Configuration management

You probably roughly know what configuration management is, and you probably think it's boring.

It is – when it's done well.

The material in this section is not in "Using UML".

What is configuration management?

CM (or SCM) is:

- not just version control of source files.
- one of the most important aspects of process which a QMS may prescribe.
- essential to traceability, problem-tracking, making changes reliably: answering questions like:
  - which customers are affected by that bug?
  - if this team makes this change, who needs to be consulted or notified?
  - after this change which system tests do I have to rerun?

  (Note overlap with requirements management.)
- particularly important when there are multiple versions of a product (i.e. nearly always) and/or multiple teams

"Looking after what you've got so far" (Geoff Cozens)

Slide 35

Notice that it's important both in the initial development of a product (making efficient use of the developers' time) and after release. For example, if a major customer reports a bug or requests a feature as a result of their evaluation of your product, you may well want to release them a special version as well as, or even instead of, incorporating the change into future versions of the product. Your ability to do this fast – to find out what changes are required, approve them, make them and retest with minimum effort – can easily be the difference between making a major sale and not. And major sales influence minor sales...

Most importantly, good CM avoids wasted time and hence developer frustration. The fact that people hate spending their time making a change which turns out to be useless because of a change somewhere else that they weren't told about is even more important than the fact that the company hates paying for their time to do so. Of course it's important that the CM system be streamlined with a minimum of form-filling.

Version control

Basic ingredient of configuration management: important even on tiny projects and to individuals.

"Check in" changes with comments on what changed.

Tools such as SCCS, RCS, CVS, ... allow you to revisit previous versions, check what changes have been made, view change history, etc.

Extremely useful for backtracking when something goes wrong; means you can try something out knowing you can easily go back.

Emacs vc-mode makes this very easy: see VC submenu of Tools menu.

Strongly recommend you use VC in CS4 projects, for report, code, tests...

Personally I version control just about everything that might change – source code, tests, test results, papers, book chapters, CV, .emacs, ... and it has saved me considerable hassle many times.

Low overhead, high value.

Slide 36

Beyond versions of individual files

make and similar tools specify many practicalities of how to build a system out of its components.

However common problem is incompatible versions: part of CM's job is to record what works with what.

NB versions of compilers, linkers, system libraries etc. are just as crucial as versions of source, documents etc.

Common supporting practice: a component development folder collecting requirements, design notes, source, unit tests, test results, notes, problem reports, etc. pertaining to one module.

Slide 37

CM and QA

QA requires CM:

- CM practices have to be "defined" to reach CMM Level 3.
- ISO9000 doesn't say much explicitly about CM but does require e.g. being able to identify what versions of what components appear in what product – and in practice you won't be able to provide the proof of following good process without it.

CM supports QA:

- enables efficient review and audit: routine tracing work can have automated support
- if you can prove that something (e.g. a test, a review) has been done before – using CM – don't need to duplicate it.

Slide 38

Implementing CM: organisational issues

Tool support is essential to get full benefit – but many companies still use nothing more than version control (or not even that!)

No point in trying to introduce CM unless VC is already in place – walk before running.

Buy or build? issue.

Difficulty in identifying requirements: levels in the company have diverse requirements.

Organisational culture interacts of course...essential not to lose sight of aim to *reduce* effort overall and to get buy-in.

Evaluate tools: try out on pilot project: roll out to organisation: REPEAT.

Slide 39

In short, adopting CM is an example of a major process change. Like a development project, such an undertaking has objectives, stakeholders, requirements, risks, stages, different design possibilities... It may well be worthwhile managing such a change as though it were a development project, i.e. bootstrapping with what you know about how to handle these issues in another context, software development. For example, consider an iterative approach to manage risk, rather than an all-at-once switch to the desired final process (even after prototyping it on a trial project).

More often, organisations assume that changes can be introduced at a stroke: IMHO this is largely responsible for the negative attitudes we often see to QA in general.

Remember training!

Daily Build and Smoke Test

Good configuration management is one way of controlling *integration risk* – the risk that the components they don't work together properly when you integrate them – because it helps track the way in which things change and make sure the communications happen.

One way of verifying that certain such errors are absent is to *look for them early and often:* in the extreme, some projects especially at Microsoft every day:

- build (get out of CM, compile, link)
- "smoke test": run a suite of simple tests looking for serious errors.

Slide 40

Advantages of daily build and smoke test

- minimise integration risk
- reduce risk of low quality
- support easy defect diagnosis
- improve morale
- ... ?

Slide 41

Risks of daily build and smoke test

Cost: as usual, managing a risk has an associated cost and whether it's worthwhile depends on the magnitude of the risk in this project's case

Confusion: developers are trying to "follow a moving target"; they must update their code to a changed interface immediately.

Complacency: may mistake passing the DB&ST for being bug-free, since if something is hard to achieve it's difficult to remember that it may not be everything (cf ML "if it compiles it must be right")

Difficulty of making major changes: if the norm is to have a working version every day, there's a strong disincentive to consider major architectural changes. (But this is ambivalent.)

Slide 42

More details in `www.construx.com/stevemcc/bp04.htm`

As you see, there are difficult issues involving both technical factors (what should the software be like, how can we build it, how can we be sure it's correct) and organisational factors (what will work for us, how will we get it done in time and to budget, how will we stop all the developers getting fed up and leaving). Like design problems, these can be addressed using patterns; the best source I know is Jim Coplien's "Development Process Generative Pattern Language" at `www.bell-labs.com/~cope/Patterns/Process/` which is very well worth browsing. For example, he addresses the question of how often builds should happen in his pattern Named Stable Bases. In very brief:

Coplien's pattern Named Stable Bases

See `www.bell-labs.com/~cope/Patterns/Process/` for details.

Problem is how often to integrate.

Forces are basically the advantages and disadvantages of DB&ST: they pull you towards and against integrating daily, so what do you do?

Solution "Stabilise system interfaces – the architecture – no more frequently than once a week. Other software can be changed (and even integrated) more frequently."

May want to combine approaches: e.g. from an ATT project: nightly build (only), weekly build with sanity check, biweekly more thoroughly tested build.

Slide 43

Defect tracking

When a problem is reported (from a customer or from an internal customer) essential to make sure it gets dealt with in a timely fashion even when many people and stages are involved.

Bug reports will be categorised (how severe? really a bug, a misunderstanding, a change request?), analysed, assigned to someone to act on, acted on. The bug reporter will need to be informed.

You need to keep the information to avoid doing all the work again next time someone reports the same bug (even to a different person).

Tools exist: probably best overall is GNU's GNATS which also has the advantage of being free.

We use GNATS in the department to track defects in our systems (starting with mail to faults).

Slide 44

Finally used cartoon on bugfixing, carrying the (elementary but important) point that when you fix a bug you should routinely ask yourselves:

- Where else might I have made this mistake?
- What next bug will I find hiding behind this one now that I've fixed it?
- How can I improve my process so that this kind of mistake becomes less likely?

In each case the idea is to decrease the overall amount of time spent bugfixing. Note that this applies just as well to design bugs as to coding bugs.