19 Configuration Management

Configuration management

Introduction

What

This chapter deals with how to control a software product as it evolves. It describes Levels of control and management (p.19-5), and describes means to cope with the complexity of product management.

The material covered here is complementary to the theme on Process models, which deals with the *processes* that lead to changes. Both chapters present different perspectives on what is generally known as *configuration management*.

Configuration management has hitherto been an immature field in software engineering. There is little consensus in terms of standards or handbooks. The most important standard, ANSI/IEEE Std 828-1983 *IEEE Standard for Software Configuration Management Plans* hence states that every major project should define its own standards. Here we present our view on Configuration Management, which should help in defining such plans for projects and companies.

Configuration management is a comprehensive subject: it comprises issues as diverse as identifying a specific component in a system, to managing error reports from customers. Between such extremes we need to describe the structuring of systems in subsystems and components, and how these components and subsystems shall be generated in order to produce a system with the desired properties.

We contend that there are Levels of control and management (p.19-5) that can be identified: to achieve Configuration Management (p.19-12) we need a platform for Configuration Control (p.19-9). To achieve Configuration Control we need a platform for Version Control (p.19-6).

We also provide a Classification of configuration management tools (p.19-17), and look at issues important when Introducing Software Configuration Management into a company (p.19-18).

Why this is important

The “soft” in the word “software” is intended to indicate that programs can easily change their form. This linguistic picture is accurate enough: we all know that software can indeed be changed very easily.

At first sight, this characteristic of software seems to match very well with one of the key requirements for commercial success in software development:

> To be able to produce and sell as many concrete systems as possible while minimizing costs related to details of particular instances

Since software is “soft” and easy to change, software-based products should also be easy to change. Or are they?

Experience shows that software systems are not, in practice, so very easy to change. An important reason for this is that this very flexibility of software also has its negative side: software items not only *can* change but typically *do* change - and often. This leads to large numbers of "versions"[1] of software items, and there are often difficulties in maintaining a clear picture of:

- exactly *which* versions exist
- the essential *characteristics* of each
- *inter-dependencies* amongst the versions.

The lack of effective methods to handle variability in software items can cause problems ranging from minor frustrations to considerable development and maintenance effort being wasted through selection of incorrect versions.

A related issue is that of software *reuse*. The advantage of reuse is clear: development costs are reduced through utilization of existing items instead of developing new ones. But successful reuse adds to the problem of handling variants. In simple cases, an item can be reused "as is" (i.e. without any change being applied). Even this adds to the complexity of variant handling, as extra inter-dependencies have to be handled. In more complex cases, items are reused with changes to customize for the new intended use of the item. This is an added source of variability.

So: if you need to produce new system instances quickly and easily, or if you want to re-use software components to minimize development costs, you need to have effective ways of handling product variants.

Objective

The goals of configuration management can be summed up as follows:

- identifying the items that are to be managed
- controlling changes in terms of the change process, and registering and reporting status
- controlling that items are complete and correct, involving registering and reporting item status, and controlling the distribution and use of changed items
- maintaining approved configurations of the items, implying the control of correctness, completeness and consistency
- controlling which items are included in a given product

These are the practical and pragmatic reasons for employing configuration management and configuration control in a software engineering context. I.e. it is part of ensuring that the products one develops, delivers and maintains adhere to the required quality w.r.t.

- the customers use of the product
- the resources used by the producer to develop and maintain the product

1. The word "versions" is used here in a wider sense than simply *file* versions. For instance, a single C source file might contain many #ifdef directives, resulting in a very large number of semantically distinct programs (depending on the values chosen for the parameters which control the #ifdefs).

Definition of terms

As there is no universal consensus on the use of terms in the field of software configuration management, we provide definitions of the terms we use:

1. Configuration Control
2. Configuration Management
3. Configuration Control Board
4. Configuration Management Plan
5. Version
6. Variant
7. Revision
8. Identification
9. Attribute
10. Status
11. Configuration
12. Configuration Item
13. Baseline

Note that we in configuration management are only concerned with the “physical” Configuration Items that are parts of the systems and products we are building and maintaining, as apposed to the (semantic) building blocks we deal with in the system domain. There is not always a one-to-one correspondence between logical building blocks and physical Configuration Items; you may experience never locating the logical building blocks when looking at the structure of the Configuration Items.

Levels of control and management

The issues involved in configuration management are many and diverse, as we have seen. We believe that by categorizing them into levels, where each level provides a platform for the next, both talking about it and doing something with it becomes easier.

We have found the following levels to be of use:

- Version Control (p.19-6), which has to do with describing, identifying and retrieving any significant version of an atomic entity (item) in the (syntactic) description domain. Included here are issues such as efficient representation and retrieval of configuration items
- Configuration Control (p.19-9), which deals with controlling the development of products that are composed of different parts, and where the parts (items) evolve into many different versions. Controlling the transformation of components by tools into complete products (build support) also belongs here.
- Configuration Management (p.19-12), which deals with the management view, i.e.:
  - what is delivered to a specific customer (release support)
  - which problems and discrepancies are reported for which delivery, by whom (and what status do they currently have?)
  - that improvements and changes are initiated depending on analysis of technical and economical consequences, and only then (change control)
  - that the quality of the product and development process is monitored (process management)

Note that all three levels are often called configuration management, while we have reserved the term Configuration Management to the managerial aspects of the broader issues of Software configuration management.

Version Control

In this section we describe What can be obtained (p.19-6) by Version Control, and what Techniques (p.19-6) are available.

What can be obtained

With Version Control we obtain an unambiguous identification of the (syntactic) elements that a system is built from. This provides a necessary prerequisite for building a correct system.

Version Control is the minimum level of control needed to manage the development and maintenance of entities that make up a product.

Version Control essentially provides an unambiguous identification of entities. It discriminates between revisions and variants, thereby identifies improved and alternative revisions of entities.

Furthermore it enables a customer and a producer to communicate about versions of products, so that errors and enhancements can be performed with knowledge of the delivered components.

Version Control must include measures to avoid simultaneous (and destructive) updates of entities.

Versioning mechanisms can also be used to classify entities and to make statements about properties and quality aspects of a component, e.g. whether it is a working version or a more official version.

Techniques

The following techniques for Version Control are presented:

- Revisions, Versions and Variants (p.19-6),
- Variant Production (p.19-7) and
- Identification and selection of versions (p.19-8).

Revisions, Versions and Variants

These are terms that often are used in an imprecise manner. The definitions we have chosen for Version, Variant and Revision are in short that any change to a Configuration Item produces a new version, which is either a revision (that is meant to replace the previous version), or a variant (which is meant to coexist with the previous version).

Note therefore that a variant can be modified to produce a revision of the variant (meaning that it is a “better” variant than the previous version).

Variant Production

Often one has a need to develop several variants (of a component or system) to satisfy incompatible needs of customers (e.g. variants for different computer platforms) or for reasons internal to the developing organization (e.g. maintaining several product releases with different functionality simultaneously, with and without corrections).

This can very soon lead one to a situation where the number of variants of a single component makes management awkward, like having to perform identical changes (e.g. due to a bug fix present in many variants) to many versions. One needs to rationalize; below a number of alternatives are given.

1. Variants can be given different names in their Identification, where one can see from the name what variant it is (e.g. file name with naming conventions to distinguish between variants for different computer platforms).
   This alternative easily gets out of hand if the variants become many and not related to commonly understood variability such as computer platform or language, and also requires physical storage for complete versions of all variants.
2. Variants all share the same name, but are distinguished by their revision and version number in the Identification. This is the method typically adopted in file repositories such as RCS, CMS PVCS and SCCS, which adopt measures to reduce storage space using some delta mechanism (only storing one “master” version and the differences - deltas - between versions). A configuration description can determine which version one should use of modify.
   Although this saves storage space and name space, one still may have to perform identical changes to many versions. Merge functions in the versioning tool can reduce the number of versions one has to modify for e.g. a common bug fix, but all “external” variants must non-the-less be maintained in parallel.
3. Conditional text is a mechanism that has been present in programming languages for many decades, and which to a certain degree is found in other sources such as text processing systems (like FrameMaker and Word), but seldom in design tools. The idea is that to use a preprocessor on the source text (containing “macros” that indicate inclusion or exclusion) before invocation of a tool (e.g. a compiler), and configuring the tool (supplying parameters that select macros).
   This mechanism, if available, is useful in situations where the number of variants is limited, the macros are concepts that are easily understood (e.g. “test”, “host”, “target”), and one can control the transformation process (selecting the correct macros for the correct transformations). Applied in addition to versioning systems one can reduce the number of physical variants (file versions) one has to modify.
4. Change oriented versioning is a fairly new approach that is not commonly found yet, unfortunately. The idea is that the repository can itself add or remove macros according to conditions (intentions) supplied in the form of logical statements (e.g. test AND host AND bug_fix_345 AND NOT bug_fix_211).
   Change oriented versioning is the only mechanism that would significantly reduce the number of parallel updates required to maintain many variants, and is therefore a promising future possibility. Text processing systems implementing conditional text are the closest one comes to change oriented versioning at present.

The opposite function, that is merging variants to become a single version, is supported by many CM tools (at least on textual sources like code files). The most advanced tools offer multiple windows, showing the file versions being merged, the merged result, and lets one select contents at the event on conflicts, also letting one enter new input into the merged file if appropriate.

Identification and selection of versions

All repository tools (like RCS, CMS, PVCS and SCCS) have their own way of assigning an Identification of a Version of a Configuration Item. For items not stored in the repository (e.g. physical items) one should adopt an identification scheme which is as far as possible in accordance with the repository tool's.

The identification scheme typically consists of a name (e.g. file name) and a set of numbers, denoting revision and variant in some structured fashion.

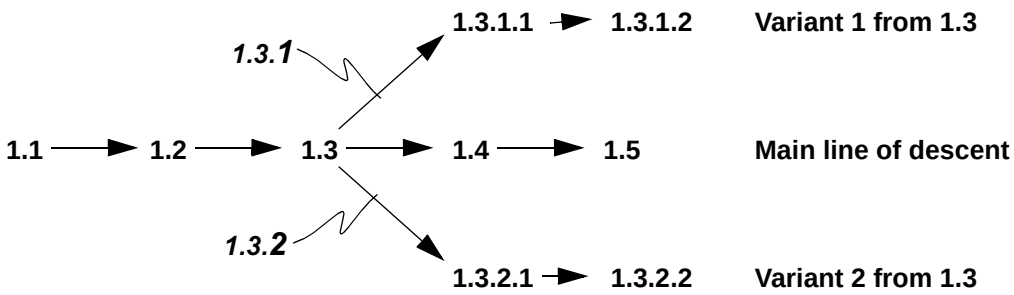

Figure 19-1: Revision tree (RCS type)

Selecting a version (either one wants to modify it, simply look at it, or use it in a build operation) can be a non-trivial task. The simplest and most common is to select the "latest and greatest", that is along the main line of descent. But one may want one that is identified in a configuration description for a certain product version, or one may want some other version of particular functionality or quality.

One technique for selecting versions is based on assigning an Attribute to items. Such attributes can denote general properties that are common for many items (e.g. quality attributes such as "approved", or other general attributes such as "host", "language", "author" or "modified"), or properties that are particular for few or only one item (e.g. attributes identifying functional changes and bug fixes).

Each attribute may have a legal value set (e.g. language = English, German, Norwegian). Such attributes must be recognized by the repository, which must associate each item version with applicable attribute values; it must be possible to uniquely discriminate each version by a combination of attribute values. Not all attribute values need to be bound, only enough to uniquely identify a version. Default attribute values can be defined (e.g. approved == TRUE).

This approach is possible in some of the more advanced tools ("Views" in ClearCase, intentional version selection in PCL).

Configuration Control

In this section we describe What can be obtained (p.19-9) by Configuration Control, and what Techniques (p.19-9) are available.

What can be obtained

Configuration Control adds structure to Version Control: a formalized description of how a product is composed of (syntactic) parts.

It also adds formalized procedures for the transformation processes, describing required input (e.g. source code) and the resulting output of the transformation. This is called Build support (p.19-10). By adhering to naming conventions, checking on the existence of input and output entities, and checking on generation times (timestamps), transformations can be performed automatically and only when needed.

With Configuration Control we can improve system documentation by expressing the composition of the system in a Configuration Description (p.19-10). Regeneration of a specific configuration becomes simpler and more reliable, and automation is made possible.

The production of new variants can be performed with greater ease, quicker and with less probability of errors by making new versions of the configuration descriptions. Such descriptions can be maintained independently from other version.

The combination of configuration descriptions and component classifications (see Identification and selection of versions (p.19-8)) enables one to deduce the status for a particular configuration from the individual component status, with the addition of results from integration tests etc.

Communication about the system becomes easier and more secure, in that structures can be identified and isolated, and given appropriate identity in the configuration description. Evolution of the system can be performed with greater control and with less dependence on the individual developer.

Techniques

The following techniques for Configuration Control are presented:

- Product structure (p.19-9)
- Build support (p.19-10)
- Configuration Description (p.19-10)

Product structure

Maintaining a formalized description of the structure of the (semantic) parts that make up a system or product is invaluable to configuration control.

Product structure should not only describe this part-of structure, it should also document interrelationships between components. Documentation items should be linked to the components they explain, requirements should be linked to their solutions, executable software items should be linked to descriptions of the hardware that run them and so on.

The level of detail, the granularity and the classes of components that are part of such a description depend on the needs of the company and the system being described, but unfortunately also very much on the tools that are available. The current situation in tool support leaves much to be desired, as typically non-software items cannot be modelled decently, and user-defined relations are not commonplace.

Build support

An addition to the description of the product structure is the description of the transformation process carried out by software production tools like compilers, linkers and loaders.

Such a formalization has been commonplace for the most common tools for many years in the form of make-files (native to unix and PC platforms). But with integrated tool chains as one finds in the SDL world, the transformation process can consist of many steps: selecting an SDL component (or collection of components), generating code using a code generator with appropriate options (e.g. determined by the software design/ framework), and then compilation and linking of the generated code. The transformation description must cater for all this, and preferably be capable of automatic tool invocation. Support for defining manual steps formally is seldom a feature of CM tools.

Given that the product structure and the transformation process has been described, one can hope for good support from the CM tool to rebuild subsystems and systems efficiently, i.e. only regenerating the parts that are effected by changes since the previous system generation (typically only generating or compiling the parts that are affected by a recent change). Advanced support for parallel, distributed building is offered by some of the CM tools.

Some tools also not only rely on timestamp information to determine the need to regenerate, but also record which options where used for the transformation. Such information is then stored as an attribute of the destination elements (outputs of transformation tools like compilers). This level of refinement is necessary where several components with identical names but different building rules are used in more than one place in a system.

Configuration Description

A configuration description is a formal description identifying exactly which versions of which configuration items are involved in a particular system build, and how each derived component was produced.

This information is essential for the capability of rebuilding (reconstructing) a particular system version, and for identifying exactly what went into the system and how transformation tools were used; all important information when analysing error reports and change requests.

Maintaining such configuration descriptions can render the need to store derived components in the repository unnecessary, although the choice between saving space by not versioning derived components must always be weighed against the time required to rebuild them on demand (some types of system generations can take days).

Configuration Management

In this section we describe What can be obtained (p.19-12) by Configuration Management, and what Techniques (p.19-12) are available.

What can be obtained

Configuration Management adds the managers view to Configuration Control. It formalizes the contact with customers, registering error reports and requests for new functionality (enhancements). It initiates and controls the evolution process, and controls the termination of changes and the release of new product versions.

Procedures for change management give the company a controlled, efficient and market-driven product development throughout the product life-span. It helps build up a base of customers and experience, and is a place to find market trends amongst customers.

Configuration Management contributes to highlighting valid configurations, and lets one define standard solutions where undesirable (conflicting) feature interaction is avoided. A company can more easily answer tenders that do not require new (unexpected) development work.

Techniques

The following techniques for Configuration Management are presented:

- Managerial issues (Roles in Configuration Management (p.19-12)):
  - Quality Control Board (p.19-13)
  - Configuration responsible (p.19-14)
  - Configuration Management Plan (p.19-14)
  - Project Worker (p.19-15)
- Change control (p.19-15)
- Release support (p.19-16)
- Process management (p.19-16)

Roles in Configuration Management

A traditional project model is taken as a basis in the following. A project is given a Configuration Management Plan defining rules and procedures. The plan is defined and controlled by the Quality Control Board (p.19-13), while the Configuration respon-

sible (p.19-14) performs the bureaucracy concerned with Configuration Management.

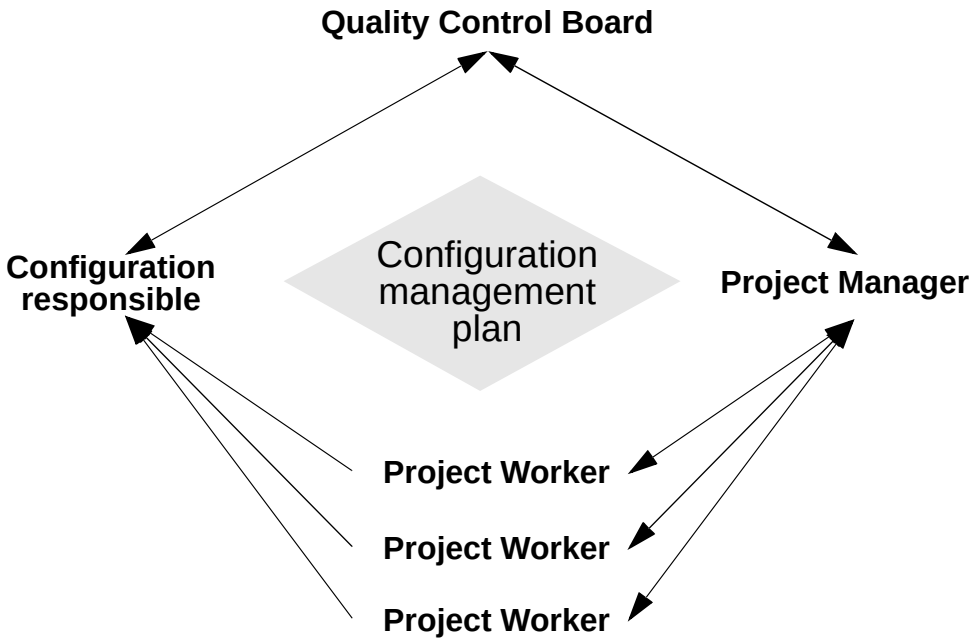

Figure 19-2: Roles in configuration management

The types of roles we thereby define are

- the *performing* role played by the Project Worker (p.19-15)
- the *controlling* role of the Quality Control Board (p.19-13) that makes decisions regarding requested changes, approves of implemented changes, and determines which choices (variants) are to be available
- the *registering* role of the Configuration responsible (p.19-14), that is a link between the performing and controlling roles, by maintaining lists of change requests and making approved changes available for further development

Note that there roles will exist independent of what tool support one has; good tool support simply helps people perform these roles.

Quality Control Board

The company should have a body that is responsible for quality assurance, as required by the ISO 9001 standard series, both for development and manufacturing. Configuration Management applies to product development, and comprises:

- defining plans and standards for quality assurance and configuration management
- ensuring that these plans and standards are followed
- initiate development and maintenance projects according to analysis
- approve development results at project milestones

The organization of the Quality Control Board (including the Configuration Control Board) must be tailored to the companies needs: in small-scale situations the project manager, quality manager, product manager and configuration responsible may be roles played by a single person and constitute the control board; in large-scale situations a hierarchy of responsibility and control may be appropriate.

The important issue is to focus on roles, assign them in a rational way, and be conscious of which role one is playing at a given time.

Configuration responsible

This role is a support function that takes care of the necessary "red tape" involved in configuration management, the object being to alleviate the creative work of the Project Worker (p.19-15) from such issues. Examples of the activities performed by the configuration responsible are:

- create, implement and manage procedures and rules, as approved of by the Quality Control Board (p.19-13)
- register and collect problem reports (errors) and change requests (suggestions for functional improvements), and submit these to the Quality Control Board (p.19-13)
- identify components (items) created and modified during development
- identify and register approved configurations of these items
- performing regression tests and reporting results to the control board
- composing complete configurations for releases
- manage tool use (and versions of tools)

Good tools can support many of these activities.

Configuration Management Plan

As mentioned earlier there is little consensus in terms of standards or handbooks for configuration management. The most important standard, ANSI/IEEE Std 828-1983 *IEEE Standard for Software Configuration Management Plans* hence states that every major project should define its own Configuration Management Plan.

Such a plan should be an integrated part of the quality system required by the ISO 9001 series of standards, and typically specify:

- the organization, roles and responsibilities within quality assurance, including configuration management
- procedures to manage error reports and change requests, showing how decisions are to be made and carried out
- procedures to be followed when performing functional changes to products or systems, identifying quality assurance activities
- which quality assurance techniques are to be performed (e.g. walkthroughs, reviews, testing, validation) in which situations, paying special attention to regression testing

- how achieving and identification of versions is to be performed, and how to represent variants wrt. customers
- procedures for managing deliveries to customers
- use of tools for configuration management

An appropriate plan will vary depending on:

- the degree of parallel development
- the ratio of error correction versus development
- quality requirements (risk involved)
- number of variants available for customers
- the number of configuration items

A tool with process support can define much of this; but planning is still required!

Project Worker

One should strive to relieve the individual project worker from the tedious details of configuration management. This is only partly possible, depending on what (tool) support is available.

The project worker will be involved in:

- determining which configurations of items are possible (i.e. yield a valid result) - information which the Configuration responsible (p.19-14) must record for use by the Quality Control Board (p.19-13)
- performing the functional changes decided by the Quality Control Board (p.19-13) via the Configuration responsible (p.19-14), who identifies the Baseline from which to develop. Information about the functional properties of the baseline should be found in the configuration description.

Change control

The management aspects of change control have already been mentioned in the previous sections on Quality Control Board (p.19-13) and Configuration responsible (p.19-14). There are in addition a number of technical issues that are relevant:

1. Change records: the change history of a component should record not only the time and date of change and the difference, but also the reason for perform the change. Such change forms can be manual or part of CM tool functionality.
2. Change groups: a single change request as supplied by e.g. a customer will more often than not require changes in many components, and in components of various types (e.g. both in computer sources and documentation). The changes to the components should be grouped to be able to initiate, monitor and report on the changes to all affected components as a unity.
   Some CM tools now provide mechanisms for change sets or change packages, where the tool provides support for grouping the individual items changed, and naming the change with an identifier.

Release support

Keeping track of which customer has received what can be a very important set of information if there are many variants of a system in the market. This should preferably be maintained by the CM tool; at least there must where necessary be unambiguous links between the customer database and the version descriptors for the delivered systems, or the customer should be able in some way to find version identifiers on or in the system itself, which can be quoted when communicating with the producer or service personnel.

Process management

The Configuration Management Plan defines amongst other things a process that shall be followed in Configuration Management activities, as mentioned before. The most advanced CM tools can be adapted (programmed) to support many such activities, plus other activities that are part of the software development process; we call this Process Management.

Such tools can support the life-cycle phases which a component runs through, and can define different schemes for various classes of components (e.g. a textual requirements specification and an SDL operator will typically have very different steps of quality assurance).

We recommend that one doesn't invest in tools of this kind (or use these features) unless the software process is well defined, well understood and stable. Support in this field is a enabling technology on the higher level of maturity on the CMM capability model.

Classification of configuration management tools

A useful classification of CM tools follows our definition of Levels of control and management (p.19-5) with some overlaps:

1. Basic versioning tools. Support Identification and selection of versions (p.19-8), Version Control (p.19-6) and basic (not distributed) Build support (p.19-10). Simple tools to use that solve rudimentary issues such as identifying revisions and variants, and the automatic regeneration of a system after a change. Classic examples are the basic use of *rcs* (or *sccs*) and *make*.
2. Configuration Control tools. In addition to the functionality of Basic versioning tools, they support Product structure (p.19-9), Configuration Description (p.19-10) and Release support (p.19-16). Tools at this level should combine the configuration description and the transformation description when performing build support, and offer distributed Build support (p.19-10).
3. "proper" Configuration Management tools (using our definition of the term). Supports (in addition to the above) Roles in Configuration Management (p.19-12), Change control (p.19-15) and Process management (p.19-16). This implies handling error reports, change requests, impact analysis, approval mechanisms etc. They are capable of describing and formalizing processes which do not lend themselves to automation (i.e. humans are involved).

Pleasingly enough, OVUM in their 1995 evaluation of Configuration Management Tools [148] report that "CM tools have advanced by leads and bounds" since their 1993 evaluation, producing some very interesting results, the most important being that several mature tools in the third level are now available (e.g. ClearCase, Continuus/CM and PCMS). This is also the conclusion of the latest SISU evaluation of ClearCase [170].

Introducing Software Configuration Management into a company

Introducing Software Configuration Management (SCM) is and should be tightly linked with the definition of the quality system, such as defined by the ISO 9001 standard series. Much of what applies to introducing ISO 9001 into an company also holds for SCM, and involves improving the development process:

- analyse your present situation, identify areas of improvement, select the areas within which improvement is sought (priority depends on risk analysis, cost of means and level of improvement), make a plan for introduction
- involve all levels of the organization, especially seeking management commitment to the plan - and involve enthusiastic promoters (look for influential individuals)
- inform all involved of the plan
- implement the plan, and measure its effects on chosen metrics
- evaluate results at predefined checkpoints, and inform all
- learn from your mistakes and successes - and document them!
- iterate all the steps above as long as you live

Tools or manual procedures?

Experience has shown that the main problem with introducing Software Configuration Management (including Configuration Control) is not the use of tools, but establishing routines and procedures. This causes us to recommend that manual procedures be an appropriate starting point for any organization trying to swap chaos with control.

The following organizational measures are recommended:

1. Create a body responsible for quality assurance and configuration control, the Quality Control Board (p.19-13)
2. Create a support function for filing and possibly regression testing, the Configuration responsible (p.19-14)
3. Define a Configuration Management Plan (p.19-14) as part of the company's quality assurance system

Development work is performed by the developers (project workers) as usual.

Only when the organization is capable of defining, recognizing and playing these roles will the addition of tools be of any significance.

Whether you choose to leap-frog from "no CM tool" to a level 3 CM tool (see Classification of configuration management tools (p.19-17)) or whether you choose a stepwize approach is a matter you must settle in your improvement plan; note though that tools on level 3 differ in the extent to which they let you refrain from defining your process, so that you can make do with functionality at levels 1 and 2; it is possible to apply a stepwize approach without changing your CM tools!

Many find the price of advanced CM tools intimidating; it may therefore be of little comfort to learn what experience has shown: you can count on spending more money in terms of time and energy when introducing the tool into the company than the actual cost of the tool.

For more about introducing CM tools into a company, refer to e.g. the 1995 OVUM report *Evaluation of Configuration Management Tools* [148].

List of figures

List of definitions

Attribute

is an aspect of a Configuration Item that gives additional information, e.g. about its functionality.

The attribute is not part of the Identification of the item.

Baseline

is the designation of a “snap-shot” (typically in time) of a product or system, with a specification of all Identifications of all Configuration Items that are part of it.

A baseline may also have a more specific definition, implying that all the configuration items included in the baseline have a certain Status.

Configuration

is a particular composition of a product or (sub)system from particular components (items) with a defined functionality.

Configuration Control

The formal guidelines for

- describing the configuration of a system or product on the basis of the identification and status of each Configuration Item it consists of
- describing the derivation process and rules from source components through derived components to a complete system
- coordinating and approving changes in this description

Configuration Control Board

A body which is

- responsible for evaluating change requests
- capable of ordering their execution
- capable of monitoring their completion

Configuration Item

is an entity which is subjected to Configuration Management and is treated as atomic (indivisible) in this respect.

A configuration item may consist of parts, but these parts are then not managed as parts according to configuration management (e.g. a printed circuit board may be a configuration item, while the components on it are not subjected to configuration management).

A configuration item is concerned with the (syntactic) descriptions that a system consists of, and not the (semantic) building blocks in the system domain.

Is also informally called part, entity or component.

Configuration Management

The formal guidelines for

- identifying and defining the Configuration Items a system is composed of
- recording and reporting the status of entities and requests for change throughout the components life-span
- evaluating and initiating changes
- controlling the change process
- verifying the release of system versions

Configuration Management Plan

A document which describes how Configuration Management shall be carried out in a project or for a product. It describes:

- roles and role responsibilities
- the process for evaluating and implementing change

It defines:

- component statuses and corresponding approval criteria
- identification criteria
- methods for inspection, approval, filing etc.
- types of items to be managed

- tools to be used

Identification

is an unambiguous designation of a Configuration Item that is part of a system or product.

The identification can consist of a name, type, revision number and variant.

The identification can not be changed during the life-span of the item.

Physical items will bear the identification.

Revision

is a Version of a component that is derived from an earlier version, and that is designed to replace the earlier version. The difference between two succeeding revision is usually a “small” improvement (error correction or enhancement in functionality). The latest revision is the version one intends should be used (“latest and greatest”).

Status

is an Attribute of a Configuration Item that qualifies it, e.g. in terms of formal approval or what quality criteria it fulfils.

As apposed to the Identification of the item, the status will change.

Variant

is a Version of a component that is designed to co-exist in parallel with other versions of a component, as an alternative. One variant of a component is seldom “better” than another, but offers different alternative functionality (e.g. for different computer platforms).

Version

is a common term for Revisions and Variants.

Is also used to denote an identified product configuration with a defined status, typically indicating a larger change than a new revision (e.g. version 5.0 of FrameMaker).