

A Simple Change Control Strategy

Language: Various, Platforms: Various

MarkCarroll, 2001

Introduction

This article outlines a very simple model which can be used as a basis for a change control strategy suitable for software development and maintenance projects. It requires no additional expenditure on tools or equipment unless you choose to take it further than what is outlined here. For complex development or maintenance projects, professional change control tools should be considered where a payback of less than eighteen months can be projected. Three key features for change control systems were kept in mind when writing this article.

1. There must be a clear route for rapid deployment in emergency or low risk situations.
2. Recognition that there is a scale based on impact of changes to the product and the risk should things go wrong - in other words there are BIG changes and there are little ones, some you feel relaxed about some make you lose sleep.
3. The change process must be reliable enough not to introduce additional errors itself.

Let's discuss risk here for a moment: it is pivotal to the approach discussed here. I know that there is a school of thought in Software Quality circles, which believes that all changes have the potential to be high risk and so, as the argument goes, should be managed as such. There is undoubtedly sound basis in fact for that school of thought but in reality commercial pressures often over-rule the 'always high risk' approach.

If a change system doesn't cater for those "fast track" changes, then often the choice is to either stick to your systems and potentially lose a customer or circumvent the systems to allow the change to take place. Both options are, I believe, less desirable than having a system that caters to quick fixes albeit with the odd gamble that occasionally the risk assessment was wrong and the dreaded defective upgrade has occurred.

For argument's sake, let's assume that you agree with the approach that allowing and planning for fast tracked releases, which are somewhat of a gamble, is a better option because the other options are even worse. How can we minimise the risk and impact of defective upgrades which have been fast tracked via the change system ?

Two complementary approaches are available:

1. Technical - have a back-out option available where possible.
2. Customer communication - "We believe the risk is worth the satisfaction that you will derive from these changes but beware software change risk assessment is a risky business - do you really want us to execute the fast track option for you or do you want to play it totally safe"?

You will be surprised how often customers will actually take the conservative approach and your need to deploy fast track release techniques goes away for a given situation.

The fast track approach in these notes is called the Patch process - if you find yourself hating the concept please consider the preceding paragraphs before dismissing it as being appropriate only for software "cowboys/cowgirls".

One final introductory comment - this article is based on the idea that change control is all about meeting customers' needs, and not about creating change systems. For that reason our main focus here is on the getting the two basic principles of effective change control right - those two basic principles are Identification and Storage. The suggested processes that work with these principles are not in the scope of this article.

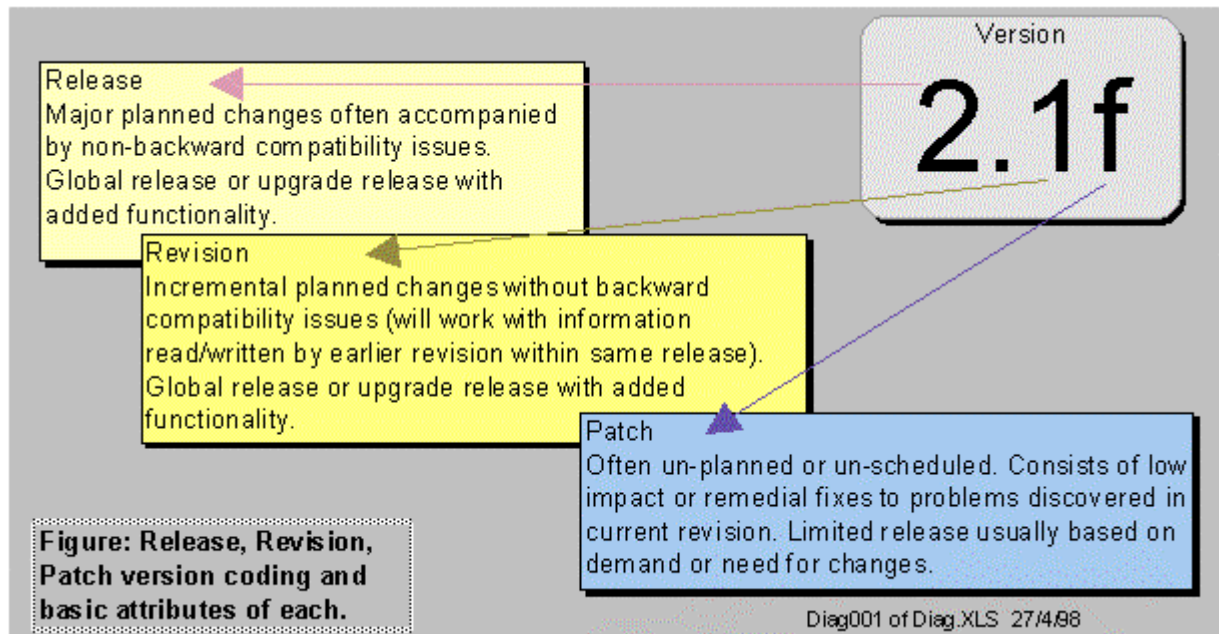
Basic Principle Number 1 - Identification

If you do not identify your product in a unique manner, confusion will always be inevitable. Confusion is costly for obvious reasons including:

- Wastage when the wrong product gets worked on only to have the endeavours scrapped and re-done on the correct product when it is identified.
- Loss of customer goodwill for the product, arising from situations where the wrong product is used or locating and verifying the use of the correct product is difficult for the customer.

In many software development and maintenance circles 'product' refers to the particular executable file or code carrying spreadsheet or Access database or Help file or whatever. Positive identification goes right down to that level.

Take a tip from your local automotive parts distributor - the part is a product and it always has a unique identifier even if it is a part of a bigger also uniquely identified product. In software this is often accomplished by using the file-id as we know, but that system can fall down when it comes to having multiple versions of the same product due to changes arising from fixes and enhancements. So here is the second part of identification: version numbering. The scenario suggested is similar to many common implementations that are in use with some well known products.



Basic Principle Number 2 -Storage

Using the Release, Revision and Patch identifying principles we tailor our storage model appropriately. The example shown below is done in Windows 95 and is appropriate for small systems development where less than approximately twenty products (.exe,.hlp etc files) contribute to a project which in itself is often a versioned product.

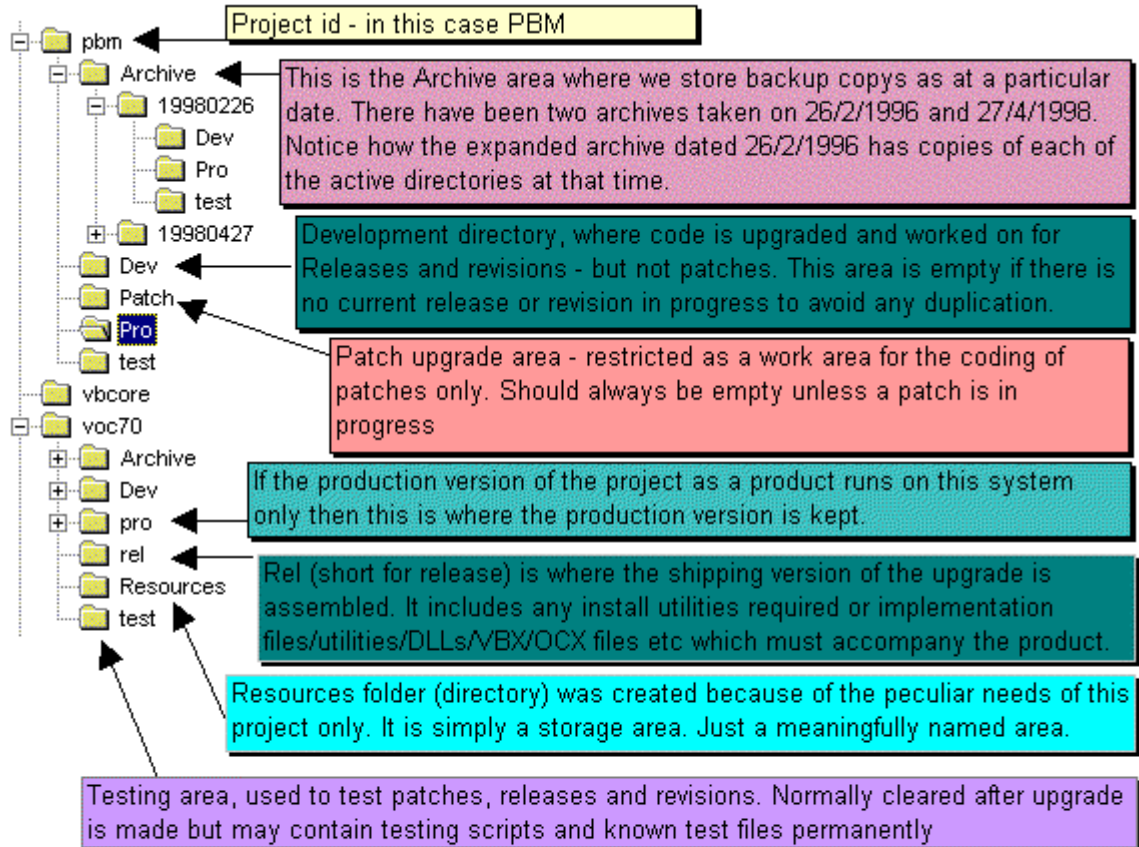


Figure: Example of product development and maintenance storage plan in Windows 95.

Diag002 of Diag.XLS 27/4/98

In future updates of this article we will discuss some important process and customer related issues

- When is an upgrade a release, a revision or a patch?
- How can something like the model shown be applied to Lotus Notes?
- How do customers expect to be informed of changes and how is that process best driven using the basic principles of identification?
- Is continually patching a sign of problems with the product, the maintainers or not a problem at all?