Software Lifecycle

Edmond Cheng, Bureau of Labor Statistics, Washington, DC

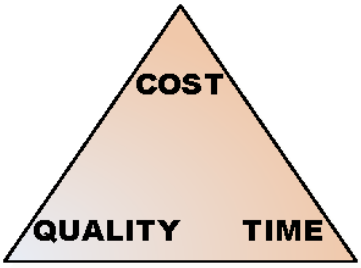
ABSTRACT

In designing software applications, the enduring process faces realistic business challenges to overcome the restricted time, limited resources, and quality constraint in engineering paradigm. Overemphasizing the three engineering limitations upsets the balance of producing highest quality, shortest time, and lowest budget cost during product development lifecycle at the highest efficient mode.

By considering the following four design practices during your SAS® software lifecycle, it is possible to define an approach of building SAS-based software thru maximizing limited constraints. The paper presents 1) functional techniques in expanding single-use program's flexibility and maintainability, 2) practical guidelines to promote code library and reusability, 3) strategies and advantages of modular process design, and most importantly 4) connection of individual program components into user-driven application. In the end, it makes it possible for project managers and developers to promote a better, faster, and cheaper SAS programming software life cycle.

INTRODUCTION

In industries such as engineering, manufacturing, computer science, and others, where projects involves research and development, managers often face unavoidable constraints of budget, schedule, and quality. Limited resources affect the cost, time, and quality, and present dilemmas for managers in decision making. Given that constraints are generally predetermined and fixed, managers utilize the best efficiencies of resources in interest of business. Ineffectual allocation or scarifying resources from one dimension of the paradigm to satisfy another dimension, often lead to unpredictable consequences or undesired business outcomes.



The purpose of this paper is to consider several practical software engineering methods and their implication to the SAS software lifecycle.

WHAT IS THE SAS SOFTWARE LIFECYCLE

If one asks what a software lifecycle is, the answer is the period of time that begins when a software product initiates and ends when the software is no longer in use. At the problem identification phase, the need of SAS software is identified. The progress proceeds to planning phase with defining requirements, setting out specifications, and the advancing to development. The phase continues along the way with testing, user acceptance, deployment, and maintenance. Finally, the product lifecycle ends when the software becomes obsolete and finishes the final shelf life.

To define the engineering paradigms of limited resources needed to have a better, faster, and cheaper SAS software lifecycle, let's begin with some particular design practices applicable to SAS software development.

PROCEDURAL PROGRAMMING

Software engineers and developers practice procedural programming in almost assurance, while most occasion programmers take on the approach to certain extend, even if not aware in some cases. Procedural programming involves designing and writing concise programs in a functional orientation, with limited function utility in scope. Yet, these programs can run independently without inputs from other programs within the same group.

For example, a block of frequent use PROC SQL code with additional flexibility in parameters and macrotization can be written as a procedural program. Repetitive file input/output, database conversion, query reporting, file transfer, and many other candidates can be structured in a more reusable manner.

Table 1: Suggested practices applicable in SAS programming

Practice	Description and usage
functional	program should be concise to perform specific function – write program defines by function, rather than one to do it all – approach with a top-down design by addressing overall requirements at the highest level, then fashion details to meet purpose at the bottom level – knowing the desire state of outcome – consider between data flow versus process flow – utilization thru standalone, INCLUDE, or macro programs
simplicity	keep the basis brief and simple – write the least number of code possible for the job – consider procedure (PROC) steps over data (DATA) steps – remove test code – remove unnecessary variables in dataset
flexibility	easy to make modifications – keep design adaptable and expandable – use for parameterization and avoid hard coding – code defensively to anticipate changes – limit dependency between procedure programs
maintainability	make it effortless for anyone to maintain – easy for corrections and quality assurance – consider use of control and format datasets for storing lookup parameters – follow meaningful variable name, database name, macro definition
structured programming	logically break the program into smaller parts – logic composition with consistency, separation of concern – concatenation, selection, and repetition – avoid jumping (GOTO) between or within programs
documentation	add readable language to improve understandability of programming code – standardize a header section to identify the program purpose – include comments, for yourself and others, walkthrough the program – write descriptive title, column headers, variable labels for tables and graphs

There are enormous benefits of practicing procedural programming. Writing SAS programs with a functional orientation makes it easy to build accessible programs performing specific functions, which can call upon and promote reusability in future. Procedural programming makes the job of maintaining or modification often a lot simpler. Less time is spent making changes to the programs, thus improving project quality. As result, more time and resources can be allocated building additional procedural programs to perform different tasks.

LIBRARY COLLECTION

A well-centralized location for program storage is like shelving a library collection of readily available block of codes at one's disposal. A library collection facilitates the storage and retrieval of functional codes, as well as coordination of database, documentation, and documents. Together with configuration management implementation, the task to maintain file version control is less complicated. Project managers can track project progress throughout the each lifecycle phases.

Assembling procedural programs into a library collection, promotes a 'bottom-up' modular process design, which will be described in the next section. In SAS, the functional standalone programs can be stored in a designated location, which provides a central area for repetitively retrieving code. Macros can make use of the compiled macros library and the AUTOCALL facility in SAS for secured storage and efficient invocation. Other frequently used formats, templates, scripts, and external files can be stored in the centralized location in same manner as well.

The table below suggests some possibilities of what components can be setup in a library collection.

Table 2: Setup of a centralized library collection

Component	Description
programs	– standalone procedural programs and similar functional programs – INCLUDE programs – test programs, connect programs, administrative programs
compiled macro library	– macros compiled in a saved permanent macro library – protect source code along with increase in quality and security – save compiling, CPU time, AUTOCALL facility
control datasets	– two types: statics versus dynamics – datasets storing program fixed parameters – datasets storing program changing variables values
database	– storage of database in a central location facilitates organization aspects – storage, retrieval, archival – group similar database in closer proximity
documentations	– project plans, communication documents, test reports, operating manuals – development requirements and specifications – enhance maintainability and reusability in the lifecycle
others	– PROC FORMATS, PROC TEMPLATE, ODS tagsets – SAS logs, SAS lists, reports, graphics – various scripts files, raw files, external files, language files like html, xml, java

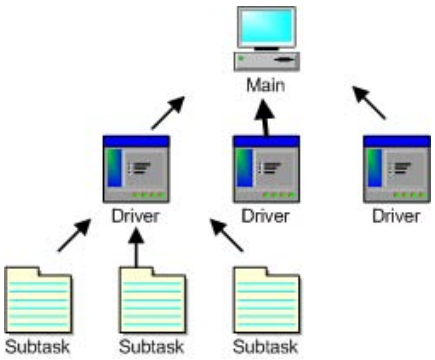
MODULAR DESIGN

The concept of practicing modular design is breaking a larger development component into separated pieces. The modules are divided into low-level subroutines to perform limited functions, and then they are incorporated back into high-level main programs as a whole. One can think of smaller individual modules as fundamental blocks of nuts and bolts in the software system.

Development cycles with modular designs representing different views vary by level of complexity and intensity, depending on the organization practices. The scale and complexity are mostly determined by the project manager or developers at the early stage of software lifecycle phases. Generally, the low level modules consist of procedural programs, INCLUDE programs, and a compiled macro library, retrievable from a centralized library collection. The top level is most likely being the interface between users and the software system. Notwithstanding the design based on data modeling or flow-process modeling, the intermediate level are series of 'driver' programs connecting boundaries between the subroutines and main interface.

The amount of interaction between or within modules to be considered is the degree of 'coupling' and 'cohesion'. In general, the desirable practice in modular design is to achieve low coupling with minimum of interactions between modules, while at the same time, manage high cohesion with strong degree of responsibility within module. The combination of low coupling with high cohesion addresses adverse dependency concern during design and

maintenance lifecycle phases, leading to lower project cost plus improvement in quality at the meantime.



The advantages of designing in modules set higher efficiency and productivity at various software lifecycle stages. Modularized programs are much easier to debug at a lower level, increasing flexibility and customization within specific function of the module, rather than taking on the program as a whole. Modular designs introduce a 'multiplier effect' by breaking down the overall development effort into subtasks where increasing number of developers can collaborate on implementation phase at a given time. In addition, allocation of workload can be diversified base upon the skill and experience of programmers.

Table 3: An example of modular design structure

Level	Description and usage
Main (top)	end user level application – package with user interface – yield business product oriented for end users, driven by output, generate deliverables – highest level in the architecture
Driver (middle)	driver program calls upon group of stored subtasks, procedures, or databases to perform assigned task – desired tasks are creatable on demand to meet needs – high interactions across top and bottom level – minimum interactions between other modules at the same level
Subtask (bottom)	independent procedural programs, include programs, macros, control datasets – retrieval from library collection to perform defined function – configuration and modification efforts are concentrated – minimum interactions between subtasks

BUILD AS APPLICATION

Depending on project requirements and user acceptance, building system software into user-driven application bears added benefits. Depending on the nature of business, customers (not the managers) are not fond of understanding what goes in neither the working design nor the process-flow behind the system generating outputs, as long as the deliverables are acceptable. To address customer usability, an application in software development inputs the functional codes, database, and other components hidden from users. The customers would then focus on interaction with the designed end-user interface and communication with the system, minimizing the 'need-to-know' technical aspects.

An advantage of building applications is the degree of configurability from earlier mentioned modular design. The flexibility from procedural programming establishes reusability at the bottom level design further lower the expenditure cost of building applications comparing to reinventing. By overlapping 'build as application' and use of library collection, different configurations and localizations become possible. This characteristic also promotes quicker prototyping and quality improvement based on initial deployment phase. Additional budget cost saving comes from adopting developed application framework or existing application, i.e. product branching and version upgrades.

Application ties communication linkage between two entities such as users to hardware, users to software, software to software, and so forth. Applications are more user-friendly and often packaged in presentable graphic user interface i.e. web-based interface, off the shelve software packages, in-house application, simple command line…etc. They execute the user's commands generating desired outputs like ones familiar as survey database, merchandise inventory report, adverse drug event statistical table, analyst portal, and so forth.

Table 4: Tools for SAS applications

Tools	Description and usage
In-house	– various SAS products packaged into in-house interface to meet the needs of business functions – Customized end-user applications driven by underlying SAS processes and functions from combination of SAS products – example: SAS/BASE for core functionality, SAS/CONNECT for delivery mode, and SAS/AF for presentation
SAS/IntrNet	– web services broadcast to multiple audiences – easing from BASE programming skills and limited html knowledge to building web applications – AUTOCALL facility, SQL web query, limited html knowledge requirement
SAS/AF	– graphical interface presentation between users and system applications – readily available procedures and built-in components expediting development time – object oriented interface
SAS Business Intelligence	– business solution product: integration between software and users, suitable for various size corporation – attractive package, access level control, board audiences, personalized portals – internal available applications and call procedure process, all with less programming – structured delivery, reporting, analysis, query, visualization
SAS/Base	– customizable, extensive functions, adaptive to user, less restrictive, compatible with other SAS applications – custom reports, dataset deliverable, graphics, tables, ODS delivery – cost effective, more human resources intensive – interfaces are not as rich as other SAS application products
Others	– Microsoft® Office® integration – ACCESS to 3rd party's database – other various SAS products

OTHER PHASES

While the scope of this paper focuses much on the efficiency aspects at development phase, yet the mentioned practices in defying limited resources are applicable to other phases in the software lifecycle as well. The payoff of adopting good practices will be greater at the earliest stage as possible within software lifecycle.

Briefly onto other phases: defining clear requirements and specifications with consistency during planning phase contributes well investment for the overall project cycle. A working configuration management in place promotes centralized repository, as well as improving risk manageability. Thorough quality assurance and user acceptance testing manages the reliability risks together with improving performance before deployment phase.

Since the business environment is dynamic, it might not facilitate mentioned practices, due to cases such as the interest of organization, accessibility, skill sets among technical groups, or a particular style between developers. Overall, it takes a well planned combination of good software engineering practices to launch and maintain an efficient lifecycle.

CONCLUSION

The paper presented practices in the software development lifecycle designed to overcome the constraints of cost, time, and quality. The suggested methods of design with procedural programming, library collection, modular design, and 'build as application' set achievable goals to lower development cost and to lead improvement in quality.

From a project perspective, saving cost and meeting deadlines might only be relative in short term objectives, but the ultimate goal is to create model software development strategy at the organization level. To defy the limited resources paradigm, the insights shared in this paper hope to inspire readers with proficient approaches to build a better, faster, and cheaper SAS software lifecycle.

REFERENCES

Bell, Doug. (1987) *Software Engineering: A programming Approach.* Prentice Hall International (UK) Ltd, London.

Gill, Paul. (1997) *The Next Step: Integrating the Software Life Cycle with SAS Programming.* SAS Institute Inc., North Carolina.

Hallsteinsen, Svein and Paci Maddali. (1997) *Experiences in Software Evolution and Reuse.* Springer, Berlin.

McClure, Carma. (1997) *Software Reuse Techniques: Adding Reuse to the System Development Process.* Prentice Hall, New Jersey.

ACKNOWLEDGMENTS

Thanks Curtis Reid, Bureau of Labor Statistics, for sharing his knowledge in software design practices. And thanks Bryan Beverly, BAE Systems, for editorial revision on this paper.

CONTACT INFORMATION

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

Edmond Cheng
U.S. Bureau of Labor Statistics
2 Massachusetts Ave., NE
Washington, DC 20212-0001
(202) 691-5458
cheng_e@bls.gov