Cruising to new altitudes after struggling up the Y2K hill. Lessons learnt and how to make the most of them.

Bruce Kayton, Independent Consultant, San Diego, California

ABSTRACT

The year 2000 is fast approaching and by now systems should be in the process of final remediation if not already fully tested and compliant. In the past months many individuals have been affected in one way or the other by the coming change in century. People that are closely involved, are probably acutely aware of the "if only we had done this that and the other" syndrome. In this paper I wish to present some issues that arose out of the Y2K compliance process and how we can take that experience and work more effectively and efficiently in the future.

INTRODUCTION

This paper evolved out of the occasional sense of frustration while working on a Y2K remediation effort. These flashes of frustration arose not because of the programming changes that had to occur, but by the delayed implementation of development processes, change control procedures, test environment configurations. Not to mention all the seemingly endless stream of documentation that could eradicate a small forest.

I often ask myself if things would have been any easier if the Y2K issues were addressed earlier or if life could have been a little less harassed if the control processes had been implemented long before Y2K projects even started.

Although we won't have another century change in our lifetime I would hope that we can learn from our endeavors and that we can move forward into the new millennium with a clearer vision of what constitute a healthy and ultimately a more productive system development environment.

The areas I'd like to focus on are:

- Software upgrades.
- Coding practices.
- Development environments.
- Change Control.
- Application consolidation.
- Documentation.
- Management awareness of SAS® software and it full suite of tools and capabilities.

SOFTWARE UPGRADES

Keep up to date with the current versions of SAS. While this may seem to be a budget issue especially if the current version seems to 'work just fine' the benefits of keeping up to date with technology and new direction have many rewards. Staying with the old version for too long may ultimately have a downside of having to upgrade in a hurry when the old version is no longer supported.

Some of the advantages of maintaining a current version of the software:

- Competitive advantage, you can stay ahead of the competition with additional functionality in your software.
- Enhanced productivity. Get your batch processes running quicker and as your business expands so can your user base.
- Enhanced inter system connectivity with improved information transfer and delivery.
- Employee job satisfaction. Learning new skills is always a crowd pleaser.
- Software support. Most user groups pay more attention to the most current versions of software.

Dedicate time and resources to R&D to make the most of the software at your disposal.

CODING PRACTICES.

Use comments to help anyone looking at the code to understand the purpose of complicated code. Comments should be restricted to explaining code that is not intuitive and in layman's terms.

If you need comments to understand every detail in the code, then you have no business touching the code in the first place.

Hard coding should be avoided wherever possible. I use the word 'avoid', because in a realistic world it is just not reasonable to eliminate hard coding entirely.

Keeping a history of changes to a program is commendable, but embedding it in the source code is more often than not a futile exercise. It is preferable to keep a separate change control history in a central repository. Most reasonable change control software packages facilitate this. When it is unavoidable, use some of the following programming techniques to minimize maintenance:

- Use macros or PROC DISPLAY with SCL methods to execute common code that can be used by a number of programs. This way maintenance is restricted to smaller subsets of code and is not scattered throughout numerous programs, which by definition requires more maintenance and has a higher risk of error. This technique also lends itself well towards sharing of code.
- Initialize macro variables that contain hard coded values at the top of the program for easier maintenance. Where possible use dataset values to populate macro variable values that change frequently.
- Any hard coding or calls to hard coded modules should be accompanied by comments to explain the purpose of the called modules.

Use logical or local names where possible to assign libnames and filenames. Don't hard code disk or directory references. This impacts the portability of your code.

APPLICATION CONSOLIDATION

Part of the Y2K process has been to identify redundant systems and retire them from production in order to minimize the remediation effort. In some areas this has made the coding effort even more demanding because the retired functionality has had to be transferred to a non-retiring application.

A good application development practice for the future would be to constantly access current applications and consolidate wherever possible. Not only can it save software-licensing costs, but it can also reduce maintenance effort and eliminate functional redundancies.

DEVELOPMENT ENVIRONMENTS

Keep development, testing and production environments separate. If possible have your production environment on a separate machine. This will prevent any development or testing processes from impacting production resources. Make sure that debugging options such as MPRINT, MLOGIC, SYMBOLGEN and FULLSTIMER are switched off. FULLSTIMER can have resource implications while turning the others off will just prevent unnecessary or repetitive information from being logged.

A good model for a development environment allows for different phases of application development whilst giving free reign to testers and developer alike without impacting production.

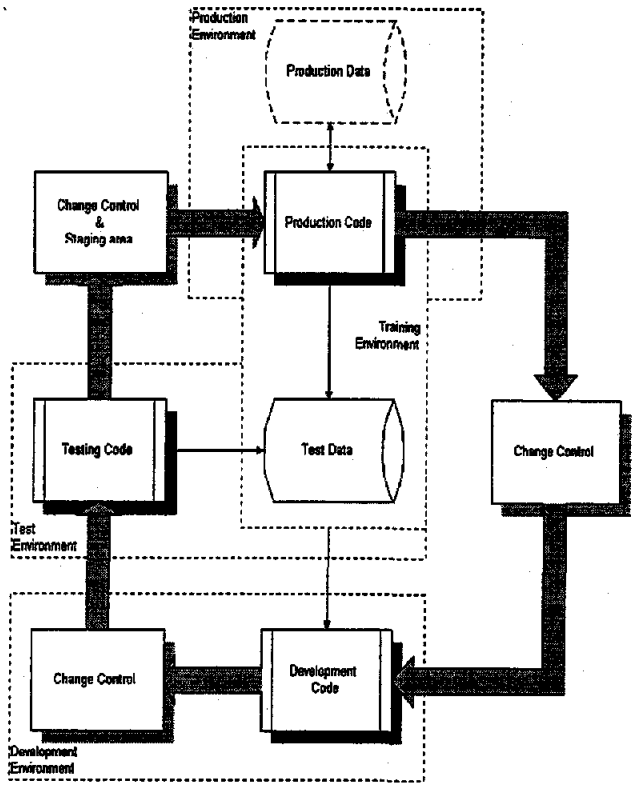
Useful software for testing incorporates scripting tools whereby test scripts are recorded against an application and results are stored for later comparison. Subsequent changes to the application then utilize the same scripts, with some appropriate modification to accommodate the new changes and results are compared.

An entire scripting library can be developed that can be just as intricate as the application itself.

CHANGE CONTROL

Shudder. Most programmers hate the mention of change control, but anyone with an objective opinion will easily identify the advantages it can bring to a development environment. A few of the advantages of change control are:

- A controlled environment that reduces the margin for error by eliminating concurrent code updates by multiple developers.
- By forcing an application to exist in multiple environments it facilitates testing and training environments.
- Implementation of coding standards.
- Tracking of system development and maintenance costs.
- Change history that can include documentation on business processes represented within a system.
- Rollback capabilities combined with disaster recovery.



The diagram above denotes a model that allows for development, testing, training and production environments. It also facilitates an area to exercise change control by preventing code from being introduced into production unless the correct process has been used. The staging area would be an area that would only allow the automated promotion of code to production once it had passed through the change control process. If sufficient controls are in place to prevent manual code transfers into production then this method can be a successful change control mechanism.

Many software packages can be used to manage the change control process, however SAS/AF catalogs do not lend themselves very well towards such software. To circumvent this problem a simple application making use of PROC CATALOG or PROC DATASETS could migrate individual catalog objects or alternatively the SAS catalog would need to be treated as a single entity.

DOCUMENTATION

If all code changes are passed through a change control process then documentation is limited to process flows and initial system design documents. Data backups and or change control versions negate the requirement for hard copy versions of programs to be stored. Test plans for any changes should have a change control life cycle of their own.

MANAGEMENT AWARENESS OF SAS® CAPABILITIES

SAS has a poor reputation among non-SAS users as being a cumbersome CPU hogging relic from the past. I've seen some code that has been in production for years and nobody dares to touch it because it works. Unfortunately it is code such as this that gives SAS a bad rap. Most people are unaware of the tremendous power that SAS software can deliver and how much of an asset it can be to a company and it's up to the individual users to explore SAS and it's potential and seek management approval.

CONCLUSIONS

Can we make use of the lesson learnt from the Y2K remediation efforts or do we do bad business as usual.

Due diligence goes beyond Y2K. You may not be around for Y10K, nor will the software you're writing now (one would hope anyway), but a disaster recovery or change in platform implementation may go that much easier if we can move ahead without the baggage of bad development practices.

Next big issue will be the change to 4-digit area code for telephones systems, will it be a synch or a coding nightmare.

ACKNOWLEDGMENTS

SAS and SAS/AF are registered trademarks of SAS Institute Inc., Cary, NC, USA

Thanks to Dave Sisemore at CCN Managed Care Inc. for introducing me to the development environment model explained in this paper as well as many other sound coding practices.

CONTACT INFORMATION

Bruce Kayton
bakayton@ix.netcom.com