Better, Faster, and Cheaper SAS® Software Lifecycle

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. Over emphasizing on share of 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 defy an approach to build SAS 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. After all, it makes possible for project managers and developers to promote a better, faster, and cheaper SAS programming software life cycle.

INTRODUCTION
The nature of business in engineering, manufacturing, computer science, and others, where projects involve research and development, managers often face unavoidable constraints of budget, schedule, and quality. The paradigm of limited resources affects the cost, time, and quality presents difficult dilemmas for managers in decision making. Given with constraints are generally predetermined and fixed; managers continues to 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 motivation of this paper is to consider several practical software engineering methods and its implication to SAS software lifecycle. Especially to address practices to reduce the limitation of resources throughout development stage.

WHAT IS 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 specification, 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 defy the engineering paradigms of limited resources bringing a better, faster, and cheaper SAS software lifecycle, let's to 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 program 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 section of frequent use PROC SQL code with additional flexibility in program parameter and macroization 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

<table>
<thead>
<tr>
<th>Practice</th>
<th>Description and usage</th>
</tr>
</thead>
</table>
| functional        | program should be concise to perform specific function  
|                   | − write program defines by function, rather than one to do it all  
|                   | − top-down design determines the top level requirement, then fashion details at the bottom level to meet purpose  
|                   | − knowing desire state of outcome  
|                   | − consider between data flow versus process flow  
|                   | − utilization thru standalone programs, INCLUDE programs, or macros  
| simplicity         | keep the basis brief and simple  
|                   | − write the least number of code possible for the job  
|                   | − PROC steps over DATA steps  
|                   | − remove test code  
|                   | − do not keep unnecessary variables in dataset  
| flexibility        | adaptable and expandable  
|                   | − use macro variable for parameterization; if appropriate, less hard coding  
|                   | − code defensively to anticipate changes  
|                   | − limited or no independency between procedure programs  
| maintainability    | make it effortless for anyone to maintain  
|                   | − easy to correct  
|                   | − consider use of PROC FORMAT, control datasets  
|                   | − meaningful variable name, database name, macro definition, graphics title  
| structured programming | logically break the program into smaller parts  
|                   | − logic composition, program data vector  
|                   | − concatenation, selection, and repetition  
|                   | − separation of concern  
|                   | − avoid jumping (GOTO) between or within programs  
| documentation      | add readable language to improve understandability of programming code  
|                   | − header section, style consistency, layout format  
|                   | − include comments for yourself and others, external documents come helpful  

There are enormous benefits of practicing procedural programming. Writing SAS program in functional orientation becomes easy to build a handful of accessible programs performing specific function, which can call upon and promote reusability in future. Procedural programming makes the job of maintaining or modification often a lot simpler. It helps spending less time to make changes to the programs, conversely 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 disposable. A library collection facilitates the storage and retrieval of functional codes, as well as coordination of database, documentations, and documents. Together with configuration management implementation, the task to maintain file version control is less complicating. Project managers can track project progress throughout the each lifecycle phases.

Assembling procedural programs into a library collection, promote a ‘bottom-up’ modular process design, which is will be described in the next section. In SAS, these standalone programs can be stored in a designated location, which allows a universal ground for repetitively code retrieval. Macros can make use of the compiled macros library and autocall facility in SAS for secured storage and efficient invocation. Other frequent use of 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

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs</td>
<td>- standalone procedural programs and similar functional programs</td>
</tr>
<tr>
<td></td>
<td>- INCLUDE programs</td>
</tr>
<tr>
<td></td>
<td>- test programs, connect programs, administrative programs</td>
</tr>
<tr>
<td>compiled macro</td>
<td>- macros compiled in a saved permanent macro library</td>
</tr>
<tr>
<td>library</td>
<td>- protect source code increasing quality and security</td>
</tr>
<tr>
<td></td>
<td>- save compiling, CPU time, autocall facility</td>
</tr>
<tr>
<td>control datasets</td>
<td>- datasets storing program parameters</td>
</tr>
<tr>
<td></td>
<td>- datasets storing program variables values</td>
</tr>
<tr>
<td></td>
<td>- statics versus dynamics</td>
</tr>
<tr>
<td>database</td>
<td>- storage of database in a central location facilitates organization aspects</td>
</tr>
<tr>
<td></td>
<td>- storage, retrieval, archival advantages</td>
</tr>
<tr>
<td></td>
<td>- group similar database in closer proximity</td>
</tr>
<tr>
<td></td>
<td>- archival database, backups, zip and tar</td>
</tr>
<tr>
<td>documentations</td>
<td>- requirements and specifications</td>
</tr>
<tr>
<td></td>
<td>- project plans, development documents, test reports, operating manuals</td>
</tr>
<tr>
<td></td>
<td>- enhance reusability in future projects</td>
</tr>
<tr>
<td>others</td>
<td>- PROC FORMATS, PROC TEMPLATE, ODS tagsets</td>
</tr>
<tr>
<td></td>
<td>- various scripts, raw files, external files, SAS logs, SAS lists, reports,</td>
</tr>
<tr>
<td></td>
<td>graphics</td>
</tr>
<tr>
<td></td>
<td>- various language files like html, xml, java</td>
</tr>
</tbody>
</table>

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 cycle with modular design represents 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 early stage of software lifecycle phases. Generally, modules at the lowest level consist of procedural programs, INCLUDE programs, and 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 for someone to debug at a lower level, increasing the flexibilities and customization within specific function of the module, rather than taking the program as a whole. Modular design introduces 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 based upon the skill and experience of programmers.

Table 3: An example of modular design structure

<table>
<thead>
<tr>
<th>Level</th>
<th>Description and usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main (top)</td>
<td>− end user level application &lt;br&gt;− package with some sort of user interface &lt;br&gt;− yield business product, driven by output, generate deliverables &lt;br&gt;− highest level in the architecture</td>
</tr>
<tr>
<td>Task (middle)</td>
<td>− ‘driver’ program calls upon group of stored subtasks, procedures, or databases to perform assigned task &lt;br&gt;− high interactions across top and bottom level &lt;br&gt;− minimum interactions between other modules at the same level &lt;br&gt;− additional tasks easily creatable on demand to meet needs</td>
</tr>
<tr>
<td>Subtask (bottom)</td>
<td>− independent procedural programs, include programs, macros, control datasets &lt;br&gt;− retrieval from library collection &lt;br&gt;− configuration and modification can be done at the lowest level &lt;br&gt;− minimum interactions between subtasks</td>
</tr>
</tbody>
</table>

BUILD AS APPLICATION

Depending on project requirement and user acceptance, building system software into user-driven application bears added benefits. Depending on the nature of business, customers (nor the managers) are not fond of understanding what goes in neither the working design nor the process-flow behind the system generating outputs, for as long as the deliverables are acceptable. To address customer usability, an application in software development put 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 application 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. Application is more user-friendly and often packaged in presentable graphic user interface i.e. web-based interface, off the shelf software package, in-house application, simple command line…etc. They perform user’s command 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

<table>
<thead>
<tr>
<th>Tools</th>
<th>Description and usage</th>
</tr>
</thead>
</table>
| In-house               | - add SAS product engine to in-house interface for performing business functions  
- building with SAS/BASE program codes and functions, involving additional features from other SAS products  
- combination of various SAS products to customize a package meeting business needs  
- 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 to building web applications  
- autocall facility, SQL web query, limited HTML knowledge requirement                                                                                      |
| SAS/AF                 | - graphical interface presentation between users and system applications  
- object oriented interface  
- readily procedures plus built-in components expediting development time                                                                                     |
| SAS Business Intelligence | - integration between hardware and software systems  
- business solution product, suitable for various size corporation  
- well balance package, user access level, board audiences  
- personalized portals, reporting, analysis, query, visualization  
- internal available applications, less programming  
- higher complexity in architecture                                                                                                                             |
| SAS/Base               | - highly customizable, extensive functions, adaptive to user, less restrictive  
- compatible with other SAS applications  
- custom reports, dataset deliverable, graphics, tables, ODS delivery  
- ODS delivery with attractive presentable outputs  
- human resources intensive, more cost effective  
- 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

Whilst 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 along testing and user acceptance pays off the reliability issues together with improving performance before deployment phase.

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

### CONCLUSION

The paper presented practices in software development lifecycle 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 deadline might only be relative in short term objectives, but the ultimate goal is to create model strategy at organization level. To defy limited resources paradigm, the motivation of this paper hopes to inspire readers with proficient approaches to build better, faster, and cheaper SAS software lifecycle.
REFERENCES


ACKNOWLEDGMENTS
Thanks Mr. Curtis Reid, Bureau of Labor Statistics, for guidance and support of 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

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.