Software Release Management

Presented to

Lorette Clement-Smith
Instructor, Communications 250
University College of the Fraser Valley

Prepared by

Wim Kerkhoff
CIS Diploma program

April 10, 2002

Transmittal Memorandum

To: Lorette Clement-Smith, Instructor

From: Wim Kerkhoff

Date: April 10, 2002

Subject: Software Release Management

Attached is my completed research report that you requested in January. My chosen topic, Software Release Management (SRM), explains the methods involved in releasing software in a controlled and optimized manner. While reading through the many sources that I found in the library, I realized that most software is of poor quality, is produced over budget, and is rarely on time. Many people do not realize this. I am learning practical ways of dealing with these common problems, so that I can help others mature their development processes and become more capable.

This report summarizes software version and revision numbers, release milestones, release tools, build reproduction, and how to simultaneously develop multiple versions of the same product.

I am grateful to my business communications class for helping to clarify the proposal and stimulating me with new ideas. Colleagues at my employer, Merilus, and online SRM professionals were also indispensable in providing examples of handling parallel software development and contributing pointers to other sources.

This paper provided an opportunity for me to formally research this topic. Before I started the course I had already started reading this intriguing topic, so that I could better aid and contribute to development processes. I now know more about how software engineering works in the real world.

Editorial Memorandum

To: Peer editors

From: Wim Kerkhoff

Date: April 10, 2002

Subject: Software Release Management

Thank you for editing my paper and providing constructive feedback.

Most of your suggestions have been implemented by reworking the introduction, conclusion, and the document styles. One of the suggestions was to left align the headings instead of center aligned. This minor change increased the aesthetics of the page layout.

Feedback obtained early in the writing stage alerted me to the fact that most of the audience is not in the Computer Information Systems degree program. Because of this reasonable fact, I was able to change partitions of the report, such that it would be understandable to a greater audience. The appendices and glossary were added for the benefit of those unfamiliar with many of the key terms discussed.

These changes have increased the readability and visual presentation of the paper. I really appreciate the opportunity of obtaining your feedback. Being able to review and proof your drafts also helped me recognize your writing strengths, and apply them to my paper.

Table of Contents

Executive Summary

Purpose of this report

The purpose of this report is to investigate Software Release Management, and how a software development company can efficiently produce timely releases of multiple versions of their applications.

Software Release Management

Many software development companies are not aware of the significance of controlling, documenting, and automating software releases.

Most software development teams are working on multiple versions of the same application: past production releases, a version that the Quality Assurance team is verifying, and a current development version for new features. However, they have poor methods of handling the changes between these versions. For example, program fixes that were made to the Quality Assurance release need to be merged back into the main development tree. If not, then when a new Quality Assurance release is created, the program bug will reappear. In addition to not being able to control changes to released versions, many developers have problems keeping logs and audit trails of all changes. The absence of change and version management is caused by not knowing the basics of software configuration management and how to apply it to everyday problems.

This research shows that the proper use of a versioning system tool is required to keep track of software revisions, versions, branches, changes, and merges of changes between branches. Automatic build reproduction is required to produce high quality and auditable software releases.

Recommendations

Recommendations for Software Release Management include:

1. Using a configuration management tool such as CVS
2. Make effective use of branches and tags for releases
3. Create automated build scripts

Introduction

Problem

"Controlling source code and development artifacts is a critical part of modern software development" (Mikkelsen and Pherigo, 1997). Rarely is software shipped on schedule, on budget, and with the features and stability desired in the original specifications.

Purpose

This report will investigate methods of controlling software releases and the components. To do so, software engineering concepts will be defined, and methods of enforced software source code control will be explored.

Background

Software Configuration Management (SCM) is primarily concerned with managing source code, the blue prints to software programs. In the last decade, Software Configuration Management has received an increasing amount of attention. This increasing attention is due to the size, complexity, and constant change in modern software.

A Software releases is a fully constructed version of a software product which is ready for use by a specific audience. Releases include the distribution media, documentation, and training materials. Distribution mediums include floppy diskettes, CD-ROMs, DVDs, and Internet downloads. Releases are typically directed at end consumers, but can also be for demo, preview, or testing users. Software release management describes methods of assembling the complete release and uniquely identifying its components.

Software development commonly deals with four components: the software programmers, source code they write, the constructed software, and the development environment. Development environments typically consist of the software, hardware, and development tools required to create the software product.

Even when using Software Configuration Management, it is difficult to release a quality release on time (McCarthy, 1995). Because scheduling is harder, a development team will have a much better chance at releasing a quality project on time at a lower cost.

Scope

This report does not cover the entirety of Software Configuration Management. Instead, it deals with Software Release Management, which deals with how to simultaneously work on multiple versions of the same software application in an efficient manner. Issues such as planning, development models, quality assurance, defect management, and standards are not covered.

Organization

Software Release Management deals with four main concepts. Revision and Version Numbers identify changes in each source code file and the final product. Milestones and tags are used to create a snapshot of groups of source files at a point in time. Branching deals with how to development on parallel versions of the same source code. Any software version can be constructed at any time when build reconstruction scripts are used. These subtopics are interdependent, but presented separately.

Sources and Methods

Books from a university library and from the author's personal bookshelf were the primary sources of information. Internet web sites, electronic mailing list archives, email communications with SRM professionals, and online journals were also vital. Online resource centers provided indicators to useful books and industry research by others.

Revision and Version Numbers

Every file in a software management system is given a revision number. When the file is changed, it is automatically incremented. For example, with the Concurrent Versions System (CVS), when a file is first added, it is assigned a revision number of 1.1. Upon the next save, it would be incremented to 1.2. It is not possible to use an arbitrary revision number; these numbers are assigned by the source code management tools, and have special meanings.

The release manager can assign an arbitrary free form version number to a file or group of files. For example, s/he could assign “Version 8.0” to the entire application. S/he could then assign “Version 8.1” to the next version of the program when it becomes available. Behind the scenes, the software revision management system has its own internal revision number, such as 1.31. A release manager is also known as a release engineer, and is responsible for assigning version numbers, creating builds, and creating tags for the major milestones in the software product. Often, the release engineer is a senior developer, the Quality Assurance manager, or somebody who is tasked to perform such work.

The SCM tool ensures that the correct versions are incorporated into the system builds (Leon, 2000). Version management is a critical function of SCM and is the basis on which other functions are built. Version management maintains the storage of multiple intertwined versions of related files. With a good version management system, any image of these versions can be retrieved for inspection, comparison with another image, or to rebuild that version. When changes are made to a version, the SCM tool allows the developer to define a new version.

Milestones and Tags

Because version and revision numbers are hard to remember, people involved with software construction prefer easier ways of referencing particular versions. There are easier ways of grouping files that go into a release (Mikkelsen and Pherigo, 1997).

Without the use of a software configuration management system, developers tend to create their own solution. Typically, program components are stored in a single directory which may in turn contain further sub directories. When the program is released, that directory is backed up. If version 1.0 of WidgetABC is released, then the C:\WidgetABC would get copied to C:\WidgetABC-1.0. Development then resumes in the original C:\WidgetABC directory. This method is inefficient, as it wastes disk space by creating a duplicate of every single file, even though most files have not been changed. To be able to just backup the incremental changes is better.

Most software configuration management systems have a way of adding a tag to file. This tag is associated with a particular revision of the file. For example, revision 1.7 of the file abc.txt could be tagged as “RELEASE_1.0”. At a later date, it would then be trivial to locate and retrieve the state that abc.txt was in when WidgetABC 1.0 was released.

In fact, tags can be applied to any number of files. Tags applied to a file must be unique. The tag RELEASE_1.0 can’t be added to revision 1.9 of abc.txt if revision 1.7 is already marked as RELEASE_1.0.

When a build is done using a SCM system, most release engineers will tag the entire tree, marking which revisions were included in the build. A tag can be added to any revision of a file. This allows one to add revision 1.5 of def.txt to RELEASE_1.1, even though the current version is newer at revision 1.7.

When the files in the repository are tagged for milestones and major builds, then it is easy to retrieve any files that are marked with a specified tag. Once the group of tagged files has been retrieved, then a build using those files can be done. This has many benefits. If a critical data-corruption bug is discovered in WidgetABC 1.0 but the current

version is 2.0, then the files for 1.0 can be retrieved, the bugs fixed, and a patch for the 1.0 bug can be sent to those customers that have not upgraded from version 1.0 to 2.0.

Many software construction schedules define milestones for particular stage points in the development cycle. When these milestones are reached, a tag will be created in the SCM system. As development continues, it will then be possible to compare against this known state to measure progress.

In addition to predefined milestones, developers can of course create any arbitrary tag. For example, if they are fixing a problem with the source code, they might tag the code before as “PRE_BUGFIX” and after as “POST_BUGFIX”.

One of Cisco’s departments, the Multi-service Switching Business Unit (MSSBU), has a very strict process for release management (Jarvis and Hayes, 1999). Instead of milestones, they use checkpoints which they strictly enforce. A Vice President must approve any deviations from the process. This is understandable considering the size of their development team, but is too involved and complicated for a small to medium sized team. Their four important checkpoints must be met before any development can continue: Project Definition Checkpoint, Release Commit Checkpoint, Time to Market First Customer Ship Checkpoint, and Time to Volume First Customer Ship Checkpoint. Respectively, these are milestones for the beginning of development, the end of development, the end of the quality verification cycle, and the beginning of mass production. Because MSSBU primarily deals with computer hardware devices, they have developed a highly detailed and enforced release system.

Branching and Parallel Development

To an extent, development in the software development industry can be compared to parallel development in house construction. When building a house, there are multiple tasks that can be performed at the same. While the plumber is connecting the hot water pipes, the electrician may be completing the wiring. While one programmer is adding a new button to the toolbar, another programmer might be fixing a spelling mistake in a startup window.

However, the changes made by one programmer may be in the same source file that another programmer is working on. In the house construction example, two sub trades may need to work in the same room. One of the two workers would be forced to wait for the other to finish before being able to continue. Thus, this is not an appropriate situation to compare parallel software development with. If a book author emails a copy of the manuscript to her editor to proofread, the editor can make changes and even rewrite portions, while the author may be rewriting the same portion in preparation for adding a another chapter to the book. In this scenario, both are working on the same part of material, but on their own copy. Any changes they make are not reflected in the other person's copy. At some point, somebody will have to open both documents and manually merge them.

SCM tools make it easy to work on parallel copies of the same product. To do so, a developer creates a branch of the main version. In the version management repository, nothing has changed yet. As the developer saves his changes to the parallel copy, or branch, they get marked in the SCM tool as belonging only to the branch. Later on, when the programmer is confident that his changes are good and ready for inclusion in the main copy, he can use the merging functionality of the SCM tool.

If changes were made to the exact same source files in branches that are being merged together, the changes may conflict. For example, Alice may add a word into hello.txt. Bob, while working on a separate branch, might remove a different word from the same line. When Bob merges his branch back into the main copy of hello.txt, his

change will conflict with Alice's change. The SCM tool will alert him to the fact. The two programmers will need to talk until they agree whose change will remain.

It is important that all build scripts, tests, and documentation are kept with the branch, and promoted to the next release as appropriate (Kit, 1995). If the build scripts are not synchronized, it may become impossible to rebuild the next release.

There are powerful software tools that can take care of merging changes between versions (Leon, 2000). However, care must be taken to verify that conflicts were resolved correctly, and that all changes were applied to the destination files. The merging mechanisms in CVS are not very robust (Spitzbarth, 2001). Sometimes, even though a merge appears to be successful, it may fail to apply some of the changes.

Build Reconstruction

Not all software requires compiling, that is, converting human edited and readable text files into ones and zeros that the computer can understand (Mikkelsen and Pherigo, 1997). Sometimes, construction of a build requires combining multiple files into a combined file. In the case of a web site, building might involve adding a common header and footer to the bottom of every page before publishing it. A build script contains the commands that are necessary to assemble the pieces into a final product. By running this build script, the final product is assembled.

In an email conversation, Spitzbarth (2001) describes how he has set up the build and branch system at his company. Every night, a new build of the software is automatically created. A precise name such as "project7_Build5_Mar13_2002" enables anybody to see that the build was from the night of March 13, 2002. If Quality Assurance approves this build and agrees to test it, the build is renamed to something more formal, such as "Project101_Build5".

A build system should be designed to run on a scheduled basis, such as every night, or on demand to include changes. The scripts for the build system should be stored in the SCM system, so that any release can be built at any time.

The product should be built often and regularly, rather then just before shipping (McCarthy, 1995). "The point of this rule is to engage the team in building the product frequently, regularly, throughout the development cycle, with the highest possible quality, and in a public place where all team members can have access to it" (Pg. 109).

This allows everybody to assess the daily status, and see whether everything is on target. Otherwise, not everybody will know how far along development is, and problems that appear closer to shipping time will postpone the shipping time. If somebody adds code that breaks this regular build, everybody will notice. There will be increased pressure to commit high quality changes so that the build can complete and be tested.

Brooks (1995) agrees with how McCarthy describes the build system at Microsoft. In his 20th anniversary version of the book "Mythical Man-Month", he admits that the Waterfall method of software development is wrong. He describes an

Incremental-Build model, where the system is always running. In the beginning of a product, a simple framework is started. The idea is to always have the framework running without any problems. New components get tied into the framework one at a time. First versions will be boring and feature incomplete, but will always run. This has the added benefit of being able start automated and manual testing early in the design and development phases, long before it is feature complete.

Progressive refinement systems are a much better way of developing than the waterfall method, where the design, implementation, and testing stages must be completed consecutively. Projects where an automated building and testing system is not set up do proceed as smoothly as well as projects where building and testing is automated... Rather then doing a lot of up front planning work, design when design is needed, and continually add to and repair the working code when necessary.

Appendices

Appendix A: CVS Change Logs

One of the most common revision control systems is the Concurrent Versions Systems (CVS) software package. CVS is an extension of the Revision Control System (RCS). Below is the complete log for one of the source code files in a web based email application that the author of this paper has worked on. As the name implies, French.pm is a module that handles a French version of the application.

Important portions of the log file below have been highlighted in bold. The original author is 'acme', but since then the user 'wim' made changes on a couple of occasions. For each change, a new revision number is automatically assigned, starting at 1.1. Version numbers have been manually assigned to the file, which the 'symbolic names' header shows. It appears that the 'acme' author tagged it as a development version in April 2001.

Change logs are a way of showing when a file was changed, who changed it, why they changed it, and what they changed.

```
RCS file: /cvsroot/acmemail/sparkle/Acmemail/Lang/French.pm,v
Working file: French.pm
head: 1.3
locks: strict
symbolic names:
      Devel-2001_04_27: 1.1
keyword substitution: kv
total revisions: 3;     selected revisions: 3
----------------------------
revision 1.3
date: 2001/09/05 03:59:53;  author: wim;  state: Exp;  lines: +31 -4
Updated the French module to reflect all the numerous changes made to
the English module.
Still need someone who actually knows French to go through and
translate, however.
----------------------------
revision 1.2
date: 2001/06/23 04:13:14;  author: wim;  state: Exp;  lines: +4 -33
- move language eval code to a new module, Acmemail::Translator, just
like Acmemail::Mailserver does it
- fixed some typos in the English language module
----------------------------
revision 1.1
date: 2000/03/05 21:57:18;  author: acme;  state: Exp;
Added French language files
```

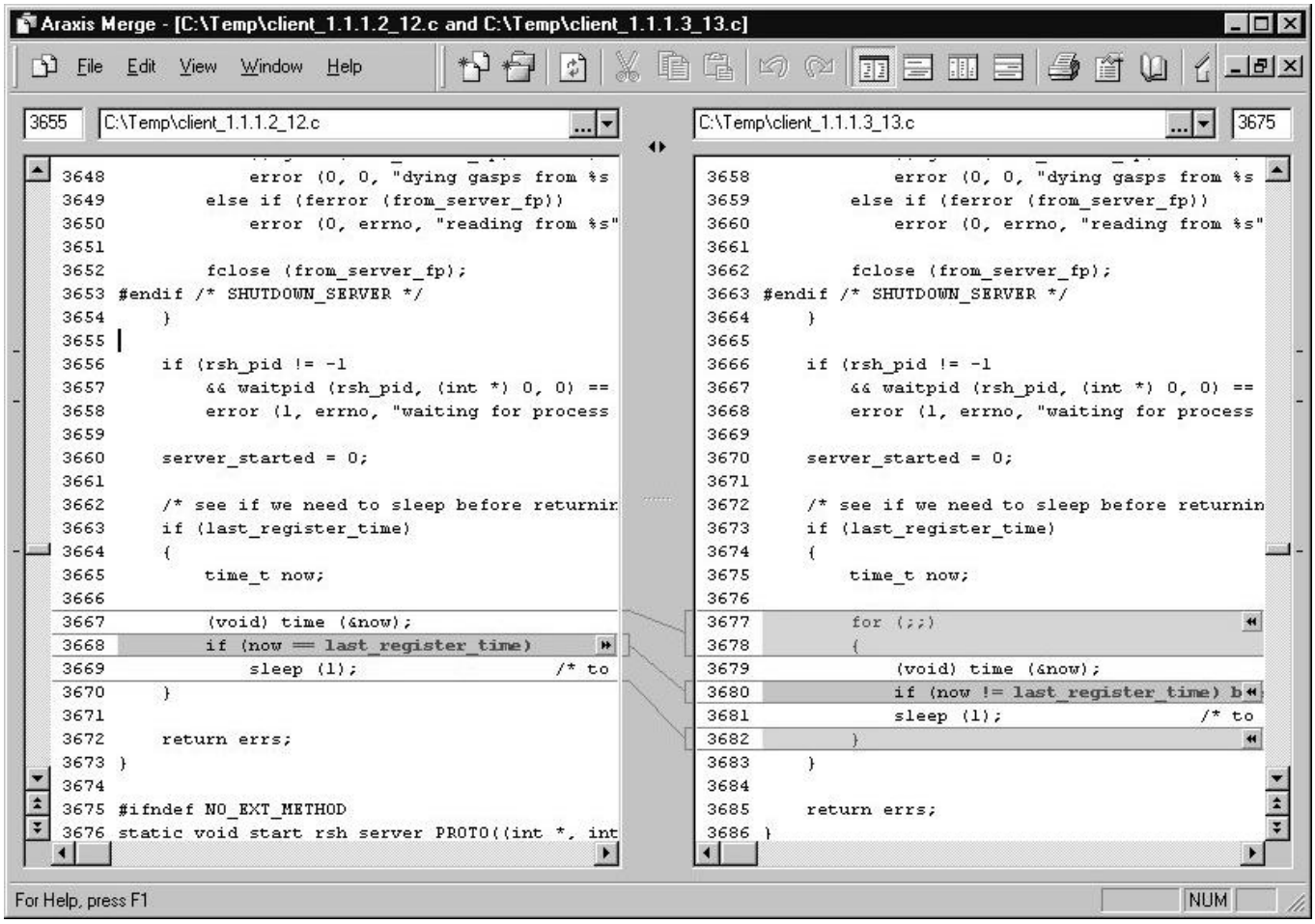
Appendix B: CVS Changes

Most revision control systems have a way of showing exactly what lines were changed in a file. This can be done between arbitrary revisions, versions, and dates. The change below was generated by the command ‘cvs diff –r 1.3 –r 1.4 README’. Automatically the server retrieves revisions 1.3 and 1.4 from the repository for comparison. The change below was caused by somebody adding several lines. The ‘12a13,19’ annotation represents the location in the original document that the lines were added. Lines added and removed are denoted with the ‘>’ and ‘<’ symbols.

```
Index: README
===================================================================
RCS file: /cvsroot/acmemail/sparkle/README,v
retrieving revision 1.3
retrieving revision 1.4
diff -r1.3 -r1.4
12a13,19
> ** IMPORTANT **
>
> Configuration is no longer done with AcmemailConf.pm or MyAcmemail.pm,
> but instead with Acmemail::Config. At the moment, you have to edit it
> by hand. Eventually we'll get an installation script which will help
> you write it.
>
```

Appendix C: Graphical CVS clients

In addition to the command line based versions of CVS, there are many graphical interfaces. The following screenshot shows how the WinCVS program allows a software programmer to easily see the changes between two revisions of a file. Color highlighting makes it very apparent what lines have been added, and what lines have been changed.



Glossary

Artifact

An object produced or shaped by human craft, especially a tool, weapon, or ornament of archaeological or historical interest.

Build

The act or result of assembling separate components into a complete deliverable program.

Branch

Split a project's development into separate, parallel histories. Changes made on one branch do not affect the other.

Checkpoint

A place (as at a frontier) where travelers are stopped for inspection and clearance

CVS

Concurrent Versions System, the dominant free and open source versions control system. It is useful for both individual developers and larger distributed teams.

Fix

A fix is a simple term to describe fixes for software defects. *See also Patch.*

Program

A sequence of instructions that a computer can interpret and execute

RCS

The Revision Control System. The father of CVS, RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, and form letters."

Release

A fully constructed version of a software product, that is ready for use by a specific audience. Releases include the distribution media, documentation, and training materials. Distribution mediums include CD-ROMs and Internet downloads. Audiences include end consumers, demo, preview, and testing users.

Repository

A file tree kept on a central server where things may be put for safekeeping.

Revision

A unique number that is automatically assigned to every change to every version controlled file. Whenever a change is saved, the revision number is incremented by one. *See also Version.*

Software Configuration Management

Software Configuration Management is the process of identifying, organizing, controlling, and tracking both the decomposition and recomposition of software structure, functionality, evolution, and teamwork. SCM is the "glue" between software artifacts, features, changes, and team members; it forms the ties that bind them all together from concept to delivery and beyond.

Tag

A tag is a unique name given to a revision of a file. The tag can be used instead of the revision number to access that revision of the file.

Patch

A patch contains changes made to original source. By using special tools, these changes can be automatically applied to the original source so that it includes the changes.

Version

An identifier arbitrarily assigned to a software product by the release engineer. Typically version numbers are assigned and maintained without the aid of a Software Configuration Management System.

References

Bawtree, H. (2001). *Beyond version control.* http://www.sdmagazine.com/documents/s=731/sdm0105d/ [2002, Feb. 13]

Brooks, F. (1995). *The mythical man-month.* Boston, MA: Addison-Wesley Publishing Company.

Brown, W. (1999). *AntiPatterns and patterns in software configuration management.* New York, N.Y.: John Wiley & Sons

Jarvis, A. & Hayes, L. *Dare to be excellent: Case studies of software engineering practices that worked.* Upper Saddle River, NJ: Prentice-Hall.

Leon, A. (2000). *A Guide to Software Configuration Management*. Norwood, MA: Artech House.

Kit, E. (1995). *Software testing in the real world.* Boston, MA: Addison-Wesley Publishing Company.

McCarthy, J. (1995). *Dynamics of software development.* Redmond, Washington: Microsoft Press

McConnell, S. (1993). *Code complete.* Redmond, Washington: Microsoft Press.

Mikkelson T. & Pherigo S. (1997). *Practical software configuration management: The latenight developer's handbook.* Upper Saddle River, NJ: Prentice-Hall.

Raymond, E. S. (1999). *The cathedral and the bazaar*. Sebastopol, California: O'Reilly & Associates.

Vance, S. (1998). *Advanced SCM branching strategies*. http://www.vance.com/steve/perforce/Branching_Strategies.html [2002, Feb. 13]