Paper 271-2009

Configuration Management for SAS Software Projects

Steven O'Donoghue, SAS Institute, UK
Andrew Ratcliffe, RSTL, UK

ABSTRACT

IEEE Std-729-1983 defines Configuration Management (CM) as "the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items." After a more complete description of CM, its purpose and benefits, this paper describes the authors' experiences in implementing a CM system for a major financial project. For reasons explained in the paper, the authors considered the use of third-party packages but ultimately chose to write their own solution. The authors describe their decision-making process, the specification of their CM system, lessons learned during development and subsequent usage. The paper concludes with recommendations for applying CM to other projects.

1. INTRODUCTION

The SAS system is an enterprise approach to Business Intelligence (BI). The software is being using in increasingly more complex scenarios; projects involve more hardware, more people, longer timescales and more risk. With this in mind, it is critical that we have a standardized and repeatable methodology for deploying, configuring and maintaining the software.

Irrespective of the technology being used, all software projects go through the development lifecycle and address common areas such as requirements gathering, build, test, bug-fix etc. An important aspect of any software project is the configuration management (CM) discipline. The purpose of this document is to provide recommendations for the approach to CM for the SAS system. For our case study we refer to our experiences with a major financial institution.

This is a technical document and our assumption is that the reader is familiar with both the SAS Intelligence Platform and the general principles of configuration management.

2. NAMING CONVENTIONS

The following standard for SAS levels is used throughout this document:

Level	Environment
Lev 0	Production
Lev 1	Development branch #1
Lev 2	Test for development branch #1
Lev 3	Development branch #2
Lev 4	Test for development branch #2

Table 1: Naming convention for SAS logical levels

3. WHAT DOES COFIGURATION MANAGEMENT MEAN FOR SAS?

Configuration Management is a broad terms and has various interpretations. The International Standards Organization (ISO) defines CM as:

A process workflow whose purpose is to identify, define and baseline items; control modifications and release of these items; report and record status of the items and modification requests; ensure completeness, consistency and correctness of the items; and control storage, handling and delivery of the items.

The following processes make up the CM discipline [1]:

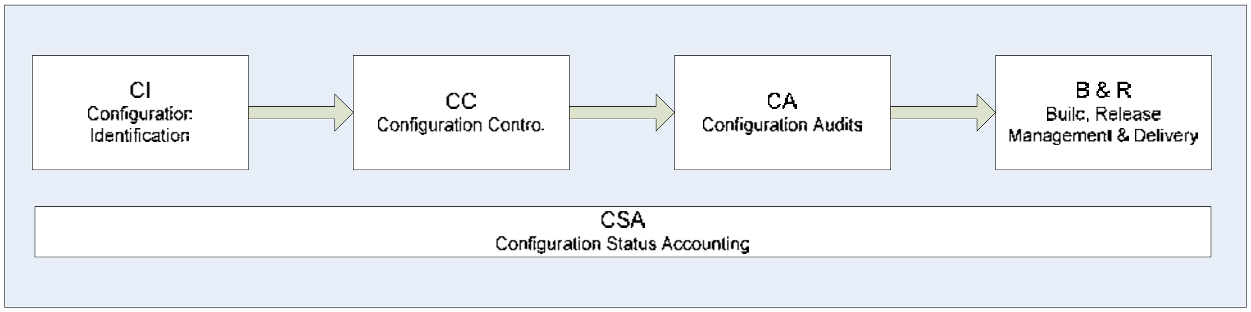

CI
Configuration Identification
CC
Configuration Control
CA
Configuration Audits
B & R
Build, Release Management & Delivery
CSA
Configuration Status Accounting

Figure 1: Core Configuration Management Processes

- Configuration Identification
  - *Which items will be controlled and managed as part of the CM process?*
- Configuration Control
  - *How do we control changes to configuration items?*
- Configuration Audits
  - *How do we prepare for formal auditing of our CM process?*
- Build, Release Management and Delivery
  - *What is the approach to build, release and delivery our software in a repeatable, scalable, auditable and efficient manner?*
- Configuration Status Accounting
  - *How do we report on our configuration management process to stakeholders (i.e. how many defects do we have and when will they be fixed)?*

Before any technical work commences, the approach to the above processes should be documented in the SAS Configuration Management Plan for the project. There are a number of standards-based approaches to CM plans e.g. IEEE Std-828-2005 [2] and an example CM plan can be found here: [3].

Clearly, the scope of the CM exercise is project specific but will invariably include the versioning of units (e.g. code files) and how to package and release the software to another environment Other areas that may be covered include infrastructure components, documentation, change requests etc. In our opinion, version control (VC) and release management present particular challenges for the SAS9 system and, with this in mind, this is the focus of our attention in this paper. The approach to VC will vary from project to project depending on the customer's preference of Source Code Control System (SCCS) and so we discuss this area generically.

3.1. SAS ARTIFACTS AND CONFIGURATION MANAGEMENT

To support enterprise business intelligence, the SAS system has a portfolio of software artifacts. For example:

Code files	Stored Process
Data sets	Formats
Views	Catalogues
Indexes	Web Reports
Data Integration (DI) Studio Jobs	Information Maps
Macros	WebDAV Content
Configuration Files	Scheduling Information
OLAP Cubes	Data Mining Models

Table 2: SAS software artifacts

Broadly speaking, SAS content falls into one of three categories:

- Text files
- Data
- Metadata

(Indeed some content spans categories e.g. a stored process has both a text and metadata representation, but for the purposes of our discussions we do not need to expand upon this.)

Text files and data are both familiar concepts; however, is it important that we explore what is meant by metadata in the SAS world. SAS9 is a metadata driven approach to BI – at the heart of the intelligence platform beats the metadata server. The SAS metadata server is an in-memory database and contains definitions of servers, data, database schemes, users, reports, stored processes, web reports, process flows etc. The metadata server understands the relationships between the various components and facilitates the sharing of content across SAS applications to provide a consistent representation across the business. The metadata server is able to do this because the metadata architecture adheres to the Common Warehouse Metamodel (CWM) [4]. The CWM defines the object-oriented structure of the metadata including all of the corresponding inter-relationships between components – for example – how a Data Integration (DI) Studio job is linked to a table, which is linked to an Information Map, which is linked to a Web Report. This complex inter-relationship between objects in the CWM implies that it is difficult to version control a single metadata object since is it inherently linked to other objects in the model – metadata objects are not standalone entities like text files for example. In the absence of any metadata versioning actually built into the SAS9 system, in our opinion, the concept of versioning metadata only makes sense at a collective level i.e. all metadata - we explore this in more depth later in the document. The best practice approach to change control of the SAS metadata server can be found in [5].

In summary, the configuration management architecture of the SAS system must account for text files, data, and metadata.

4. SOURCE CODE CONTROL SYSTEMS

4.1. INTRODUCTION TO SOURCE CODE CONTROL SYSTEMS

CVS, Subversion, ClearCase, Perforce, StarTeam are all examples of Source Code Control Systems (SCCS). The purpose of a SCCS is to:

- Track changes to units (e.g. code) under software development i.e. who changed what, where, when and why.
- Version control units and provide the ability to regress to a previous version.
- Compare differences between versions of units.
- Baseline software for a release. This is known as placing a tag in the code-base.
- Support multiple branches of the code-base so that multiple development streams of the software can be worked on concurrently.
- Support concurrent development of a single unit (e.g. a code file) (either through file locking or version merging).

4.2. SOURCE MANAGEMENT MODELS

Traditional SCCS systems work by having a centralized repository stored on a shared server[1]. Developers “check-out” units to work on, make their changes and then “check-in” the units to register those changes with the central

[1] For completeness we mention there are also SCCS which take a peer-to-peer approach to version control as opposed to having a centralized, shared repository. Such systems are called *Distributed Revision Control Systems*.

server. The scenario where two (or more) developers try to access the same file at the same time is solved through one of two source management models:

4.2.1. FILE LOCKING

In this model the repository only allows one person to edit a file at a time. This exclusivity policy is managed using locks. When a developer is editing a file, the file is “locked” and cannot be edited by another user. When changes to the file are completed, the lock is released, the changes are committed into the centralized repository and another user can edit the file.

4.2.2. VERSION MERGING

In this model, multiple users can edit the same file at the same time. Each user obtains a working copy of the file in a playpen area and makes their changes. When developers commit their changes into the repository, the SCCS provides the facility to merge the differences together so that all the changes are accounted for.

4.3. INTEGRATING SAS WITH A SOURCE CODE CONTROL SYSTEM

The SAS system has been successfully integrated with many SCCS including CVS, Subversion and ClearCase. However, we point out that the integration of SAS9 with a SCCS is not straightforward, largely because of the metadata component. We have already stressed the importance that metadata plays in the SAS9 architecture and explained that it is not suitable for version control at the level of individual metadata objects.

The method of implementation clearly depends on the SCCS being used and project requirements – for example - are there project requirements to have multiple people editing the same code file at the same time? With this is mind we can only offer some generic advice with reference to integrating with a SCCS.

The simplest approach is to avoid private working copies and use file locking to prevent concurrent development. A more sophisticated and complete approach is to use version merging where each user has a private working copy of the files they are changing. Clearly if working copies are being used each user must point there SAS session to where that working copy is held. For example, consider the Data Integration (DI) Studio job below:

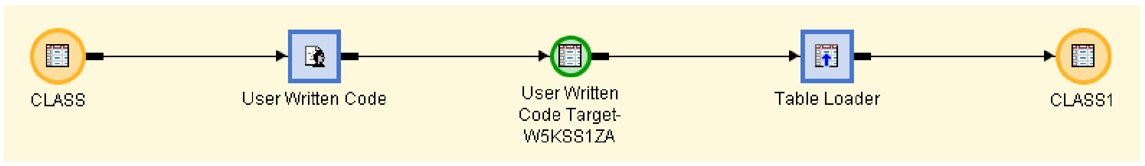
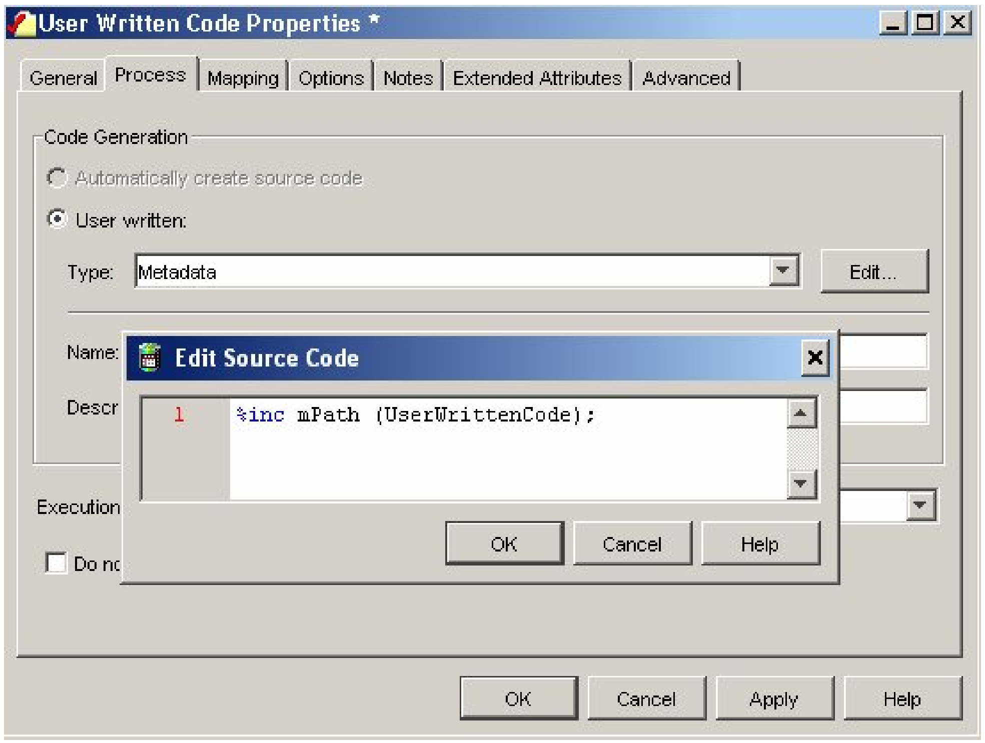


Figure 2: Data Integration Studio job flow.

This DI job contains a reference to user written code. If two developers (Harry and Sally say) are concurrently working on the user written code using a SCCS then each would have a private working copy of this code file. To test his changes, Harry would want to run the job and pick up the user written code in his working area. However, Sally would want to run the same job and pick up her copy of the user written code in her working area. In the SAS world, the SAS configuration files (e.g. `sasv9.cfg`) and the `autoexec.sas` files are used to configure the SAS session when it starts and point the session to particular areas on disk. We can easily configure these files to point to private working copies. For example, the `autoexec.sas` could contain this line:

```
   filename mPath "C:\SASPrograms\&sysuserid";
```

where `sysuserid` is a global macro variable containing the user’s identity. By configuring the SAS session this way, each user references a different area on disk for the filename mPath when the SAS session starts. We then configure the user written transform in DI Studio Job the following way:



Type: Metadata

```
%inc mPath (UserWrittenCode);
```

Figure 3: User written code within a DI job flow.

where `UserWrittenCode` is the name of the code file on disk that the User Written transformation calls. Now, when Harry and Sally run the DI Job, each can reference separate code files that are stored in their private working area. This approach illustrates a technique to integrate SAS with a SCCS. However, we stress that each SCCS is different and has its own integration challenges.

4.3.1. TAGGING AND BRANCHING IN A SCCS

The concepts of tagging and branching are central to source code control systems. A tag is used to mark a significant event in the lifecycle of the project's code. For example, when you want to baseline the code-base for a release of the software you place a tag in the SCCS called, for example, Release 1.0 so that all components that constitute that release can be referenced at a later date.

Branching in a SCCS offers the facility to duplicate the code-base at a particular point in time and allows the SCCS to maintain these two streams of development (i.e. branches) separately. Why is this a useful feature? Let's say that Harry and Sally are preparing Release 1.0 of the software. On Monday, they place a tag in the SCCS called Release 1.0 and this baseline of the code-base is deployed to a Production environment. On Tuesday they begin work on Release 2.0 of the code-base. On the Wednesday, however, a problem is found with Release 1.0 that is in Production and Harry is asked to fix this. Harry needs to go to the Release 1.0 tag in the SCCS and create a new code branch. This branch contains the code-base from production. Harry can fix the problem and place a new tag in the new branch called Release 1.1. Release 1.1 can then be deployed to production as a defect fix. Essentially, the branching facility enables concurrent development of separate work streams…in the example above, working on two releases at the same time.

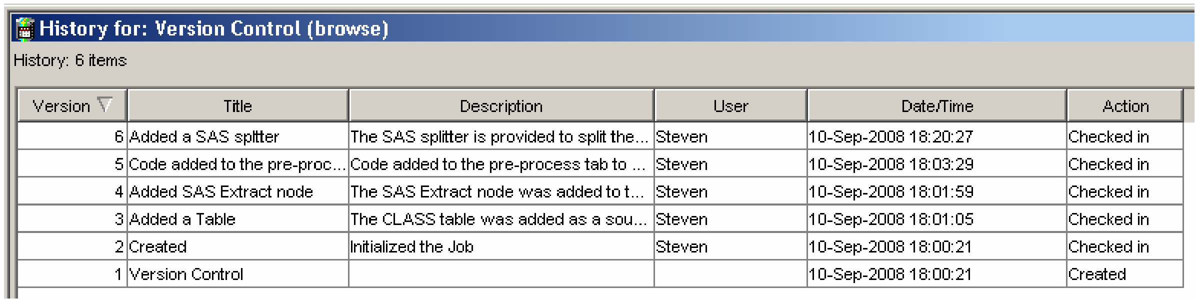
In the SAS world, tagging works well. Before we place a tag in the code-base, we export our SAS metadata components as a package(s) and place it in the SCCS. We then baseline the release by placing a tag in the SCCS. Although we can not version control individual metadata objects, placing metadata in the SCCS at the point of tagging gives us a single version of what constitutes the code-base for a particular baseline.

Unfortunately the traditional concept of branching in an SCCS is not compatible with a SAS9 deployment. The issue is that SCCSs control code and SAS9 is much more than that - specifically we have the SAS metadata server. With this in mind, a branch in the code-base of a SAS9 project requires a separate instance of the SAS software. Let us assume that a SAS project has three SAS levels set up: Lev 0 (production), Lev 1 (development) and Lev 2 (test). Let us also assume that Release 1.0 is in Production (Lev 0) and that Development (Lev 1) is now being used to develop Release 2.0 of the software.

In the event that we have a problem with the Production software that need to be addressed, which SAS level can we use to develop this fix? Clearly we can't do this straight in the Production environment. However, we also can't do this in either Lev 1 or Lev 2 because the code-base in these environments is not representative of Production…they have moved on to support development of Release 2.0! What we need is another SAS level (i.e. a separate instance of the SAS software) to support a branch of the code that represents the code-base in Production. In this separate branch we can develop the fix for the Production issue [in fact we actually need two additional SAS levels – one environment to develop the fix (Lev 3) and another to test it (Lev 4)]. We return to this subject in more detail in the Release Management section of this document.

4.4. DATA INTEGRATION STUDIO AND CHANGE MANAGEMENT

It seems appropriate, at this point, to discuss the change management facility in DI Studio. This facility is not a SCCS but shares some similarities. DI Studio is the only metadata driven SAS application that supports change management. In order to use this facility a developer must have a project repository set up in the SAS Management Console (MC) and the appropriate metadata permissions (e.g. grant *CheckInMetadata*). The change management facility supports the file locking management model detailed above. When a developer checks out a metadata object, it becomes locked and the developer has exclusive permissions to edit the object. After the changed have been made, the developer performs a check-in of their project repository to register the changes with the parent repository. As part of the check-in, it is mandatory to provide details on the changes that have been made. For metadata objects that have been maintained under the change management facility, DI Studio provides a history of the changes:



History for: Version Control (browse)

History: 6 items

Version	Title	Description	User	Date/Time	Action
6	Added a SAS splitter	The SAS splitter is provided to split the...	Steven	10-Sep-2008 18:20:27	Checked in
5	Code added to the pre-proc...	Code added to the pre-process tab to ...	Steven	10-Sep-2008 18:03:29	Checked in
4	Added SAS Extract node	The SAS Extract node was added to t...	Steven	10-Sep-2008 18:01:59	Checked in
3	Added a Table	The CLASS table was added as a sou...	Steven	10-Sep-2008 18:01:05	Checked in
2	Created	Initialized the Job	Steven	10-Sep-2008 18:00:21	Checked in
1	Version Control			10-Sep-2008 18:00:21	Created

Figure 4: Version history in Data Integration Studio

As part of the history, the metadata records the change against the version number of the metadata object. However, it is important to point out that it is not possible to regress to a previous version of the metadata, view previous versions of the metadata or compare versions of metadata to establish differences. The extent of the change management facility is to provide a lock on the object and provide a history of changes.

Best practice dictates that project repositories should always be used during DI development. DI Studio only allows check-in at the level of the project repository. For example if one checks-out ten objects but only modifies five of those objects, the whole project repository must be checked-in, including the objects that have not changed (single object check-in is provided in the SAS 9.2 release of DI Studio 4.2). With this is mind, the following guidelines should be used when developing in DI:

1. Enforce the use of project repositories when developing in DI Studio through metadata security. A separate account with a metadata profile set up to connect to the parent repository could then be used to deploy jobs.

2. For each change, check-out the minimum number of objects that you need to make that change. Make the change, check-in the project repository and then move onto the next area of change.

   For example, if one needs to edit two DI Studio Jobs called Job1 and Job2, follow this process:

   a) Check-out Job 1. Make the change. Check-in the repository with an appropriate comment.

   b) Check-out Job 2. Make the change. Check-in the repository with an appropriate comment.

Don't check-out Job 1 and Job 2, edit the jobs and then check-in together. Treat each area of change as sequential activities.

3. The check-in / check-out process is memory intensive and therefore should be restricted to as few objects as is feasible to work with for that particular change.

5. THE BI MANAGER

The BI Manager is such a critical part of the release process that it is worthy of a section of its own. The BI Manager is a plug-in to the SAS Management Console that provides the ability to perform partial promotion of SAS metadata components from one environment to another. Details of the BI Manager and special considerations for its use can be found in Chapter 18 of the System Administration Guide [6]. There are, however, a number of features of the BI Manager that we wish to highlight:

5.1. CONTENT LIMITATIONS

The BI Manager can not move all metadata objects (see [6] for a comprehensive list). For example, Access Control Templates (ACTs) and Information Delivery Portal customizations such as themes are not possible to promote using the BI Manager. These must simply be re-created in the target environment following the same approach used in the source environment. The implementation approach for these areas should be documented with such documentation being placed under VC.

5.2. MEMORY LIMITATIONS

Experience has shown that, as a project evolves, it becomes increasingly difficult to export / import volumes of metadata using the BI Manager. We have found that memory limitations mean that the export / import process must be partitioned into smaller chunks. Our recommendations for a successful metadata export / import are:

1. Be aware of the memory limitation of your operating system.

2. Try a server side export / import – typically a server will have more RAM than a client PC.

3. In these files:

   a) `etlstudio.ini`
   b) `sasmc.ini` (Windows) / `sasmc` (UNIX)

   Change or add these parameters to the `CommandLineArgs` statement to improve the memory performance for the metadata export / import process:

   a) `DentityExpansionLimit`
   b) `Xmx`
   c) `Xss`

   We have successfully used the following combination of parameters:

   ```
   CommandLineArgs=-DentityExpansionLimit=1000000 -Xss1024k -Xmx1024m …
   ```

   For advice on the optimum values of these parameters for your OS see the SAS documentation [7].

4. Separate your metadata export / import process into smaller chunks.

5. Check whether you should be exporting / importing access controls – do you have metadata security applied?

After exporting the metadata, you should have one, or more, metadata packages (i.e. .spk files) and a metadata export log. The log should be reviewed for errors and warnings.

5.3. BATCH LIMITATIONS

In SAS 9.1.3 there is no facility to export / import metadata in batch – this is a new feature in SAS 9.2 [8]. In SAS 9.1.3 we must use the GUI driven BI Manager to perform this task.

5.4. DEPLOYING DATA INTEGRATION STUDIO JOBS

Once DI Studio jobs have been promoted to the target environment using the BI Manager they will need to be deployed in order to generate the source code for these metadata objects on the file system. In our experience, the deployment of DI Studio jobs can also result in memory issues. Jobs can only be deployed using the GUI interface in DI Studio. Depending on the number of jobs and their complexity, the deployment process - although trivial to do - can become sufficiently time consuming to be the bottleneck in the deployment of a release. There is no batch interface to this facility and we are not aware of any plans for this in SAS 9.2.

A strategy which we discuss here, but at the outset stress should be adopted with caution is not to deploy jobs in any environment other than Development. Rather, we simply copy the generated code from the deployment in the Dev environment to the sister location in Test / Prod. Clearly, this can not be a straightforward copy since the generated source code will reference environmental variables particular to the Development environment (e.g. the metadata server port number). Our copy would have to go through a regular expression parsing script to whip out environmental variables and replace them with those for the target environment.

If your Dev and Prod environments are configured differently, the SAS generated code can account for this by reading environmental information from the metadata server as part of the deployment. Our reasoning for exercising caution with our scripting technique is that, although this approach may reduce deployment timescales, depending on the complexity of your environment, getting your scripting to generate the exact code that would have been system generated through a metadata deployment may be non-trivial.

5.5. RECLAIMING UNUSED DISK SPACE BY THE METADATA SERVER

All this moving metadata around using the BI Manager and potential deleting of metadata from a target environment before importing a new package file means that the metadata repository datasets will grow in size. The system admin guide [6] points out that*…"When metadata is deleted from a SAS Metadata Repository, the record is removed from both memory and disk. However, the disk space that had been allocated for the record remains in the data set after the record is removed"*.

To reclaim the unused disk space you must run `%omabakup` with the `REORG=YES` option set. Every project should be running `%omabakup` on a regular basis to back up the metadata server. Since the `REORG=YES` option involves some overheads the system admin guide recommends that this is run once a month when large amounts of metadata have been deleted.

6. RELEASE MANAGEMENT

6.1. INTRODUCTION

Modern software development methodologies such as Extreme Programming [9], the Rational Unified Process [10], Scrum [11] etc adopt an iterative [12] approach to software development as opposed to a waterfall [13] style of implementation. An iterative approach is cyclical in nature:

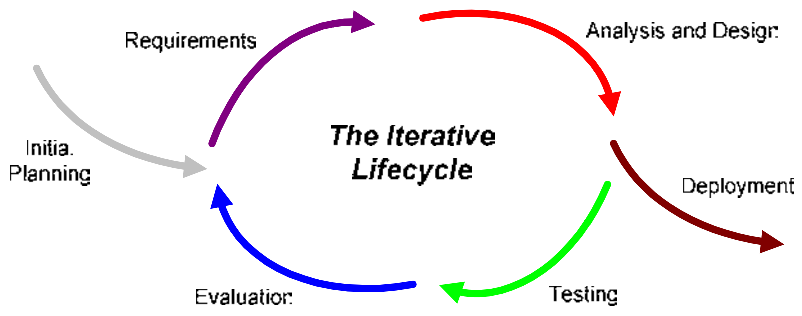


Figure 5: Iterative approach to software development

Where as a *waterfall* approach is exactly that…

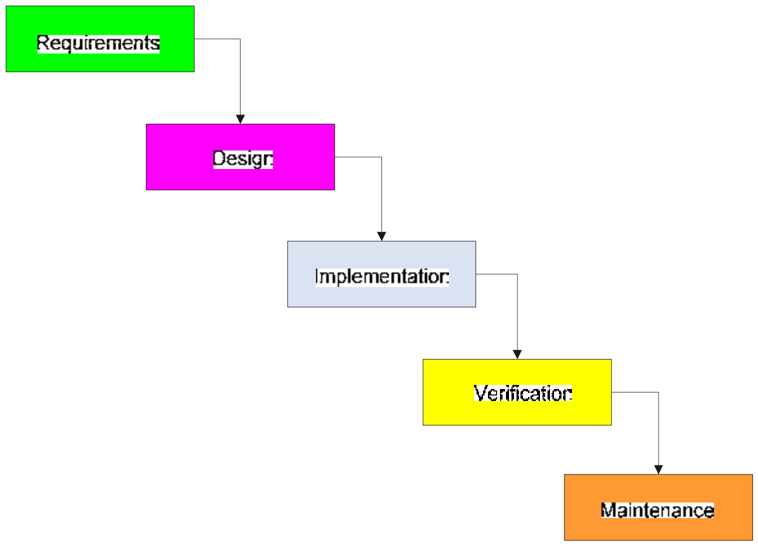


Figure 6: Waterfall approach to software development

The main benefit of iterative styles is that they are more “agile” [14] – the functionality that the development team delivers is iteratively revisited to constantly improve the software, whether that be because requirements have changed; to address performance issues etc. The waterfall approach is “big-bang” in nature and stresses getting everything right at a particular phase in the project before moving on to the next.

The breath of the SAS platform and (typically) large scope of business intelligence engagements lends itself to an iterative style of implementation. To support this approach, however is critical that the interfaces between the various software development stages are efficient, reliable and repeatable steps i.e. if we are constantly deploying code, the act of deployment itself must be a slick process.

In our next sections we discuss an approach to baseline the code-set; release the code-base to another environment and; most importantly, with our configuration management hat on, how we keep track of what it is that we have deployed?

Our intention is to define a standard and repeatable release management process that firmly sits within the iterative development lifecycle and is SCCS independent. There are a number of release management scenarios that we need to address:

- Developing for the first release of the system

- Addressing defect fixes for the Production environment
- Developing for concurrent release.
- Complex / ad-hoc changes (i.e. data model updates / system configuration files)
- Backing out changes

Before we discuss any of these scenarios, however, we should be clear about roles, responsibilities and documentation.

6.2. ROLES AND RESPONSIBILITIES

The roles to be performed as part of the CM process are project specific and should be defined in the Configuration Management Plan for the project. However, in our experience, there are three roles that must be fulfilled to ensure a successful CM implementation: Developers, the Configuration Management Administrator and the Release Manager.

6.2.1. DEVELOPERS

Every developer should be aware of the approach to CM that is being used on a project. Should a SCCS tool be in use, developers must understand the integration with SAS and be trained in the package.

Developers are responsible for

- The initial delivery of software components.
- Unit testing those components.
- Packaging those components.
- Documenting the contents of that package i.e. creating a Release Note (RN).
- Addressing any bugs / defect fixes to software components.

6.2.2. THE CONFIGURATION MANAGEMENT ADMINISTRATOR

The Configuration Management Administrator (CMA) plays a key role in the CM process. The CMA is the technical authority for CM for the project and assists in designing and supporting the technical implementation and any issues that arise from this. If a SCCS is in place, the CMA is an administrator of the system. The role should be performed by one or more members of the SAS development team.

The CMA is responsible for executing the developers' Release Notes to perform a release (whether that is partial or full). Security should be applied to the target environments (e.g. Test, Prod) so that only a CMA can perform a release. (This dictatorial approach to executing a release is necessary to control what is moving between environments.)

The CMA works closely with the Release Manager (detailed below) to ensure that the Release Management Spreadsheet (RMS) is updated.

6.2.3. THE RELEASE MANAGER

Essential to the success of any Release Management strategy is an owner for the process i.e. the Release Manager (RM). The responsibilities of the RM are defined as follows:

- Schedules releases
- Works with the customer to discuss what components are expected for delivery in a release
- Co-ordinates resource to package and deploy a release
- Signs-off a release as being both complete and accurate
- Works with the CMA to improve the configuration management process
- Understands exactly what components of the solution are in which environment and documents this in the Release Management Spreadsheet (discussed below).
- Before issuing a full release, compares the release with the previous release and validates the changed units as those corresponding to the RN expected for this release.

- Owns any issues which arise out of the release management process.

For small projects, the release management role is typically performed by the Project Manager (PM).

6.3. DOCUMENTATION

6.3.1. RELEASE NOTES

As developers write software, we need some means of documenting the units that are being created / edited for the following reasons:

- Potential audit requirements require a history of changes.
- Provides a list of units so that we can move this functionality to another environment i.e. it is supporting documentation for the release process.
- Provides a list of units that have changed should these need to be regressed to the previous state.

The Release Note (RN) provides a means of capturing this information. An example RN is available, upon request, from the authors. If a developer is tasked with a piece of work to archive some data, then maybe s/he writes some code and develops DI jobs. After development and unit testing have been performed, these components would be documented in the Release Note. Population of a release note following a piece of development work should be mandatory.

6.3.2. RELEASE MANAGEMENT SPREADSHEET

At any arbitrary point in the development process *N* developers may be producing *M* release notes. How do we keep track of all these changes? What RNs goes into what release? How do we track what RNs are in what environment (e.g. Development / Production)? Where do we sign off a release as being complete and accurate?

The Release Management Spreadsheet (RMS) provides a template for capturing this information. An example RMS is available, upon request, from the authors. Our style of RMS has two tabs – one to track Release Notes, the other to track releases.

The RMS is a living document; accountability for ensuring this spreadsheet is complete and accurate lies with the RM.

6.3.3. ARCHIVING DOCUMENTATION

Release Notes and the Release Management Spreadsheet constitute important project deliverables and could be required in the event of a project audit. They should be stored on a centralized network share that the development team has access to. An example folder structure is given below:

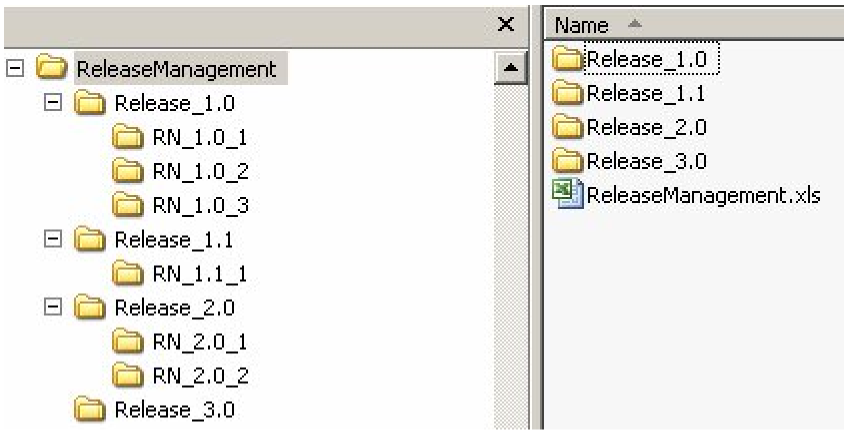

```
ReleaseManagement
    Release_1.0
        RN_1.0_1
        RN_1.0_2
        RN_1.0_3
    Release_1.1
        RN_1.1_1
    Release_2.0
        RN_2.0_1
        RN_2.0_2
    Release_3.0
```
```
Name
Release_1.0
Release_1.1
Release_2.0
Release_3.0
ReleaseManagement.xls
```

Figure 7: Folder structure to archive release management documentation

The folders are organized by release. Each release folder has the following naming convention:

Release_<Major Release #>.<Minor Release #>

A major release is one which delivers considerable new functionality i.e. at the end of an iteration of work. A minor release is typically the application of defect fixes to a major release.

Each release has a sub-folder for each RN. The naming convention of the subfolders indicates the release that the Release Note is contributing towards, followed by the Release Note number:

RN_<Release #>_<Release Note #>.

The RN numbers are just sequential positive integers. The RMS sits at the root of the ReleaseManagement folder.

6.4. THE FIRST (FULL) RELEASE OF THE SYSTEM

We have a number of different release scenarios to consider. However, the easiest to describe, is when we are developing for the first release of the system. We take this opportunity to describe a number terms and features that we then use elsewhere in the document.

6.4.1. REQUIRED ENVIRONMENTS

The first release of a SAS9 project will require at least three instances of the SAS software to support the following functions:

- Development
- Testing
- Production

Best practice dictates that the production system should always run on separate hardware. However, the development and testing SAS environments can run on the same physical hardware but be logically separated by SAS levels. (Depending on project requirement, even more environments may be required to support system testing, user acceptance testing, performance testing etc.). Our naming convention for our SAS levels is as per Section 2.

In the example below, server B has two logical levels: Lev 1 to support development and Lev 2 to support testing. A separate physical box runs the production system under a Lev 0 logical level.

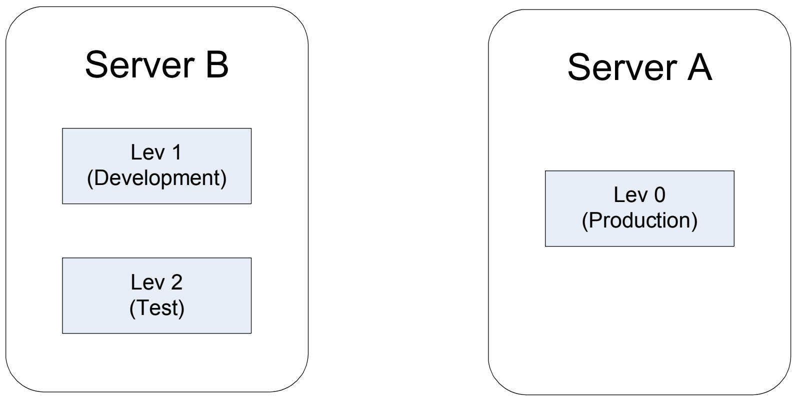


Figure 8: Example server configuration to support an initial release.

Another valid configuration would be to have each logical level running on separate hardware - the main point is that we have at least three environments. For the purposes of this document we assume that the server configuration in Figure 8 has been adopted.

6.4.2. ENVIRONMENT USAGE

Development

The Dev environment is used by the development team to write new software and unit test their work. If an SCCS tool is used, it is this environment that the tool is configured to work with. Security on the development environment is sufficiently open to allow developers to edit, move, delete and copy files and metadata.

(On large projects, our development teams have been split into DI developers and BI (Business Intelligence) developers and we have experienced conflict between these two groups with the usage of the Dev environment. This has arisen because of the natural dependency of BI on DI. The DI developers must produce tables for the BI team to use for information maps, web reports etc. The conflict arises when the BI team is half way through development and the DI developer wants to delete or modify data and tables that the BI team is currently using. The dependency has led to all sorts of duplicate data, duplicate metadata and synchronization issues – undermining our configuration management process. Although we do not present a solution to this problem here, we highlight it as a potential issue to watch out for on future projects.)

Test

Testing is, again, a broad term and the remit of the testing activity will be project specific. However, as a minimum, the testing discipline should be used for unit testing, system testing and regression testing. We discuss testing in more detail later in the document. Security on the Test environment is such that only an administrator(s) can modify the system.

Production

This environment is used to run the Production service i.e. the live version of the software. Security on the Production environment is such that only an administrator(s) can modify the system.

6.4.3. PROMOTION PATH

In SAS, the word ‘promotion’ formally refers to the movement of metadata from one environment to another. For the purposes of this paper we extend the use of this term to refer to the movement of any object (whether it is code, metadata, data etc) to another environment.

For the first release of the software, the promotion path is straightforward. Developers iteratively perform their work in the Development environment. Once components are ready for testing, these units are moved to the Test environment. Should testing find any issues with the deliverables from Dev, then we go though the cycle again…Dev, partial release, Test. This iterative process continues until all of the components required for the first release are in the Test environment and have been validated.

Once we are confident that the Test environment is correct, we are ready to deploy the release into Production. As part of the deployment process, we archive the release in the Release Repository which we discuss in Section 6.4.8. We move all text files, data and metadata into the production environment as a full release of the solution.

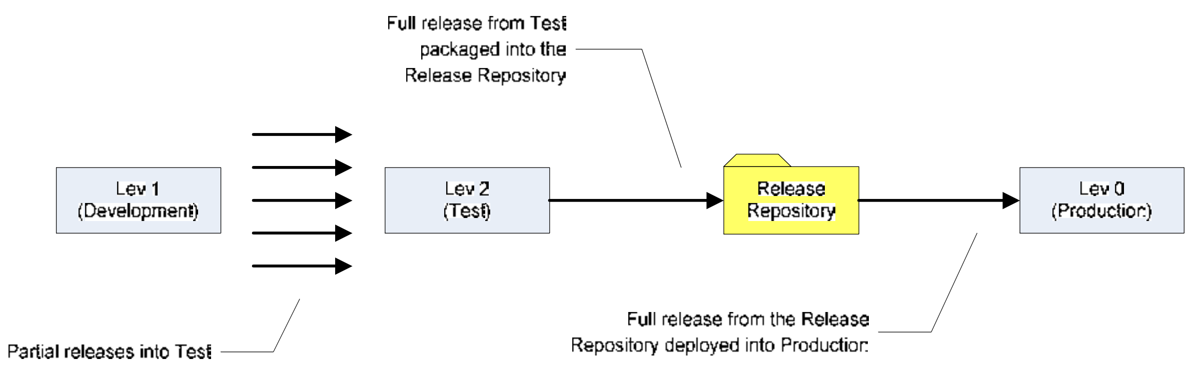

Full release from Test packaged into the Release Repository
Lev 1 (Development)
Lev 2 (Test)
Release Repository
Lev 0 (Production)
Partial releases into Test
Full release from the Release Repository deployed into Production

Figure 9: Promotion path for the first full release of the system.

(With reference to metadata, we are not here referring to full repository promotion, but rather using the BI Manager to move all metadata that have explicitly been developed for this release i.e. DI jobs / information maps but not environmental metadata such as server definitions / users. Using full repository promotion for the first release may, indeed, be valid and can be used at the discretion of the project. However, be warned that using this technique replicates all metadata in the target environment exactly and this usually is not a desirable effect. For example metadata users / security typically vary by environment.)

6.4.4. THE RELEASE CYCLE

Even though we are talking about the first (full) release here, it is safe to assume that there will be more than one release of the software. A sensible pre-requisite before writing any software, is for the RM to establish the nature of the full release cycle:

- How many releases of the software are required?
- When are the releases schedules for?
- What will be in each release?

For example, the software could be delivered in ten releases, two releases per week on Tuesday and Thursday. As a minimum we should understand the deliverables for the first release and when the due date for this release is!

6.4.5. PARTIAL RELEASES

Let us assume that we are ready to being writing software in the development environment and, for the first release, we have five pieces of functionality to be delivered.

The developers write the software and, for each deliverable, write a suite of unit tests. As far as practical, unit tests should be developed programmatically so that they can be automated to run at a later stage in the project (clearly the interpretation of unit testing must be aligned to the component. For example unit testing SAS code is different to unit testing a SAS Web Report).

The components that have been developed or altered, for each deliverable, are documented in a RN. In our example we would have five RNs delivered for this release. Each of these RNs would be recorded in the RMS by the RM.

The purpose of the RN is not only to document changes but also explains how to move the components from one environment to another. As part of the promotion process, the units that are to be changed are backed up and the new version of these units archived on the file system. In the case of metadata we can use the folder structure in Figure 10 to support the backup and archiving of metadata based components by creating a 'Backup' and 'New' folder beneath each release note folder e.g.

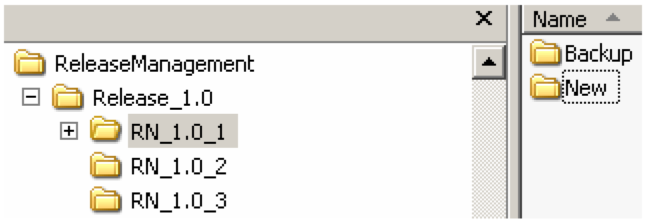


Figure 10: Folder structure to backup and archive metadata changes.

The backup and archiving of text files is not quite so straightforward. We can not necessarily use the folder structure above since this uses the Windows OS and we may want to store, for example, UNIX files. In addition, it is convenient if the folder structure reflects the location of the file that is changing.

For example, if we need to change the file

```
\\<<ServerName>>\SAS\Lev1\SASMain\SASCode\code.sas
```

on the target environment then we should mimic this folder structure to support backing up the old version of this file and *archiving* the **new** version:

```
\\<ServerName>\RM\ReleaseNotes\SAS\Lev1\SASMain\SASCode\RN_X.Y_Z\Backup\code.sas
```

```
\\<ServerName>\RM\ReleaseNotes\SAS\Lev1\SASMain\SASCode\RN_X.Y_Z\New\code.sas
```

In the above pathname, RM stands for Release Management and RN_X.Y_Z represents the Release Note documenting the change. We recommend that this folder structure (i.e. the folder structure with the root: C:\ReleaseNotes\) is maintained on the target environment since this is the environment that is changing.

The combination of the RN and our backup and archiving folder means that we have a historical, auditable and documented set of changes. Should we need to, we can restore the original versions of the files using the backup locations.

When we are documenting our units in the RN and exporting them to the file system…how can we be confident that we have captured all of the necessary units? Of particular concern are metadata changes. If we update metadata to change the name of a column, how do we capture all associated changes? For example that column may be used in n DI Studio Jobs. This can be mitigated by using the impact/reverse impact analysis feature in DI Studio for all metadata objects that have been modified.

The CMA should perform the promotion of the RNs to the target environment under the authorisation of the RM - this action, of course being tracked on the RMS. The BI Manager is used to move metadata objects. Metadata objects that have changed should be deleted from the target environment before importing the new versions. Operating system commands are used to move text files. SAS is used to move data objects [we advise SAS over the OS here (to ensure that indexes are moved correctly and to avoid issues with SPDS tables for example)].

Once a RN has been delivered into the Test environment, the functionality should be unit tested again. Since this has already been unit tested in the Dev environment, what is the reasoning for unit testing again in Test? The Dev environment is a 'dirty environment' in the sense that it is open for access with many developers making concurrent changes. With this in mind, the results of a unit test in Dev can not be guaranteed to be the same as those in Test and therefore should be validated again in the target environment. The fact that unit tests should be in the form of executable code means that re-executing in Test should not be a time consuming task.

How can we be confident that the functionality we have just deployed has not impacted another part of the system? Have we added some great features but also broken what has been working for the past six months? In order to validate that our system still works after executing a RNs, we must regression test the whole system.

Each of the five pieces of functionality that we are delivering in this example should be accompanied by a set of unit tests that can be run in an automated fashion. As more and more functionality is added to the system, out test suite grows. The test suite forms an important part of the code-base and essentially gives us the ability to perform regression testing. Regression testing the software confirms that the results that we get out of our software now, are the same as those results that we got out previously.

Regression testing can be performed at the level of a single unit, or the entire system. In order to confirm the global impact of our changes we must do the latter and run the full test suite on a regular basis to regression test the full solution e.g. performing regression testing on a nightly basis.

There are a number of testing frameworks to support these activities such as JUnit (Java), NUnit (.NET). A framework for unit testing the SAS system called FUTS has been developed by ThotWave [15]. Setting up and maintaining a testing framework is a non-trivial exercise and clearly the scope and approach needs to be visited in light of project requirements. The purpose of this document is not to explain how to set up the framework, but rather to instil the right cultural mindset to the testing discipline in light of the impact on the release management process. Release management is all about 'change' and it is only though a rigorous approach to testing that we can exude confidence that we have changed the right thing without impacting the rest of the system.

Below, we provide a summary of the partial release process:

#	**Task**
1	Developer delivers the functionality in the Dev environment.
2	Developer unit tests functionality.
3	Developer performs impact / reverse impact analysis on metadata objects to help diagnose the knock-on effect to other metadata objects.
4	Developer writes a Release Note.
5	The RM authorises the promotion of this RN to the Test ENV and this is captured in the RMS.
6	The CMA executes the RN. Metadata objects that changed are deleted from the target ENV. The RN and the old and new versions of files and metadata are stored on the network.
7	The developer unit tests the deliverable in Test.
8	Full regression testing of the system on a daily / nightly basis by running the test suite.
9	The above steps are repeated for each RN.

Table 3: Summary of the partial promotion process.

6.4.6. DEFINING A FULL RELEASE

In our example, let us assume that our five release notes are in the Test ENV and that our regression testing shows that everything is working fine. At this point, the first release is complete. We now need to package this release up, from the Test environment, and deploy it into Production.

What exactly constitutes a full release? What components do we package up?

The constituent components are those that the software development team has

- Added above and beyond the vanilla SAS installation.
- Changed from the vanilla SAS installation.

We do not include environmental metadata definitions such as servers / users

There are three major areas that contribute towards the release: text files, data and SAS metadata:

Text Files

Text files for a project will consist largely of code files. Code files should not contain any hard coded references that are environmentally specific e.g. references to server names. There are three approaches that we have used and recommend to get around this.

Always use relative path names (e.g. `../SASMain/SASEnvironment`).

Place all the hard-coded values that you need into a single file as macro variables (say `ProjectParameters.sas`). This file and its content can then be referenced as and when needed by using the %include statement. The `ProjectParameters.sas` file could (and probably would) be different per SAS environment.

For example say we have twenty code files that reference a password in `ProjectParameters.sas` - rather than change twenty files to update this password - we just update `ProjectParameters.sas`. The main advantage is that since our twenty code files reference values from `ProjectParameters.sas` through macro variables (and therefore do not have any hard-coded values), we can easily move them between environments.

Set system environment variables via the operating system (per environment) and then access these in SAS via the `%sysget` function. For example in Dev we could set the environment variable `PORT=8561` and in Test we could set `PORT=8562`. However, our code below would now be portable between these environments and yet use different port numbers for each:

```
let PortNo=%sysget(PORT);
```

Writing code files in this way means that they are portable and can be easily moved from one environment to another. All code files that have been developed for the project contribute to every full release.

There will also be configuration files that have been updated (e.g. `sasv9.cfg`). Configuration files will always be environmentally specific and, by definition, should rarely change - if ever. This nature of configuration files means that they are never considered part of the full release cycle – it is not possible to move them without changing them afterwards anyway. Configuration files are updated in an ad-hoc manner that we discuss in Section 6.7.

Data

Where possible data should not be moved from one environment to another – rather is should be derived or generated as required (the same approach it true for catalogues e.g. SAS formats). However, they will be times when data sets form a critical part of the solution (maybe as configuration data) and should be moved from one environment to another as part of the full release cycle.

Metadata

SAS metadata will be generated and stored by the SAS administrator (defining users, configuration settings, portal customisations, security etc, the development team (DI Studio jobs, tables, stored processes etc) and end users (information maps, web reports etc).

Typically, the administrative metadata elements of the system (e.g. metadata users) are different per environment and therefore we are not interested in porting 'administrative' metadata to another environment (we would change this in an ad-hoc manner across Dev/Test/Prod). End user metadata (e.g. Web Reports) is not something that we have control over generating and the only thing we must ensure here is that we don't overwrite or delete it!

The SAS metadata that we interested in capturing, as part of the release cycle, are standard metadata artifacts as generated by developers during the software development lifecycle. The type of contents that we are referring to is that found under the Custom Tree and reporting elements under the BIP tree. When we refer to moving all metadata in the remainder of this document it is this 'developer' metadata that we mean[2].

6.4.7. PACKAGING A FULL RELEASE

Hopefully, project specific knowledge together with the guidelines above gives us a good indication of the components that we need to move into the Production environment. Although we glossed over administrative parts in the previous section, we revisit this in Section 6.7. How do we collate our release components into a package so that we can move and deploy it into another environment?

[2] Clearly the metadata components to be moved are project specific and these are general guidelines.

The release process must be an efficient and repeatable process and, with this in mind, we should automate the packaging of the release as far as possible.

Text and Data Files

Text and data files for the project are likely to be distributed on a number of servers under a number of directories. A control file should be created that lists the servers and directories that hold content for the release. This control file should then feed into a script(s) that iterates through the system and collects (e.g. a UNIX tar) and zips (e.g. a UNIX gzip) these areas into one or more packages.

On UNIX the shell provides an immediate utility for scripting and often a distribution of Perl is available. On the Windows OS, VBScript, Perl, Windows Services for UNIX and the Windows PowerShell (Monad) provide some scripting options (although you will have to get to permission to install some of these). Our recommendation for using scripting to do this as opposed to SAS code is because the task at hand relies heavily on operating system commands. Although SAS is capable of executing OS commands through the 'X' command for example, a default SAS installation does not have the `allowxcmd` option turned on for the objectspawner. Therefore, when the workspace server is launched the execution of such commands is not permitted in the SAS session[3].

This scripting component is central to having an efficient, reliable, automated and repeatable process for packaging (and deploying) a release. A sensible fraction of time should be dedicated to addressing this area of the project. Scripting is not a SAS skill and should be resourced carefully.

Metadata

Unfortunately this is where our potential for scripting and automation ends. In SAS 9.1.3 we must use the GUI driven BI Manager to perform this task. The metadata content that we need to export is, as defined in the previous section i.e. all developer created metadata artifacts. Ideally two metadata exports / imports would be required…one for the Custom Tree…one for the BIP Tree giving up two .spk files. Our approach to using the BI Manager follows the recommendations in Section 5.

6.4.8. STORING A FULL RELEASE

Let us assume that we have packaged up our release from the Test environment. This is a critical stage in the project; it represents a significant event in the project and is similar to placing a tag in an SCCS. We need to record the state of system at this point is time i.e. where do we archive our release packages?

The release repository is an area on the file system that is used to store full releases. We have already discussed the need for the file system to store content related to partial releases. The root that we suggested here was:

```
\\<ServerName>\RM\ReleaseNotes\
```

The release repository is stored here:

```
\\<ServerName>\RM\ReleaseRepository\Release_X.Y\
```

Where X.Y is the release number. What goes under the `Release_X.Y` folder is project specific. For example your SAS architecture may have a data server, a web server and a metadata server and your packaging scripts may reflect this setup

[3] It is not recommended to turn on the `allowxcmd` functionality due to the inherent security risk such an option poses.

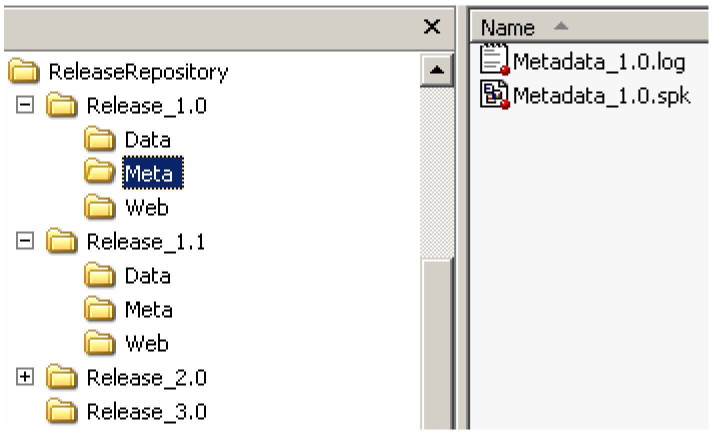


Figure 11: Example representation of the Release Repository on Windows OS.

It goes without saying that the area under `\\<ServerName>\RM` should be backed up and periodically checked to ensure there is sufficient disk space.

There is no reason why the packaging scripts can not transfer contents to the Release Repository in an automated fashion i.e. the scripts are kicked off in the Test environment and a few moments later the folders under `Release_1.0` are populated with the correct content[4].

6.4.9. DEPLOYING A FULL RELEASE

The first step in any deployment is backing up the target environment. A metadata backup should always be performed using the out-of-the-box SAS metadata backup macro `%omabakup`. Areas on the Production file system that we are going to modify can be backed-up by running the packaging scripts that we used to package the Test environment in the first place! The resulting metadata and zip files can be archived on the file system should anything go wrong with the deployment.

Actually deploying a release should be straightforward. We need a set of scripts to unpack and deploy text and data files to the correct locations on the file system i.e. a set of scripts which do the reverse of what they did to package in the first place. We reference the content to deploy from the Release Repository.

The metadata should be imported using the recommendations in Section 5. All metadata objects that we are replacing should be deleted from the target environment before the import.

Typically security varies across environments and with the number of areas that we are likely to touch with a deployment we need to be careful that we leave the environment in a secure fashion. For example our scripts may need to modify OS security both before and after they modify content. Be sure to check that the end state of the Production environment is consistent with the security model for that environment.

6.4.10. VALIDATING A FULL RELEASE

Following the deployment we need to confirm that what we have deployed is correct and works. This is where our investment in building a test suite and our practice with regression testing the Test environment on a regular basis pays off. We simply run our regression test suite against the Production environment. Should all tests pass then the deployment of the release is confirmed as successfully and the system can being operational service.

[4] Actually in some environments this can be a tricky problem because communication between servers (e.g. sftp / ssh) has been disabled in batch mode. In this instance SAS/Connect is an option if this is installed.

6.5. DEFECT FIXES TO THE PRODUCTION ENVIRONMENT

6.5.1. REQUIRED ENVIRONMENTS

The first act to be done after a major deployment into the Production system is to propagate the same code-base backwards into the Development environment.

Remember that the Dev system is essentially ‘dirty’ with open access. It is only through this backwards propagation that we can be confident that the code-base is identical through Dev / Test / Prod. The code-base should be taken from the Release Repository and the deployment and validation techniques that we have previously discussed should be used.

At this point the Dev and Test ENVs should be isolated with the sole remit of supporting fixes to the Production system that may occur through operational use. The Dev / Test ENVs are like branches in an SCCS – they contain the code-base from Production and are stable environments to address and test any Production defects. Our rigorous approach to testing should hopefully result in a zero defect release…but we can never be sure. We now have the following setup:

Environment	SAS Level	Purpose
Production	Lev 0	Supports live instance of Release X.Y
Test	Lev 2	Provides a testing environment for defect fixes to Release X.Y in production
Development	Lev 1	Provides a development environment for defect fixes to Release X.Y in production

Table 4: Required SAS environments to support a Production release.

Clearly, under this configuration, these environments can not be used for any new developments – only operational defect fixes!

6.5.2. PROMOTION PATH FOR DEFECT FIXES

For a major release, the promotion path consists of a combination of partial releases to Test with a final full release to Production. Under the scenario where we have a defect in Production we should be looking to change as little as possible in this environment to address the issue. Issuing a full release to address a single defect represents too much change, which equates to, too much risk. Therefore defect fixes should always be partial releases through the entire lifecycle of the defect fix and the approach to partial releases that we discussed above should be used.

Let us assume that over a five week period we apply five defect fixes to Production using the promotion path discussed above. After the fifth week, where is the definitive copy of the production code-base held…other than in Production? What would happen in the event of a disaster recovery scenario?

Although we have discussed the partial application of defect fix patches we must keep the Release Repository up-to-date with our code-bases – the Release Repository represents the centralized store of *all* our releases. If Release 1.0 is in Prod and we apply a defect fix, this partial application increases the release number to 1.1. We must have a full copy of the Release 1.1 in the release repository. The steps below indicate how we do this:

#	Task
1	Defect fix developed and unit tested in Lev 1 (Dev)
2	CMA performs partial promotion to Lev 2 (Test)
3	Developer unit tests defect fix in Lev 2 (Test)
4	CMA packages Lev 2 (Test) environment and stores in the release repository as a minor release X.Y+1[5]

[5] We do not package the Production environment in Step 4 to minimize interference with an operational service.

#	Task
5	CMA performs partial promotion to Lev 0 (Prod)

Table 5: Summary of the defect fix application process.

So even through the promotion process is partial the whole way through, the Release Repository stores a full release of the system – this is a minor release which includes the defect fix. The RMS is designed to capture this sort of activity.

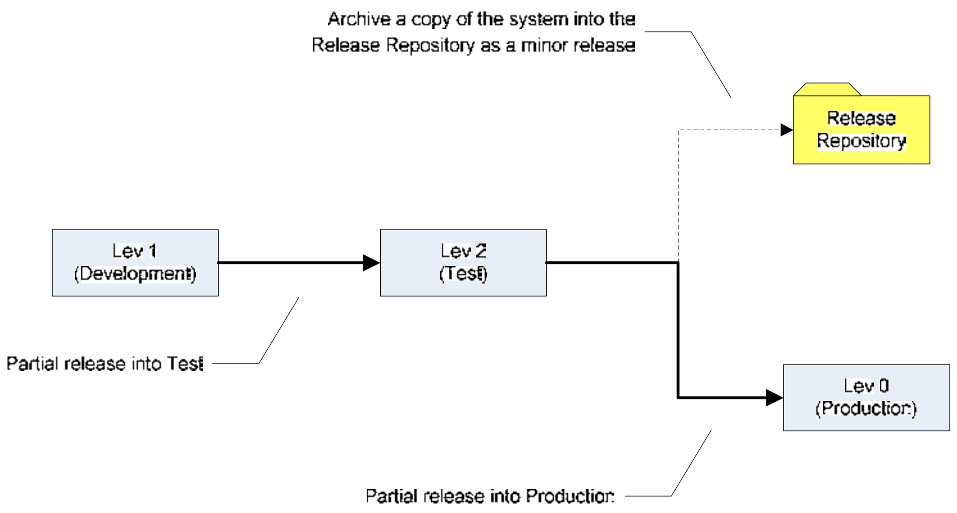

Archive a copy of the system into the Release Repository as a minor release
Release Repository
Lev 1 (Development)
Lev 2 (Test)
Partial release into Test
Lev 0 (Production)
Partial release into Production

Figure 12: Promotion path for defect fixes to the production environment.

6.5.3. VALIDATING A DEFECT FIX

Unit and system testing a defect fix during the development and testing lifecycle goes without saying. Testing a defect fix is Production is likely to be another matter, however, since this is a live service. If it is possible to test a defect fix in production then by all means this opportunity should be taken but this is a luxury that invariably will not occur.

6.6. DEVELOPING FOR CONCURRENT RELEASES

6.6.1. REQUIRED ENVIRONMENTS

So far we have discussed the deployment of the first major release of the system (1.0) and maintaining this by applying defect fixes to give us minor releases (1.1, 1.2,…,1.X). The next major release of the system will be Release 2.0. The PM and RM have discussed the new deliverables for Release 2.0 and have communicated the requirements to the development team…in which environment should they develop for Release 2.0? Clearly they can not use Dev (Lev 1) / Test (Lev 2) since they are required to support defects of Release 1.0.

In fact, we need two more instances of the software to support the development and testing of Release 2.0. If we have n concurrent development streams we require *2n+1* instances of the SAS system. These additional instances can be on physically separate servers or can be logically separated from existing environments by instantiating them as SAS levels. However you configure the additional SAS instances you must be confident that you have enough system resources to support them. An example configuration for two concurrent streams is illustrated below:

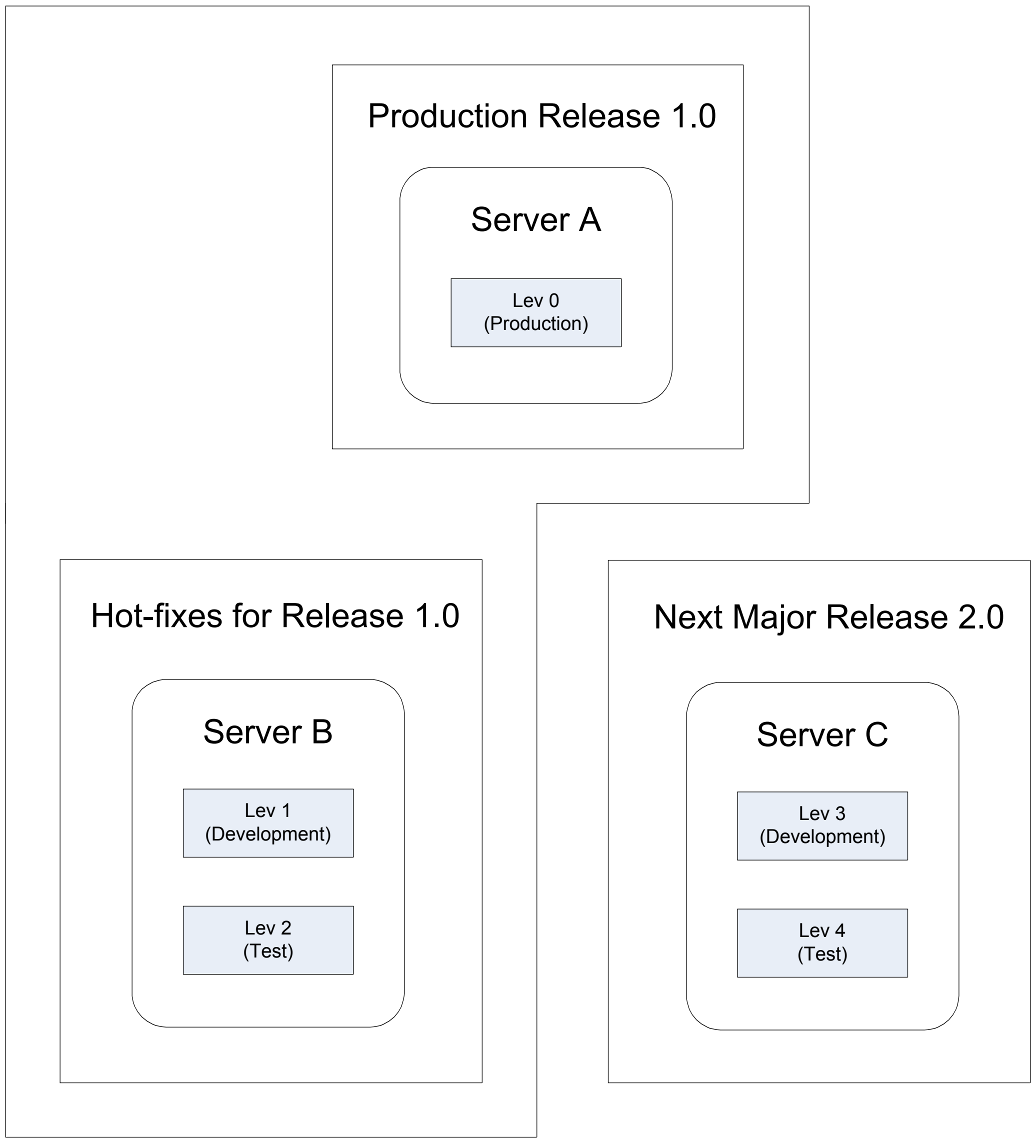


Figure 13: Example server configuration to support concurrent development branches.

In this example, Server C has been setup to support Release 2.0; it has two logical SAS levels, Lev 3 and Lev 4 to support Dev and Test respectively. Since this is a separate server instance we could have called these levels Lev 1 and Lev 2 (just like on Server B). However, it is simple too confusing to have SAS levels called the same thing, but for different purposes, in the same system architecture. Equally we could have put Lev 3 and Lev 4 on Server B if the hardware can support it.

Assuming we have the server configuration above, then the first step in the development process is to baseline Lev 3 and Lev 4 with the current operational release of the code-base so that we can build on top of it. In this model, as soon as have deployed Release 1.0 into Production, we need to deploy the code-base backwards into Lev 1 (for defect resolution) and into Lev 3 and Lev 4 for development of the next release. Therefore, the issue of a major release now constitutes four deployments.

6.6.2. PROMOTION PATH FOR DEFECT FIXES REVISITED

What does having another development stream (i.e. another development branch in the language of SCCS) do to the promotion path for defect fixes. Typically defect fixes will need to be applied in both development streams. If we have a defect in Release 1.0 then clearly it needs to be fixed using our defect fix environments. However, the development stream for Release 2.0 was built from the Release 1.0 code-base so it is likely that the bug exists in this stream too. Defect fixes should be applied in all affected development branches. It is the responsibility of the RM to ensure that all bug fixes are synchronized across the development streams and the RMS is designed to help the RM track this. Table 5 now become:

#	Task
1	Defect fix developed and unit tested in Lev 1 (Dev)
2	CMA performs partial promotion to Lev 2 (Test)
3	Developer unit tests defect fix in Lev 2 (Test)
4	CMA packages Lev 2 (Test) environment and stores in the release repository as a minor release X.Y+1
5	CMA performs partial promotion to Lev 0 (Prod)
6	Defect fix transferred to Lev 3 (Dev) and unit tested
7	CMA performs partial promotion to Lev 4 (Test)
8	Developer unit tests defect fix in Lev 4 (Test)

Table 6: Summary of the defect fix application process with two concurrent streams.

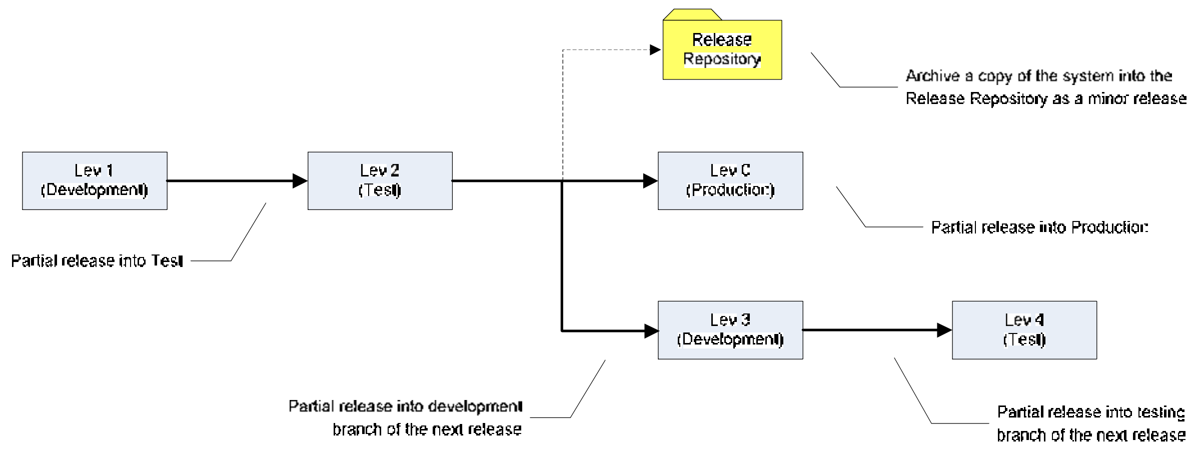

Release Repository
Archive a copy of the system into the Release Repository as a minor release
Lev 1 (Development)
Lev 2 (Test)
Lev C (Production)
Partial release into Production
Partial release into Test
Lev 3 (Development)
Lev 4 (Test)
Partial release into development branch of the next release
Partial release into testing branch of the next release

Figure 14: Promotion path for defect fixes with two concurrent streams.

6.6.3. PROMOTION PATH FOR THE NEXT FULL RELEASE

The promotion path for Release 2.0 is partial between Lev 3 and Lev 4 and then full into Lev 0 Production as per the techniques in Section 6.4. Of course, once we have deployed this release, we need to provide a defect fix environment for 2.0 and an environment to support development for release 3.0. Our deployment pattern becomes:

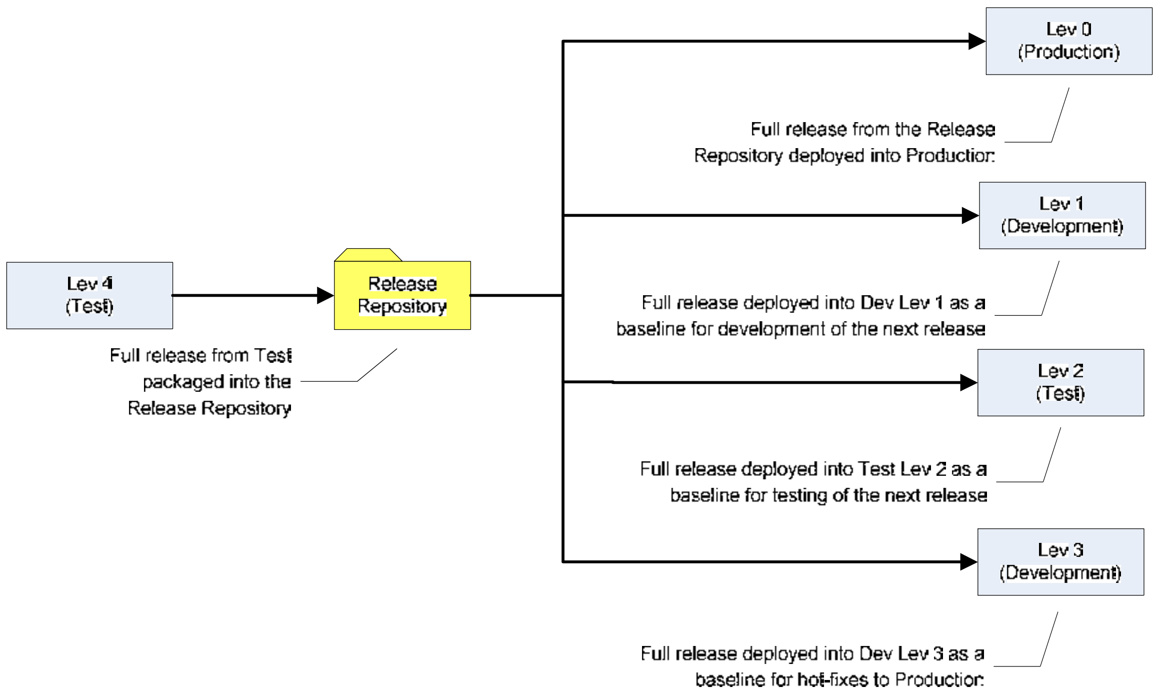

Lev 0 (Production)
Full release from the Release Repository deployed into Production
Lev 1 (Development)
Full release deployed into Dev Lev 1 as a baseline for development of the next release
Lev 4 (Test)
Release Repository
Full release from Test packaged into the Release Repository
Lev 2 (Test)
Full release deployed into Test Lev 2 as a baseline for testing of the next release
Lev 3 (Development)
Full release deployed into Dev Lev 3 as a baseline for hot-fixes to Production

Figure 15: Promotion path for a major release with two concurrent streams.

Server C becomes the environment to development for Release 3.0 and Server B the defect fix environment:

The idea is that as we deploy major releases Server A and Server C flip-flop between being the major development environment and being the defect fix environment.

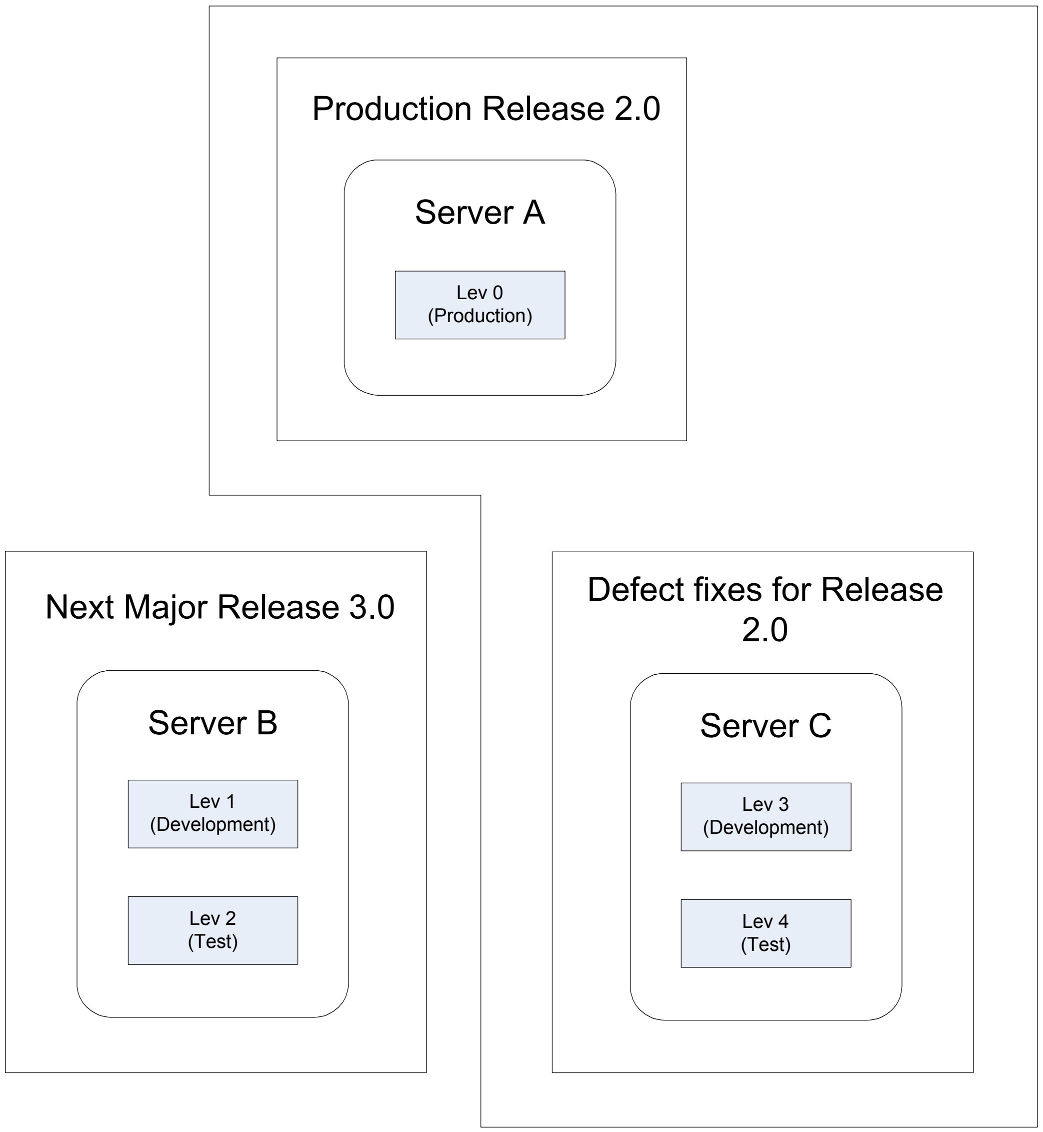


Figure 16: Server B and C in Figure 13 have swapped over 'flip-flopped' so that Server B is used for Dev and Test of the next major release and Server C becomes the defect fix environment.

6.6.4. VALIDATING SUCCESSIVE FULL RELEASES

Whenever a full release is deployed to the Production system the regression test suite should run to confirm the deployed functionality.

However, there is additional layer of validation that can be performed once we have more than one release in the release repository. We can compare releases against one another, list the differences between them and then validate that the files that have changed are consistent with the files that are supposed to have changed, as recorded in the Release Notes for a release as listed on the RMS. This validation step would occur after a release had been placed in the repository, but before the actual deployment.

In order to be an efficient and repeatable process this validation step would have to an executable script or program. Certainly in UNIX this is straightforward to write because of inbuilt OS commands to support the comparison of files (`dircmp` compares all files in a directory). If you use OS commands for you comparison then you will not be able to compare the contents of SAS data sets reliably (since every SAS data sets stores metadata as part of the .sas7bdat file such as the date and time).

We do not suggest comparing SAS metadata files since these can only be intelligently read by the SAS metadata server.

6.7. COMPLEX AND AD-HOC CHANGES

The SAS solution has a complex architecture spanning a number of technologies and third-party products (e.g. WebSphere, Xythos, PostgresSQL). It is impossible to account for all possible changes within one standardized approach. The premise of our model is that most changes are text, data and metadata based and can be captured as part of the full release cycle. There will, however, be exceptions to this rule. Examples are configuration files (e.g. sasv9.cfg) and metadata security model changes. Configuration files will be different per environment and it is likely that the security model will be too. In these types of scenarios, changes should be made using a partial release for the lifecycle of the entire change. For example, a release note is written detailing how to change a configuration file and this RN is executed through the test and production environments – the configuration file is left unaffected by the full release cycle and should never be overwritten as part of this process.

Another good example of complex changes is modifications to the data model. Typically the versioning of a data model would be done within a data modeling tool such as ERwin [16]. However, the complexity of change comes when we have to modify physical tables in the system that contains large volumes data. Changes to the structure of physical tables (and there corresponding metadata component) should be done programmatically through SAS code files. The release process would move all code and metadata across the environments and the physical update to data tables would then to be performed as an ad-hoc part of the release cycle. Clearly the modification of live data is a risky business and measures should be taken to ensure that data are backed up before any updates take place. An additional consideration is that, for an operational system, it is likely to require an outage of the live service to perform a data update.

Complex changes will need to be handled on a case-by-case basis and flagged by the Release Manager accordingly. The RMS is designed to track all changes irrespective of complexity and this would therefore remain the definitive guide to the state of our environments.

6.8. BACKING OUT CHANGES

Changes occur in one of two forms, either a partial or full release. In both instances, part of the release management process is to back up the areas that we are about to overwrite. For a partial release, text and data files can be restored from the areas that they are backed-up, as indicated in the Release Notes. Once a partial restore has been performed, those units that have been restored should be unit tested.

In the event that a deployment of a full release to the Production environment is unsuccessful then we must restore the system to the state prior to the deployment. Since the first step of our deployment process is to backup the Production environment, we have the necessary files.

Metadata can be restored using `%omabackup`. Text files can be restored using our deployment scripts to un-package our backed-up files and restore them to where they came from before we overwrote them with our failed deployment. Once we have restored the backup, our regression test suite should run to confirm the restoration process.

7. CASE STUDY: CONFIGURATION MANAGEMENT FOR A MAJOR FINANCIAL INSTITUTION

7.1. BACKGROUND

A major UK bank is using the SAS Credit Risk Management Solution [17] to meet Basel II regulatory requirements [18]. This is a high profile operational system with minimal opportunities for downtime and penalties for failure to submit results to the Financial Services Authority [18] in a timely fashion.

The project has been years in development and has involved up to twenty SAS developers collaborating across multiple sites in the UK with an off-shore component too.

In the summer of 2008 we were tasked with addressing two major problems:

Deployment Timescales

The first issue, that we were asked to address, was that it was taking too long to deploy new pieces of functionality to other environments. In particular, in order to deploy new components to the live system meant that it had to be taken offline for days while the work was taken!

Project Audit

The second issue was that a project audit was undertaken to provide confidence to the business that we were in control of the configuration management aspect of the project. The outcome of the audit was the driver for this piece of work.

7.2. USE OF A SOURCE CODE CONTROL SYSTEM

A previous attempt had been made to get a third-party SCCS to integrate with the SAS system. This was not successful for a number of reasons:

- None of the SAS team knew how to use the tool.
- For political reasons the SAS team were not allowed to administer the tool.
- Sufficient time was not allocated for the integration of the tool on the project.
- We were working in a virtualized environment and experienced issues in getting the tool to communicate with virtualized servers.

The short development timescales that we were given to address the issues raised in the project audit meant that it was not possible to resolve the above in time to meet the audit requirements.

7.3. BESPOKE APPROACH TO VERSION CONTROL AND RELEASE MANAGEMENT

Given the tight timescales we decide to develop our own configuration management system based on bespoke UNIX scripts with a view to delivering the following benefits:

7.3.1. PROPOSED BENEFITS OF BESPOKE IMPLEMENTATION

- A “Release Repository” to hold all releases of the code base.
- A “Code Repository” to hold all versions of all artifacts.
- Compare Releases to detect changes before deployment. These changes must marry with developer / release documentation.
- Releases to be signed-off before deployment.
- All artifacts in a Release to maintain a Release number.
- Artifacts to maintain a version number.
- Previous versions of artifacts to be archived.
- Functionality to enable the comparison of an artefact with a previous version of that artifact.
- Functionality to deploy a Release in as automated a manner as possible.

7.4. CONFIGURATION MANAGEMENT ARCHITECTURE

Our configuration management architecture was based on the following building blocks:

- Release Repository
- Release Management Tracking
- Code Repository
- Version Control Scripts

- Release Management Scripts

We now discuss each of these in turn.

7.4.1. RELEASE REPOSITORY

We moved to a released based approach to promoting our software into new environments. On a scheduled basis (Tuesdays and Thursdays) we issued new, *full* releases of the SAS solution to non-Development environments. The advantage of this approach is that it was a repeatable, reliable, standardized and efficient process (using the scripts we describe shortly) to packaging and deploying the code-base. Piecemeal changes, which are high risk, were restricted to development environments and critical production defects.

The new releases were taken from the development environment and included all code and metadata. The Release was packaged (i.e. tar'ed and zipped) and placed in the Release Repository much as described in Section 6.4.8.

7.4.2. RELEASE MANAGEMENT TRACKING

There were three critical components to successful release management tracking. By far, the most important was the appointment of a Release Manager to own the whole release management process. We used the Release Management Spreadsheet and Release Notes as described elsewhere in this document to record the process. The RM was responsible and *accountable* for the ensuring a timely delivery of successful Releases.

7.4.3. CODE REPOSITORY

The Code Repository (CR) is an area on disk that we created to support the versioning of units within the system. For example if a developer updates the contents of the file `class.sas` there will be at least two versions of this file. The CR is where we would store all versions of this file (e.g. `class.sas.v1`, `class.sas.v2`). In the absence of an SCCS, this was the “database” of our version history.

7.4.4. VERSION CONTROL SCRIPTS

At the heart of the configuration management architecture were a number of bespoke UNIX functions that have been explicitly developed to meet the business benefits outlined in Section 7.3.1. These were written in the KORN shell and designed to be used like any standard UNIX command. These scripts do not apply to SAS metadata in any way. An overview of these scripts is given below:

configmgtprofile

Installs the configuration management UNIX functions for a user.

checkin

Used by a developer to version a code unit. For simplicity only `.sas, .ksh` and `.sh` are recognized by the function. A copy of the unit is place in the Code Repository and the header on the file is automatically updated with the user, date/time and a mandatory comment which must be provided on the command line when the *checkin* function is called:

```
    checkin -f calculate.sas -c "This is a comment"
```

The option –f is to specify the filename. The option –c is to specify a comment. Both options are mandatory.

checkin provides a mandatory compare with the previous version of the file - so a developer must understand the changes that s/he has made to a file before committing the file to the CR. A successful *checkin* will issue a report that looks something like this:

```
Check-in Report...
```

```
-------------------------------
USER: tp6dso4
DATETIME: 20080806_200303
CHECK-IN FILE: /wload/tp6d/home/tp6dso4/Lev1/calculate.sas
CHECK-IN COMMENT: CR 567
CREATING VERSION: 1
-------------------------------
Check-in completed successfully
```

promote

In our setup we had a number of development environments. Since Dev is a 'dirty' environment we created another Dev environment that was used for SAS unit testing - a sort of dev-test ENV - this was in addition to a system test environment. We moved units to dev-test as they were completed and wanted a way of ensuring that only checked-in units were promoted up. With this in mind we developed the promote function as an administrative utility to move files around - that only completes successfully if the file to be moved is checked-in.

```
      promote -f calculate.sas –l Lev2
```

The option –f is to specify the filename. The option –l is to specify the SAS level that you want to *promote* to. A successful promotion returns a report:

```
Promotion Report...
-------------------------------
USER: tp6dso4
DATETIME: 20080806_200757
PROMOTING FROM: /wload/tp6d/home/tp6dso4/Lev1/calculate.sas
PROMOTING TO:   /wload/tp6d/home/tp6dso4/Lev2/calculate.sas
PROMOTING VERSION: 1
-------------------------------
Promotion completed successfully
```

checkcodecut

Before an environment is 'cut' for packaging and deployment. We 'check the code cut' to ensure that every file that we package is versioned and stored on in the CR. *checkcodecut* is a safety net in the CM process.

```
   checkcodecut -l Lev2 -f cntrlfile
```

The –l option is to specify the level to check i.e. the level you are about to package; the –f option take a control file as a parameter. The `cntrlfile` tells the `checkcodecut` function which files and directories to check to see if they have all been checked-in.

checkinbatch

Should checkcodecut highlight a number of files that have not been checked-in, `checkinbatch` is an administrative function to batch check-in, all unchecked-in files.

```
      checkinbatch -f <checkcodecut log file>
```

The –f option takes the log file from the `checkcodecut` function as a parameter.

7.4.5. PROCESS FLOW FOR VERSION CONTROL SCRIPTS

A high level process flow for the version control scripts is given below.

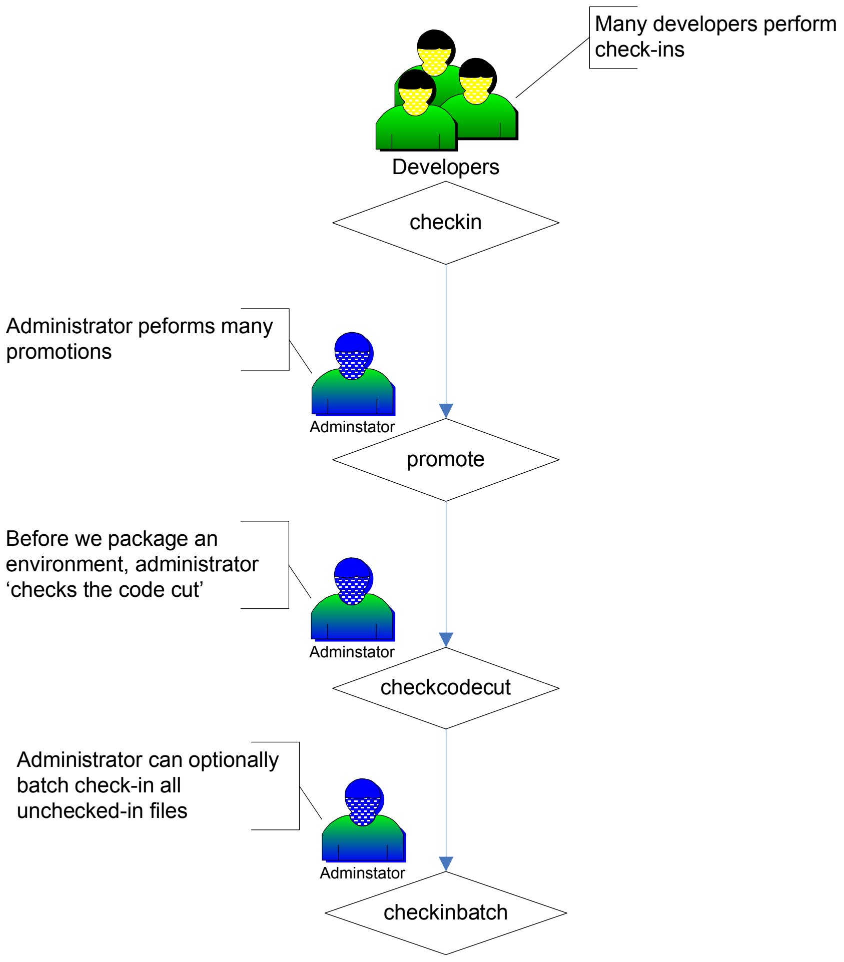

Many developers perform check-ins
Developers
checkin
Administrator peforms many promotions
Adminstator
promote
Before we package an environment, administrator 'checks the code cut'
Adminstator
checkcodecut
Administrator can optionally batch check-in all unchecked-in files
Adminstator
checkinbatch

Figure 17: Process flow for the version control scripts.

7.4.6. RELEASE MANAGEMENT SCRIPTS

Once a Release was completed and we were happy that all code units had been checked-in, another set of UNIX scripts was called to package (tar and gzip) all files and directories for the release and transfer the packages to the Release Repository. Yet another set of scripts worked on the Release files themselves – again none of these were called against SAS metadata.

updatereleaseheader

Every unit in the Release had a code header – this script executed against every file in the current release folder and modified the code header to insert the current release number i.e. every file was tagged as belonging to the current release

```
updatereleaseheader
```

The `updatereleaseheader` function dynamically determined the current release number and applied this to all files in the release.

releasecontents

```
        releasecontents
```

Once we had stamped a Release with the current release number we are ready to produce the Release Log. The Release Log is an inventory of the units in the Release (excluding data and metadata). Although it is not necessary to produce the Release Log to deploy the Release, the Release Log formed part of our quality control procedures and was always executed and reviewed.

diffreleases

Within the RR, the units between any two arbitrary Releases can be compared to see where units differ. The units which have changed in a Release should tie exactly to those described in the Release Notes for this Release and those documented in the Release Management Spreadsheet. This Release diff was a powerful validation function that confirmed to the Release Manager that the changes in a Release are those which are expected.

```
        diffreleases Release_2.0 Release_1.0
```

The script was called with the two releases to be compared as command line parameters.

Once the RM was happy with the Release diff he would sign-off the Release. Once the metadata for the Release was placed in the Release folder the RM would signal that the Release was ready for deployment to another environment.

Deployment to Target Environment

The final step was to execute a final set of scripts to backup the target environment, and then extract the current release from the Release Repository and deploy this to the target. Once this had been done, smoke tests were performed to provide confidence that the deployment had been successful.

7.4.7. PROCESS FLOW FOR RELEASE MANAGEMENT SCRIPTS

A high level process flow for the release management scripts is given below.

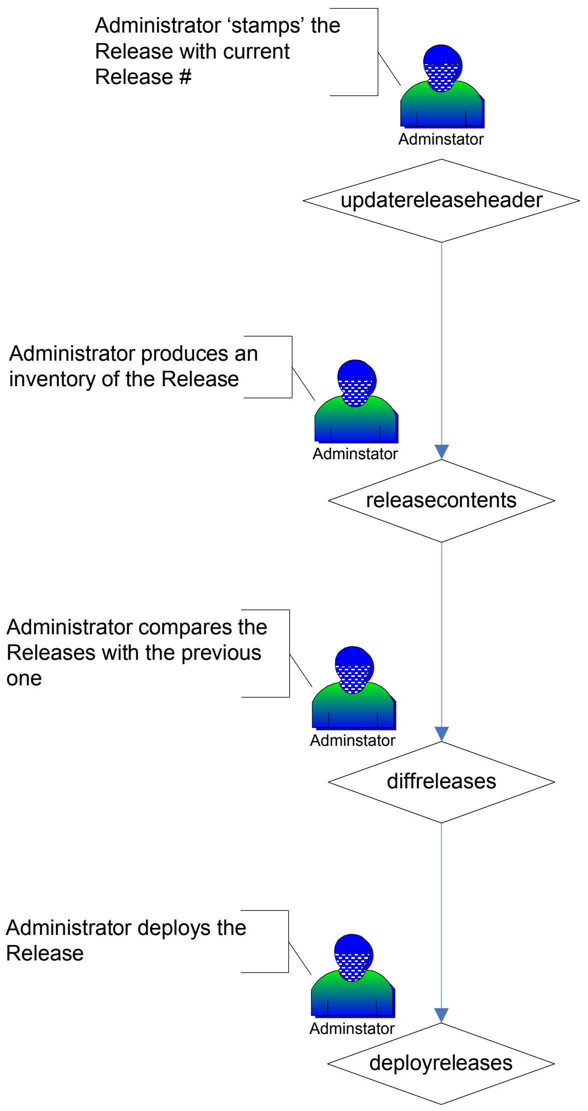

Administrator 'stamps' the Release with current Release #
Adminstator
updatereleaseheader
Administrator produces an inventory of the Release
Adminstator
releasecontents
Administrator compares the Releases with the previous one
Adminstator
diffreleases
Administrator deploys the Release
Adminstator
deployreleases

Figure 18: Process flow for the release management scripts.

7.4.8. UN-VERSIONED OBJECTS

For the sake of simplicity there were three types of object that we did not version: metadata, data and configuration files. We have discussed these components elsewhere in this document so suffice is to say here, that given the timescales we were working with, we either did not know how to version these objects or did not think they provided sufficient risk to the project to be explicitly addressed.

In the case of metadata, whenever we were preparing a release we performed full metadata exports / imports of everything under the BIP Tree and the DI Custom Tree in the SAS Management Console using the BI Manager. We followed the guidelines given in Section 5.

7.4.9. ENVIRONMENTAL CONSIDERATIONS

In order to support development of concurrent releases and various types of testing eight SAS environment were operational – each of which needed to be managed and controlled! The deployment path through these environments is too project specific for us to warrant detailing it here. Instead we have abstracted our recommendations, based on our experiences, to Section 6. The key point to remember is that code branching in the traditional sense of a SCCS is not possible in SAS due to the shared nature of the metadata server. A code branch is SAS9 effectively requires another SAS level and therefore – when developing for more than one release at the same time – many SAS levels are required.

Our strict release management tracking also helped up to makes sure that defects were fixed in all the correct SAS levels. On a number of occasions previously, defects had been fixed in the Production environment – only to be re-introduced in the next release of the system because the defect had not been fixed in the development stream for the next release. Our live service had two (Dev and Test) fix-on-fail (FOF) environments to support live defects. The figure below shows how we ensured that defects were developed, tested and deployed to Production but were equally retrofitted to the development stream for the next release:

Schematically, our defect fix process looked like this:

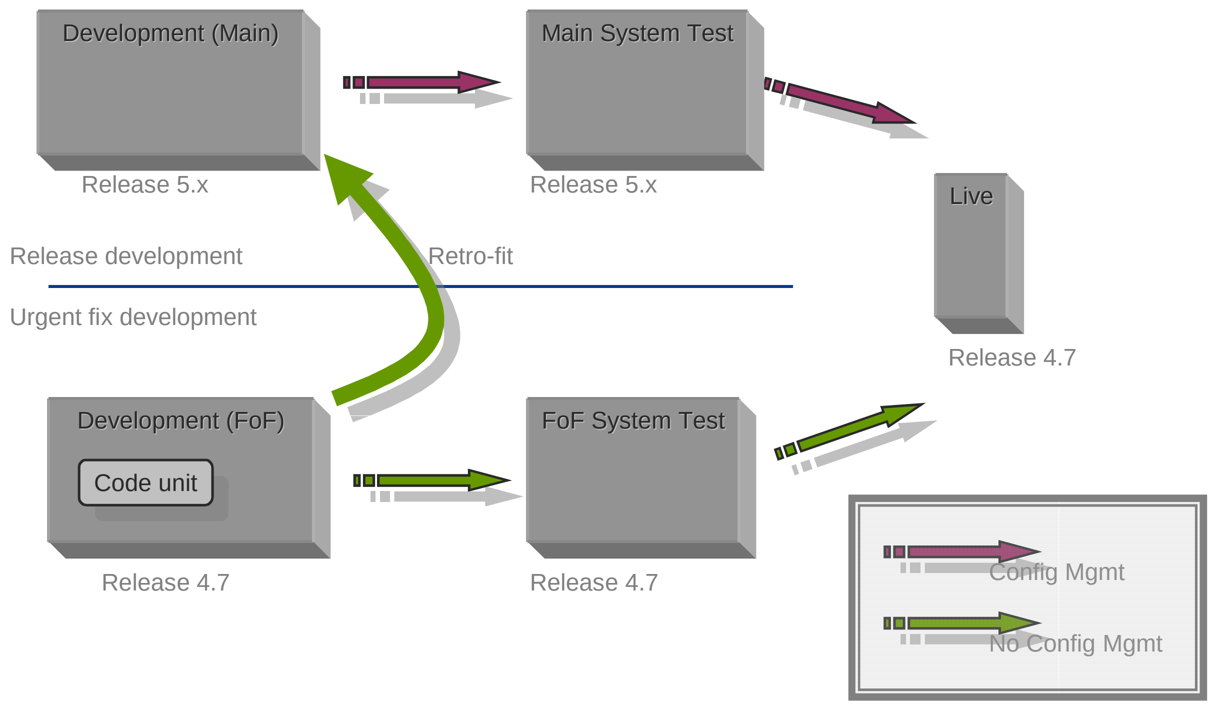


Figure 19: Applying defect fixes to all of the necessary code streams.

7.4.10. LESSONS LEARNT

We succeeded in building the system described above in time to meet the re-audit and had a profound impact of the quality control of the project. Although our approach was rather simplistic and did not explicitly account for all eventualities, we focused on the areas which we thought were important and could address in relatively short timescales. Perhaps the most significant measure of our success is that deployment of releases that were taking days was reduced to hours – much to the delight of the business.

There were a number of areas that we thought we would approach differently in the future:

- Before commencing the project, have an approach for configuration management i.e. a CM Plan and discuss this with the customer prior to beginning any implementation.
- Where possible, use a third-party SCCS to achieve much of what we achieved (and more) through our bespoke UNIX scripts. Shell scripting is not a SAS skill and the development and maintenance of such scripts can be difficult without the right skill set.
- Whatever approach is taken, the development and management team must be educated and adhere to the processed put in place. In our case, substantial documentation was generated to support our approach and this was supported by formal training sessions for the development team.
- Do not allow cross platform development. On a number of projects now we have experienced problems where developers have used a third-party tool to take files from a UNIX server to a windows client, edit the file in Windows and then transfer them back to UNIX. Invariably the file ends up back in UNIX as a Windows file and then does not respond well to UNIX commands being run against it.

- Although we had a checkin function, we did not have a corresponding checkout. Although this was suggested, time did not allow for the completion of this script. The checkout script would notify a user if a file was already checked-out…thereby warning a user of the risks of concurrent development.
- The Release Repository and the Code Repository grew in size very quickly and, at first we did not account for the disk space required for these repositories.
- The way that the current process works is that the code that is being versioned is altered – the header of the file is updated. There are disadvantages to this approach – in order to update the header we must know, beforehand, what type of file we are updating so that we have apply the necessary comment marker in the header. In addition, it means that we can only version certain file types. A better approach may have been to maintain, for each file, a separate history file. A history script could then query the history file. This is a vastly more generic approach, easier to maintain, less intrusive and lower risk. In fact, in the first iterations of the configuration management scripts – the file that was being updated was done so incorrectly thereby causing the file to produce an error when in operation.

8. CONCLUSION

This document discusses the configuration management for SAS projects, focusing largely on the release management process.

Every software project is different and, although we provide a framework for implementation, this must be customized to fit in with the constraints of the engagement at hand. Our methodology is based on a number of key features and where possible we recommend these practices are always adopted.

Roles

Appoint a Release Manger to control iterations of the development lifecycle and be accountable for what is delivered into a production environment.

Appoint a technical administrator for the configuration management process as a whole

Documentation

Document all changes in Release Note and summaries these on Release Management Spreadsheet.

Environments

For each development stream (branch / release) have at least two instances of the SAS system to support development and testing. To make it easy to move content between environments ensure that the development team are not hard-coding unnecessary variables into files.

Testing

All developers should write unit tests for there work and, where possible, these should be executable programmes. Test ruthlessly – fully regression test the Test environment every night. Always regression test a full release.

Deployments

For full releases, perform partial release from Development into Test. Once the full release has been delivered into Test through a sequence of partial releases, package the test environment and deploy this as a full release into Production. For defects and defect fixes, the release process is partial for the lifecycle of the fix. If you are applying defect fixes be careful to deploy the defect fix to all development branches.

Automation

Automate as much as possible to make the process efficient, repeatable and reliable. Using scripting techniques to package, test and deploy releases.

Backup

Backup and archive everything. Use the Release Repository to archive major and minor releases.

9. GLOSSARY

SAS	Business Intelligence software
BI	Business Intelligence
ISO	International Standards Organization
DI	Data Integration
CMM	Common Metadata Model
VC	Version Control
SCCS	Source Code Control System
CMA	Configuration Management Administrator
RM	Release Manager
RN	Release Note
PM	Project Manager
RMS	Release Management Spreadsheet
OS	Operating System
FUTS	Framework for unit testing SAS
MC	Management Console
SPDS	Scalable Performance Data Server
ENV	Environment
ACTs	Access Control Templates
RAM	Random Access Memory
CR	Code Repository
FOF	Fix-On-Fail

10. REFERENCES

1. *Software Change, Configuration and Release Management* course, given by *Learning Tree International*

http://www.learningtree.co.uk/courses/uk342.htm

2. IEEE Std-828-2005 Standard for Software Configuration Management Plans

   http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=32241

3. Example software configuration management plan

   www.osd.noaa.gov/class/docs/1001_Config_Management_Plan.doc

4. Common Warehouse Metamodel

   http://www.omg.org/technology/documents/modeling_spec_catalog.htm

5. Best Practices for SAS®9 Metadata Server Change Control

   http://support.sas.com/resources/papers/MetadataServerchngmgmt.pdf

6. SAS 9.1.3 Intelligence Platform: System Administration Guide

   http://support.sas.com/documentation/configuration/bisag.pdf

7. SAS 9.1.3 Intelligence Platform: Web Application Administration Guide, Second Edition P. 62

   http://support.sas.com/documentation/configuration/biwaag.pdf

8. Metadata export / import process in SAS 9.2

   http://support.sas.com/resources/papers/sgf2008/migratemetadata.pdf

9. Extreme Programming

   http://www.xprogramming.com/

10. The Rational Unified Process

    http://en.wikipedia.org/wiki/Rational_Unified_Process

11. Scrum

    http://en.wikipedia.org/wiki/Scrum_(development)

12. Iterative approach to software development

    http://en.wikipedia.org/wiki/Iterative_and_incremental_development

13. Waterfall development technique

    http://en.wikipedia.org/wiki/Waterfall_model

14. Agile development techniques

http://en.wikipedia.org/wiki/Agile_software_development

15. Framework for unit testing SAS

    http://www.thotwave.com/products/futs.jsp

16. ERwin Data Modeler

    http://www.ca.com/us/products/product.aspx?id=260

17. SAS Credit Risk Management for Banking

    http://www.sas.com/industry/banking/credit/

18. Basel II Accord

    http://www.bis.org/publ/bcbsca.htm

19. Financial Services Authority

    http://www.fsa.gov.uk/

CONTACT INFORMATION

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

Steven O'Donoghue, Principal Consultant
SAS Institute
Wittington House, Henley Road, Medmemham, Marlow, Bucks, SL7 2EB, UK
+44 1628 486933
steven.o'donoghue@suk.sas.com

Andrew Ratcliffe, Managing Director
RSTL
5 Willow Close, Bexley, Kent, GB-DA5 1QY, England
+44 1322 525672
andrew.ratcliffe@rtsl.eu