

Configuration Management Principles and Practices

Ronald Kirk Kandt

Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, CA 91109, USA
ronald.k.kandt@jpl.nasa.gov

Abstract. This paper identifies fundamental principles and practices essential to the successful performance of a configuration management system. Practices are grouped into four categories that govern the management process, ensure product quality, protect software artifacts, and guide tool use. In addition, the practices are prioritized according to their effect on software products and processes and the coverage of the identified principles. When these practices should be applied in the software development lifecycle are also discussed. The potential for automating each practice and its validation is also considered. Finally, the necessary capabilities of a configuration management tool to fully support these principles and practices are identified.

1 Introduction

A configuration management system includes the set of policies, practices, and tools that help an organization maintain software configurations. The primary purpose of a configuration management system is to maintain the integrity of the software artifacts of an organization. Consequently, configuration management systems identify the history of software artifacts and their larger aggregate configurations, systematically control how these artifacts change over time, and maintain interrelationships among them.

To support configuration management, several configuration management tools have been developed over the years. Some examples include SCCS [1], RCS [2], and CVS [3]. These systems are freely available, delivered with some operating systems, and still used today. Several newer systems provide much greater functionality. However, no configuration management tool widely available today provides all of the capabilities that support the practices identified in this paper, although it is probable that some will within the next five to ten years.

The primary purpose of this paper is to identify principles fundamental to successfully managing the configuration of developed software and to identify those practices that best embody them. This effort is not entirely original. For example, several configuration management best practices are explicitly defined in [4], [5], and [6]. These practices resulted from introspection and by identifying a consensus among the industry. In addition, many others have implicitly identified best practices by defining configuration management processes and procedures [7], [8], [9], [10] and evaluating configuration management tools [11].

Secondary purposes for identifying the configuration management principles and practices are many. First, organizations can use these principles and practices to train configuration management specialists and evaluate their configuration management practices. Second, these principles and practices can be used to define a core set of configuration management practices as a baseline for a software process improvement effort. Third, the identification of these principles and practices can motivate others to explore opportunities for automating these practices, as well as automating their verification.

2 Principles

There are ten basic principles that support configuration management activities.

Principle 1: Protect critical data and other resources. The process of developing software produces many artifacts. Some of these artifacts include the definition of requirements, design specifications, work breakdown structures, test plans, and code. All of these artifacts generally undergo numerous revisions as they are created. The loss of such artifacts and their revisions can cause great harm (e.g., financial loss,

schedule slip) to an organization. Thus, it is vital that these artifacts and their interrelationships be reliably maintained. This implies that these artifacts are always accessible to consumers or quickly recoverable when failure does occur.

Principle 2: Monitor and control software development procedures and processes. An organization should define the processes and procedures that it uses to produce artifacts. Such definition will provide a basis for measuring the quality of the processes and procedures. However, to produce meaningful measures of the processes and procedures, the organization must follow them. Consequently, the organization must monitor its practitioners to ensure that they follow the software development processes and procedures.

Principle 3: Automate processes and procedures when cost effective. The automation of processes and procedures has two primary benefits. First, it guarantees that an organization consistently applies them, which means that it is more likely to produce quality products. Second, automation improves the productivity of the people that must execute the processes and procedures because such automation reduces the tasks that they must perform, which permits them to perform more work.

Principle 4: Provide value to customers. Three issues ultimately affect the success of a product. The first one is that a product must reliably meet the needs of its customers. That is, it must provide the desired functionality and do it in a consistent and reliable manner. Second, a product should be easy to use. Third, an organization must address user concerns and issues in a timely manner. All three of these issues affect customer value, and a configuration management tool should automate those practices that provide the greatest value to its user community.

Principle 5: Software artifacts should have high quality. There are many measures of product quality. Such measures attempt to identify several qualities of a product, such as its adaptability, efficiency, generality, maintainability, reliability, reusability, simplicity, and understandability. Adaptable products are easy to add new features or extend existing ones. Efficient products run faster and consume fewer resources. General products solve larger classes of problems. Maintainable products are easier to fix. Reliable products perform their intended tasks in a manner consistent with user expectations. Reusable products fulfill the needs of many tasks. Simple products are easier to understand, adapt, and maintain. Furthermore, simple products are often elegant and efficient. Understandable products permit developers to easily alter them. That is, products that are difficult to understand tend to be poorly designed, implemented, and documented. Consequently, they generally are less reliable. In sum, quality is not simply a measure of the number of defects that a product has. Instead, quality is a broad characterization of several qualities, characteristics, or attributes that people value. Thus, several quality measures of software artifacts must be continuously taken.

Principle 6: Software systems should be reliable. Software systems should work as their users expect them to function. They also should have no significant defects, which means that software systems should never cause significant loss of data or otherwise cause significant harm. Thus, these systems should be highly accessible and require little maintenance.

Principle 7: Assure that products provide only necessary features, or those having high value. Products should only provide the required features and capabilities desired by their users. The addition of nonessential features and capabilities that provide little, if any, value to the users tends to lower product quality. Besides, an organization can better use the expended funds in another manner.

Principle 8: Software systems should be maintainable. Maintainable software systems are generally simple, highly modular, and well designed and documented. They also tend to exhibit low coupling. Since most software is used for many years, maintenance costs for large software systems generally exceed original development costs.

Principle 9: Use critical resources efficiently. Numerous resources are used or consumed to develop software, as well as by the software products themselves. Such resources are generally scarce and an organization should use them as efficiently as possible.

Principle 10: Minimize development effort. Human effort is a critical resource, but one that is useful to distinguish from those that do not involve personnel. The primary motivation to efficiently use human resources is to minimize development costs. In addition, the benefits of minimizing the number of personnel used to develop software increases at a greater than linear rate.

3 Practices

Twenty-three fundamental configuration management practices support these ten principles. These practices fall into four primary groups: those that govern the management process, those that affect product quality, those that protect the primary work products, and those that guide how people should use a configuration management tool.

3.1 Management Practices

Seven key management practices enhance the success of a configuration management system.

Practice 1: Maintain a unique read-only copy of each release. After each release, the configuration manager should label the entire code base with an identifier that helps to uniquely identify it. Alternatively, a configuration management tool can automatically name each release using one of many defined schemes. By protecting critical information in this manner, software engineers can easily identify artifacts that an organization used to produce a release. It also prevents software engineers from altering source artifacts and derived work products specific to a release after deployment.

Auditors should verify that an organization has labeled its releases and made them read-only, although it may be difficult to verify that the organization properly captures the correct version of each required artifact. Further, recreating a build and successfully executing a regression test suite does not guarantee that the build is identical to the original build. On the other hand, a configuration management tool can maintain a unique copy of each release and track subsequent changes to it or prevent such change from occurring.

Practice 2: Control the creation, modification, and deletion of software artifacts following a defined procedure. A defined procedure should identify every item that an organization uses to make every work product (e.g., compilers), regardless of the type of work product – code or documentation. Control procedures should also identify who can create, alter, and delete a software artifact and under what conditions. Further, each transaction involving a software artifact should be recorded. During each transaction, metrics should be collected for each type of artifact. These metrics should attempt to measure all quality attributes of Principle 5. For example, an organization should record the number of changes to each component and how many of those changes are related to defects.

A defined procedure for controlling software changes has several benefits. First, it helps to eliminate rework. In a poorly managed organization, programmers often create multiple versions of the same artifact in a disorganized manner and have difficulty recovering desired versions of artifacts. Second, it improves the predictability of performing successful software builds and deliveries by focusing development efforts on planned changes. Third, it encourages the production of software having the greatest return on investment. That is, only feature additions and defect repairs that best satisfy user needs should be approved. Fourth, a controlled procedure provides management insight into the progress of a project team.

Auditors should identify that a defined procedure exists and that personnel follow it. In addition, auditors should verify that reports are generated on a periodic basis and that management examines and acts on the reports in a prudent manner. An organization should use automated methods to generate such reports, define software processes, and validate adherence to them.

Practice 3: Create a formal approval process for requesting and approving changes. A formal approval process should identify who has responsibility for accepting a change request and allocating the work. It should also identify the evaluation criteria that determine the requests that an organization will perform, as well as how it prioritizes them. The approval process should require the requestor to produce documentation that the development or maintenance team may need, the reason for the change, and a contingency plan.

Controlling the process of requesting and approving change requests is the primary way to inject stability into software development efforts. The benefits of controlling change are that an organization can ensure that it adds only necessary or beneficial features to a system. It also allows the organization to prioritize changes and schedule change in the most efficient or practical manner.

Auditors can examine the configuration management plan of an organization to detect that an organization defines a change management procedure. A configuration management tool may also embody such a plan, model the process, and partially automate it.

Practice 4: Use change packages. A change package defines a unit of work, whether it is the repair of a defect, the addition of a new feature or capability to a system, or an original development activity. Consequently, a change package should be traceable to a planned activity of a work plan or schedule. If it is not, the schedule is not an accurate reflection of the work a team is performing or when the team is performing it. The benefit of using change packages is that they aggregate collections of individual, yet related, changes, which helps to control the instability within a software system.

Change packages can be used with or without automated support. Auditors can verify the manual use of change packages by examining the documents that describe such change packages. However, such manual use is doubtful to occur since it is not practical. Instead, support for change packages will almost surely be provided by a configuration management tool. Auditors can easily verify whether a configuration management tool provides such a capability by examining its documentation and an organization's use of it.

Practice 5: Use shared build processes and tools. It is rare that individual members of a project team use different build processes or tools. When they do, the results are often inconsistent and difficult to debug. Thus, an organization should avoid such an approach. By controlling the build processes and tools, an organization encourages all its members to produce the same work products in the same manner. An additional benefit is that members can assist one another in tool use, which helps to reduce training costs.

Auditors can easily identify if an organization follows this practice with or without an automated aid. However, the use of a full-function configuration management tool using standard build specifications will ensure that an organization follows this practice.

Practice 6: A version manifest should describe each software release. A version manifest should identify all components that comprise a release, all open and closed problems, the differences between versions, relevant notes and assumptions, and build instructions. Since a version manifest explicitly identifies the specific version of each artifact that makes up a release, it permits a software development organization to better maintain and deploy systems.

Auditors can verify that a version manifest exists for and accurately reflects each release. The verification of whether a manifest accompanies each release is trivial, but validating its accuracy is not. Thus, it is best to permit a configuration management tool to generate manifests and use auditors to validate their generation or the underlying generation process. Verifying a manual process, on the other hand, is an extraordinarily tedious process.

Practice 7: Segregate derived artifacts from source artifacts. Source artifacts are works created by an author. Derived artifacts are those artifacts that result from processing source artifacts. For example, a binary object file is the result of compiling a source file written in a programming language. Work products, on the other hand, may be source artifacts, but they may also be derived objects. Work products are those products that an organization delivers to a customer or some end user or uses to develop a product. Delivered work products are those work products actually delivered to a customer, whether they are source or derived objects. If possible, an organization should separate these different categories of artifacts to ease their management.

A configuration management tool does not necessarily have to maintain derived artifacts using version control because it can reconstruct the derived artifacts from the source artifacts. The main reason for maintaining derived objects is to reuse their intermediate results computed by one user for another. For example, some configuration management tools use smart compilation, which greatly reduces compilation efforts by not recompiling object files when suitable ones (i.e., ones equivalent to the ones that a compiler would reproduce for the given environment) already exist.

Segregation of artifacts allows an organization to limit the scope of software management activities if it chooses not to manage derived artifacts that are not delivered work products. Segregation also isolates artifacts that tend to be large or expendable, which simplifies storage management functions and allows them to be efficiently used.

Auditors can easily verify the segregation of source and derived artifacts by various means, but a configuration management tool should automate this practice.

3.2 Quality Practices

Seven key practices ensure quality of the configuration items of a configuration management system.

Practice 8: All source artifacts should be under configuration control. The quality of delivered work products derived from source artifacts not under configuration control is suspect. Further, such a situation

does not permit traceability within a configuration management system, which means that impact analysis cannot be reliably performed. The benefit of placing all source artifacts under configuration control is that an organization can better control the development process, as well as better maintain traceability between artifacts.

Auditors can verify the artifacts that an organization maintains under configuration control and it can identify those artifacts not maintained under configuration control by examining how releases are built and generated.

Practice 9: Use a change control board. Feature additions are generally expensive to implement. Thus, projects must add functionality that end-users or the marketplace determines to be essential or have high value. A change control board controls the features of a system by explicitly and consistently considering the performance, quality, schedule costs, and benefits of each proposed change. Thus, a change control board reviews suggested changes, determines the ones to accept, prioritizes the accepted requests, and assigns the implementation of each one to a specific software release. To make the best decisions, the change control board should represent a cross-section of people from the client, development, and user communities. For example, project management, marketing, development, quality assurance, documentation, and user support personnel should participate on change control boards.

Auditors can verify the adherence to this practice by verifying the existence of a change control board and assessing whether it follows the guidance described in a configuration management plan. It is possible for a configuration management tool to record the decisions of a change control board, as well as the rationale for each one. A configuration management tool could even help formalize the decision process.

Practice 10: Build software on a regular, preferably daily, basis, followed by immediate invocations of regression test suites. Increasing the frequency of software builds reduces the number of changed artifacts between builds, which reduces the number of potential flaws, or unwanted interactions, of each build. Consequently, by increasing the frequency of software builds, an organization reduces the effort to find such incompatibilities since fewer artifacts are likely to have changed.

This simple process produces two significant benefits. First, it minimizes integration risk by keeping integration errors small and manageable. Second, it improves quality by preventing a system to deteriorate to the point where time-consuming quality problems can occur. When an organization builds and tests a product every day, it is easier to identify new defects in a product, which helps to minimize the debugging effort.

Each software build should produce a version manifest including the time of a build. Both people and programs can inspect the sequences of manifests to determine that an organization is performing regular software builds. Even better, a configuration management tool could automate the execution of daily builds and regression test suites.

Practice 11: Document identified software defects. An organization can categorize software defects in many ways: by type, source, and severity of failure, by the source of detection, and so on. The documentation of identified software defects helps to identify the quality of a product. It also permits an organization to identify the artifacts that may be inherently flawed and it may need to replace.

Auditors can verify that an organization documents software defects, although it may have difficulty identifying that it consistently does so. Similarly, a configuration management tool can report the defects that an organization has added to it, but it cannot guarantee that its contents are complete.

Practice 12: Software artifacts that comprise a release should adhere to defined acceptance criteria. Typical acceptance criteria should include various metrics. For example, code metrics may include the use of code coverage criteria, various complexity metrics, and various sizing metrics. Most artifacts will have acceptance criteria that include a review or test process, if not both of them. Adoption of this practice will ensure that all artifacts adhere to a defined level of quality.

Both people and automated methods can determine that artifacts meet acceptability criteria. However, without automated aid, it will be difficult for auditors to do this. Therefore, a configuration management tool should automate the execution of acceptance criteria.

Practice 13: Each software release should be regression tested before the test organization receives it. A regression test suite should strive to achieve complete code coverage, complete testing of exception conditions for each software component, and thorough testing of boundary conditions for all actual parameters of each function, and conduct complete functional tests derived from system requirements. Adhering to this practice helps to ensure that old code works roughly as well as before, enhancing quality and reliability.

If the regression test suite produces a record for each test case then both people and programs can verify that an organization executes a regression test suite. Examining the test records for correctness, without a standard protocol, can be extremely challenging for humans and impossible for a program.

Practice 14: Apply defect repairs to every applicable release or ongoing development effort. Many software development organizations have various releases in use by their end users. These releases may be necessary to operate within various different environments composed of varying operating systems, database systems, windowing system, or other utilities. Additionally, some releases may update older releases but in the same operating environment. Regardless, each of these releases and internal development efforts may contain the same defect and the person repairing the defect should ensure that each release and internal development effort either does not have the defect or receives the repair. Alternatively, a change control board or another entity may decide which releases and internal development efforts should receive the repair. This practice simultaneously eliminates defects in multiple releases, yielding customer benefit. It also benefits ongoing development efforts, which improves personnel productivity and future product quality.

Auditors can verify that an organization follows this practice by examining several change requests and determining whether developers modified multiple releases and internal developments. If not, it can examine the organization's change management procedures. Alternatively, a configuration management tool could support such an activity by assisting the person or group who authorized the work related to a reported defect.

3.3 Protection Practices

Four key practices enhance the reliability of the configuration management activity.

Practice 15: Use a software system to perform configuration management functions. By definition, a configuration management tool provides several basic features. First, it maintains a history of each software artifact (i.e., version control). Second, it maintains configurations of larger aggregations of these artifacts (i.e., configuration management). Third, it generally provides a utility to build derived artifacts, which are usually executable programs (i.e., build management). In combination, these three features permit an organization to capture a snapshot of each release, whether major or minor, as long as it needs, which ensures that releases can be built and modified to satisfy future needs. Fourth, a configuration management tool sometimes provides a mechanism to manage software changes. Such changes may be to correct software defects or to enhance software systems. Fifth, a configuration management tool can automate the deployment of software by maintaining an accurate record of its customers operating environments and the software they are entitled to use. This last capability has yet to become commercially available.

Furthermore, a configuration management tool should manage various kinds of artifacts and interrelationships among the artifacts. Examples of such artifacts include requirements, specifications (e.g., architectural), plans (e.g., test), user documentation, and training materials. Maintenance of artifact interrelationships makes it easy to perform impact analysis.

The use of a configuration management tool to control software development activities is beneficial for three reasons. First, it provides permanence for each software release, as well as a recovery mechanism for software artifacts. Thus, it helps to protect critical information. Second, it can enforce institutional policies, procedures, and processes. Third, it can automate the deployment of software, and ensure the validity of such deployment. Auditors can easily verify the use of a configuration management tool, but may have more difficulty identifying whether the organization effectively uses it in daily operations.

Practice 16: Repositories should exist on reliable physical storage elements. An organization can enhance the reliability of a repository by using mirrored disk drives, RAID-5 drives, redundant networks, and clustered servers. Adoption of this practice will yield highly available solutions and reduce the potential for significant loss of work products, which requires time and energy to replace. Auditors can verify this practice by examining an organization's configuration management plan and verifying its implementation.

Practice 17: Configuration management repositories should be periodically backed-up to non-volatile storage and purged of redundant or useless information. Several types of backups are possible (e.g., full and incremental). A configuration management plan must define the types of backups a software team will

use, when an operator will perform each type of backup, the tape naming conventions for each one, and the manner of storing each backup. Backups may be stored remotely or locally and protected in various ways (e.g., fireproof vaults). Regular backups ensure the reproducibility of software guarding against data and program loss. In addition, removing information no longer needed uses less computer storage and makes it easier to find information.

Auditors can verify that reliable media, such as magnetic tapes, contain software backups and that the software backups are properly managed. In addition, both auditors and programs can detect the existence of backup logs, as well as examine them.

Practice 18: Test and confirm the backup process. Most organizations backup their repositories, but seldom restore them. Thus, they have faith that the repositories will be accurately restored, but seldom validate that they are correctly saved and restored. This is a critical flaw in most configuration management processes. The benefit of performing this practice is to ensure that backups accurately capture the contents of repositories and that the restoration process can fully and accurately restore them.

Auditors can validate this process by conducting interviews with development team members. However, a configuration management tool could more effectively validate the capture and restoration of repositories.

3.4 Tool Practices

Five key practices support the practical use of configuration management tools.

Practice 19: Check code in often. This practice should be constrained when checking in code on the primary development branch. That is, developers should only check in working versions of code to a development branch. The frequent check in of code helps to eliminate the loss of a large change. That is, this practice ensures that software losses, if they should occur, will be small. This practice also permits individuals to synchronize their work with the most recent incarnation of the work of others.

People and programs can trivially verify this practice. However, a configuration management tool can automate it following the specifications that a project manager provides.

Practice 20: Configuration management tools should provide patch utilities. A patch facility deploys the equivalent of a new release by modifying an existing release. The benefit of a patch mechanism is that incremental releases can be deployed using telecommunications equipment much quicker. This is especially beneficial to users that use slow communication mechanisms, such as dial-up modems. The importance of this is that by providing such a patch mechanism, an organization will provide better service to its customers.

Auditors can verify the development of patch releases, although they will have difficulty verifying the correctness of such releases. Consequently, an organization should use an automated method for generating release patches, and an audit team should verify the automated process.

Practice 21: Do not work outside of managed workspaces. A configuration management tool can only manage artifacts within managed workspaces. In addition, configuration management tools generally use workspaces to facilitate communication among developers working on related tasks. That is, each developer can examine what is happening in the workspaces of others.

Thus, by working outside of managed workspaces, several problems can arise. People that develop artifacts outside the workspace cannot easily share them with others. People that develop artifacts outside the workspace also do not have access to the automation functions provided by a configuration management tool, such as the automated generation of reports. Consequently, working within a managed workspace eliminates many problems, which reduces development effort.

Auditors could have difficulty verifying that a software organization adheres to this practice, although it can interview software engineers to verify it. Unfortunately, no program can verify this practice.

Practice 22: Do not share workspaces. A workspace should have a single purpose, such as providing a build and test area for a single developer or for a product release. When sharing a workspace, the actions of one person may adversely interact with those of another. By not sharing workspaces an organization can avoid such development problems, which reduces development effort.

Auditors could have difficulty verifying that a software organization adheres to this practice, although it could conclude one way or the other based on interviews of software engineers. On the other hand, a configuration management tool could enforce, and verify, this practice.

Practice 23: When developing software on a branch other than the primary development branch, regularly synchronize development with the development branch. A person's work in a team environment depends on the artifacts that others develop. Consequently, programmers should integrate the mainline changes of others into their workspaces on a periodic basis. Infrequent integration will lead to integration difficulties.

Adoption of this practice ensures that the development efforts of the entire team will be compatible with one another on a periodic basis. This has the effect of reducing the effort required to fix incompatibilities to manageable units of work. In addition, it prevents the surprise of having unanticipated, large integration efforts from occurring immediately before a software release. In other words, it permits managers to manage the development process by making several small schedule changes, if needed, instead of creating a few large, possibly unpredictable, perturbations.

Verification of this process is very easy to achieve by an auditor, although a configuration management tool could provide such functionality.

4 Relationships between Principles, Practices, and the Software Lifecycle

Of the ten configuration management principles, Principles 1 through 6 are the ones of primary importance. Of these, Principle 1 is by far the most important because it guarantees that the artifacts that products are derived from remain intact. Principles 2 through 6, on the other hand, help to keep customers because these principles focus on product quality and customer satisfaction. Consequently, the practices associated with Principles 1 through 6 are of primary importance, whereas the practices associated with Principles 7 through 10 are of secondary importance. Table 1 helps to distinguish between these two sets, as well as the four categories of practices.

		Practices																						
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Principles	1	<input checked="" type="checkbox"/>														<input checked="" type="checkbox"/>								
	2		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>						<input checked="" type="checkbox"/>								
	3										<input checked="" type="checkbox"/>					<input checked="" type="checkbox"/>								
	4																				<input checked="" type="checkbox"/>			
	5										<input checked="" type="checkbox"/>													
	6													<input checked="" type="checkbox"/>										
	7									<input checked="" type="checkbox"/>														
	8						<input checked="" type="checkbox"/>																	
	9																		<input checked="" type="checkbox"/>					
	10										<input checked="" type="checkbox"/>				<input checked="" type="checkbox"/>							<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Table 1. Traceability matrix between principles and practices.

The following discussion identifies when an organization should perform each primary practice, as well as how it should perform each one. If existing technology already exists that supports a practice then selected supporting products are discussed. Finally, if new technology is needed to perform a practice, or can be developed to aid it, then such technology is identified.

During the project planning phase, an organization should perform three important tasks. First, it should acquire highly reliable and redundant physical storage and processing elements for the software repository. Second, it should acquire a configuration management tool that is able to document identified software

defects and produce release patches. Several such systems exist today, although few, if any, commercial versions support the production of release patches. Third, it should establish a change control board that will define a configuration management process and rules for approving changes to artifacts.

During all development and test phases of the lifecycle, an organization should perform five critical tasks. First, it should control the creation, modification, and deletion of all source artifacts using the configuration management tool and defined procedures and rules identified during the project planning phase. Second, it should regularly back-up configuration management repositories to non-volatile storage and periodically purge them of redundant or useless data. Such backups can be automatically performed by writing a trivial program or script. Third, it should periodically verify that the backup process functions properly. This is easily achieved by writing a simple program, but seldom, if ever, done. Fourth, it should document artifact defects using the configuration management tool. Fifth, it should produce read-only copies of each release.

During the programming phase, an organization should follow four practices. First, it should build and test software on a regular and frequent basis. A configuration management tool should automate such a process, although none currently does. Second, an organization should use change packages to define collections of related changes. Third, its software engineers should frequently save incremental, working code changes in the repository. Fourth, when its software engineers fix one version of a software system, they should fix all other released versions, including the one undergoing development.

Note that most of the primary practices can be achieved using existing configuration management tools. In situations where existing configuration tools do not directly support these practices, there are generally mechanisms for an organization to easily add them through the augmentation of simple programs.

All secondary practices should be exercised during the programming phase and enforced, when possible, during software audits. These practices ask that programmers include a manifest with each software release, separate source and derived artifacts, work within managed workspaces, do not share workspaces, and regularly synchronize development. The first two practices can be automated by configuration management tools, whereas the later three must be manually performed.

5 Capabilities of Configuration Management Tools

The preceding principles and practices, along with the basic definition of a configuration management system, can help to identify the capabilities that a configuration management tool should provide. These capabilities can be used to analyze and compare various products or be used as requirements for the design and implementation of a new product. The capabilities of a configuration management tool can be grouped into several important categories: version control, configuration control, model control, build control, change control, deployment control, process control, security control, and user interface control. This section will describe each of these categories and the associated capabilities.

5.1 Version Control

The responsibility of the version control grouping is to maintain a collection of versioned artifacts. Each versioned artifact is distinguished and new versions of artifacts may evolve. These two characteristics are the first two capabilities required of the version control module of a configuration management tool.

Capability 1: Maintain artifacts in a persistent repository. An arbitrary number of objects should be representable in a configuration tool. Furthermore, each artifact must be persistent, implying that each one exists across user sessions of the configuration management tool.

Capability 2: Maintain unique versions of artifacts as they evolve. Each artifact should allow multiple versions of each artifact to be represented such that each version is always derived from another version, with the exception of the root version. Hence, the sequence of artifact versions are ordered and time dependent.

Capability 3: Permit parallel derivations of artifacts to evolve. Several versions of an artifact can be derived from the same artifact. Thus, versions of an artifact can be represented as a tree.

Capability 4: Allow the merging of two or more versions of an artifact. Since several versions of an artifact may exist, one may want to merge one or more of them into a newer version. This means that the

newer version will be composed from pieces of each merged object. That is, the new version is a shared representation derived from all the merged objects.

Capability 5: Allow the representation of relationships among artifacts and versions thereof. Representing the relationships between artifacts helps people to better understand a system of artifacts. For example, one artifact could represent a functional requirement and another a design component. An explicit relationship needs to exist between these two objects if one is derived from the other, or conversely traceable to the other. That is, in some case unary relationships may be sufficient, but other situations require binary relationships. It may even be possible that arbitrary n-ary relationships are needed.

Capability 6: Define and maintain several key relationships and attributes among artifacts and versions thereof. Examples of common relationships include authorship, genealogy, and membership. Examples of common attributes are creation time, unique identity, and logical name.

Capability 7: Permit the manipulation of the contents of a version control repository. Specifications should permit the use and manipulation of any artifact, artifact version, or relationship.

Capability 8: Provide a sort capability for all retrieved data using one or more attributes or relationships. Sort specifications provide a useful ordering of the retrieved information, which makes information processing and presentation easier.

Capability 9: Permit asynchronous access to each version of an artifact. Multiple people often need to read and write the same artifacts to enhance individual productivity. In other words, people postpone coordination issues until a more appropriate time, which is when artifacts are merged together.

Capability 10: Allow actions to be performed before and after each event. Events correspond to the creation, modification, and destruction of an artifact or artifact version, as well as the merging of several artifact versions. For each type of event, it may be useful to cause other actions to be invoked, such as the recording of an activity in a log, the computation of relevant metrics, or the realization of a process workflow.

Capability 11: Permit the identification of the differences between any two versions of the same artifact. A system should be able to ignore selected characters during match operations, such as carriage returns and linefeeds.

5.2 Configuration Control

The responsibility of the configuration control grouping is to maintain collections of aggregated artifacts that form larger systems and subsystems.

Capability 12: Provide a means for specifying the artifacts of a configuration, which is just another kind of artifact. A configuration must somehow specify its components, which are typically primitive artifacts and other configurations. A configuration should also specify the programs that will be used to make any derived objects. It may be useful to specify these components as artifacts under version control.

Capability 13: Provide a means for specifying the relevant versions of each artifact corresponding to a configuration, which is sometimes called a view. As described in the previous section, the derivation history of an artifact can form a directed acyclic network where each node in the network corresponds to a version. The view specification must permit one of these versions to be selected from the many. Specification methods usually allow a version to be selected by its creation time, by an associated label name, or a branch name. A label name is generally provided by a user to name a distinguished version of an object. Similarly, a branch name is the name of a specific arc within the network, which specifies that the selected version must come from its subnet. In some cases a specific version can be indicated by its unique identifier, which is generally a number or sequence of numbers.

Capability 14: Provide several standard views. Typical standard views represent the latest version of the software, the current version undergoing quality assurance testing, and the last released version.

Capability 15: Allow repositories to be pruned and compacted. This capability should use a specification provided by a configuration manager. Such a specification would specify the rules that the configuration controller would use to prune and compact a repository. The reason that this is a capability of the configuration controller and not the version controller is that it needs to understand the artifacts that make up individual configurations before it can delete any artifact versions. In other words, the version controller would not know what objects it could or could not keep since that information is kept in the configurations. Note that this assumes that you want to keep or remove entire configurations.

5.3 Model Control

The responsibility of the model control grouping is to provide a domain model for software artifacts.

Capability 16: Manage artifacts at the concept level. Most, if not all, configuration management tools manage software artifacts as unstructured textual objects. Instead, artifacts should be represented as structured concepts, where the model controller represents and understands the artifacts at a finer level of granularity. The model controller should distinguish between individual requirements, component designs, functions, variables, test cases, etc. It should understand the unique attributes and behaviors of each one. For example, the model controller should know that functions can call other functions and that component designs can be traced to individual requirements. In fact, by understanding these relationships, the model controller can expect that these relationships should be maintained and can notify users when they are not.

Capability 17: Define and maintain a domain model characterizing standard software artifacts, such as requirements and design artifacts. All classes of objects created and used during the entire software lifecycle should be modeled by the model controller. This includes not only the technical aspects of software development, but also the managerial aspects. Thus, work breakdown structures and schedules should be modeled by the model controller, as well as many other concepts.

Capability 18: Allow extension of the domain model by permitting the definition of new artifacts. Such definition can be accomplished in a declarative or procedural manner. This extensibility will accommodate new specification languages, techniques, and methods.

Capability 19: Provide metrics for each type of artifact. For code, these metrics should include total lines of code being managed, lines changed, inserted, and deleted, most frequently changed artifacts, and periodic summaries of changes on a monthly, quarterly, and annual basis. For project schedules, these metrics should include the number of tasks completed, the number of tasks completed on time, the number of tasks still outstanding, the number of tasks behind schedule, the number of tasks being worked, and the number of tasks suspended. In addition many more metrics are required for these two objects, as well as for all those not mentioned.

5.4 Build Control

The responsibility of the build control grouping is to construct derived artifacts from source artifacts.

Capability 20: Support build and release processes. Many existing configuration management tools use native build tools, which can make it difficult to migrate software from one environment to another. Instead, the build and release process should be independent of native build utilities. More importantly, the build and release procedures should be tailored to the individual concepts modeled by the model controller.

Capability 21: Support the building of an entire system or its subsystems or components. The tool should be able to build any object within the repository in the same manner.

Capability 22: Permit the identification of required components external to a configuration and include as part of a build record. External processors are required to produce some derived objects and should be part of the build record so that it can easily be determined what tools were used to produce a build. Furthermore, by identifying required components as first-class concepts of a build record, it will be easier to migrate a system to a new environment and rebuild it without editing build scripts or some other form of a build specification. That is, by creating references to objects that correspond to external processors, when a system is migrated to a new environment those references will be resolved in the new environment if they exist. If they do not, the build controller should notify the user of their absence.

Capability 23: Distinguish between source and derived objects, and link derived objects to their sources. The build controller should automatically link derived objects to their source objects at the conclusion of a build operation. Distinction should be accomplished by typing the source and derived objects according to the domain model.

Capability 24: Provide an accurate and complete itemization of all artifacts built at the end of a build activity and include as part of a build record. Such a build record should identify the external and internal processors used to build each object, the dependencies of each object, and the order in which individual objects were built. In sum, it should provide a record that could be later replayed to rebuild the system in exactly the same way.

Capability 25: Permit the execution of a regression test suite, as well as other computations, at the conclusion of a build. Associated with a build specification should be an indicator or a set of rules that identifies whether a regression suite should be executed after a build operation.

Capability 26: Support parallel build processes. If a machine has multiple processors then the build controller should permit multiple components to be built at the same time.

Capability 27: Support distributed build processes. If multiple machines are available to build a software system then components should be distributed to the other machines, where they are built, and then returned to the build primary host.

Capability 28: Avoid unnecessarily rebuilding components. That is, use intelligent compilation. Some configuration management tools capture all the dependencies that were used to produce a derived artifact. Then, when another build operation of an artifact occurs that has the same dependencies then the build controller reuses the derived artifact instead of rebuilding it.

5.5 Change Control

The responsibility of the change control grouping is to handle all types of change requests and monitor them to closure.

Capability 29: Manage the change process – from initiating to completing an enhancement or defect correction. This implies that the status of a change is always known, as well as there being a model of the legitimate status values. Further, it implies that there is a model of how a change request can change state and what actions are achieved or performed to transition it between states.

Capability 30: Support an automated change authorization and approval process. It is possible for a configuration management administrator to specify rules that would automate the authorization and approval process. Such rules may use attributes of a change request and other derivable measures. For example, it is possible to automatically approve all severe reported defects or defects related to complex components, as determined by a complexity metric.

Capability 31: Support the management of a group of related changes, sometimes called changed packages or change sets. This focuses on identifying the reasons for change in addition to cataloging a collection of interrelated changes. In other words, a change package or set groups a collection of related changes into a larger, all or nothing, atomic transactions.

Capability 32: Support hierarchical change packages. The reason for supporting hierarchical change packages is to create a one-to-one correspondence with a work breakdown structure. By doing so, it is possible to automate the creation of change packages, as well as their scheduling and resource assignment.

Capability 33: Support change packages that can terminate based on an event. For example, if a task of a work breakdown structure is eliminated due to a requirements change then the change controller should eliminate the corresponding change package. Further, the consequences of such action should propagate throughout the design and implementation, which may eliminate one or more planned components.

Capability 34: Save change packages and sets as atomic transactions. Since a change package represents a coordinated set of changes they should be added to the repository as a single, all or none, transaction. Of course, hierarchical change packages should be saved as nested transactions.

Capability 35: Identify the key attributes and relationships of change requests and problem reports. Some of these attributes include a priority, a severity, a status, a creation time, an author, the lifecycle phase where the request or report was created, and the category of the change request or problem report. In addition, application information should include the application name, release number, computer manufacturer and type, operating system, and revision number. Furthermore, the people assigned to the task, as well as those that will review the changed artifacts, should be identified.

Capability 36: Provide metrics for tracking change requests. Specifically, track the number and frequency of defects, changes over time for each artifact, and statistics regarding the length of time to close change requests.

Capability 37: Provide a common set of forms for managing change requests (anomaly or enhancement) and problem (defect) reports. These forms will provide an intuitive interface that help to constrain data input.

Capability 38: Permit the definition and use of alternative forms for managing change request and defects. The change controller should allow an organization to tailor it to use its standard input forms.

5.6 Deployment Control

The responsibility of the deployment control grouping is to automatically deploy software to customers or notify them when appropriate releases are available for deployment.

Capability 39: Deploy releases to users and record their deployment environments. The deployment controller must keep records of deployment environments so that when a new release is available it can determine whether it can be deployed to an existing environment. If not, it should be able to identify what incompatibilities exist in the environment and notify the customer of those incompatibilities. Once the incompatibilities are corrected, the customer can notify the deployment controller, which can deploy the release to the customer.

5.7 Process Control

The responsibility of the process control grouping is to automate the workflow of the software development process and to ensure that practitioners of an organization follow the desired software process.

Capability 40: Support lifecycle models for managing artifacts. Organizations define software processes that specify how software professionals are suppose to conduct work. Sometimes organizations allow these processes to be tailored to meet the specific demands of a project. By supporting lifecycle models, the process controller ensures that the institutional or project software process is followed. The phases of a lifecycle model must permit the definition of entry and exit conditions. Furthermore, different lifecycle models should be available for different artifacts. That is, applicable lifecycle models should be based on the source concepts defined by the domain model.

Capability 41: Provide common lifecycle models for managing artifacts. Two such lifecycle models follow. One is defined by development, alpha test, beta test, and release phases, whereas the other is defined by development, unit test, system test, acceptance test, independent verification and validation, and release phases. Regardless of the number of lifecycle models, one of them should be the default lifecycle model.

Capability 42: Support the capture of metrics regarding length of time spent in various phases of an artifact lifecycle. More accurate measurement of the type of work being performed can be achieved by monitoring the type of artifact having the current focus. When the focus changes, the session time of the previously focussed artifact is accumulated. Since this assumes that the artifact that has the focus is being worked, a process can note when inactivity occurs for five minutes or so and cause the artifact to lose focus and no new one to gain it. This approach is flawed but still is better than a manual process of recording work activity.

Capability 43: The entry and exit criteria available for use by a process model should include various review, inspection, and test criteria. For example, an exit criterion for a code artifact is that a test suite be defined that satisfies a minimum level of statement and path coverage. An exit criterion for defining a requirement may be that it be written as an active sentence. An entry criterion for a component may be that it is associated with a requirement.

5.8 Security Control

The responsibility of the security control grouping is to ensure that users of the configuration management tool perform the operations that they have been granted and no more.

Capability 44: Provide access to artifacts via a role-based mechanism. The definition of roles and the assignment of capabilities to them provides a very useful concept for managing access to software tools. Typical roles include administrator, change control board member, developer, tester, project manager, development leader, owner, creator, and build manager. Example capabilities include who can establish and merge branches, read and write artifacts, and lock and unlock artifacts. Roles should be assigned to users and user groups.

Capability 45: Permit the creation of audit trails for artifacts. Another function of the security process is to monitor access to artifacts of a repository. The creation of audit trails enables such monitoring.

5.9 User Interface Control

The responsibility of the user interface control grouping is to provide a Graphical User Interface (GUI) to users that enhance their productivity and overall experience.

Capability 46: Provide configurable user interfaces. Users should be able to tailor the interface in a manner that most suits them.

Capability 47: Provide a unique GUI for each role. The GUI should emphasize the work activities that are normally performed for each role. Therefore, there may be unique menus and menu items applicable to each role and the layout of windows, palettes, toolbars may be different to emphasize the work activities of each one.

Capability 48: Graphically present the similarities and differences between two artifacts. Users should be able to graphically select two artifacts for comparison and visually distinguish the common and different elements of the two.

5.10 Miscellaneous

Finally, a configuration management tool should provide several other miscellaneous capabilities.

Capability 49: Operate within a heterogeneous computing environment, composed of various types of computers, operating systems, and repositories. Today, it is almost impossible to find a large project that uses a homogenous computing environment. Therefore, to be widely used a configuration management tool must operate within a heterogenous environment. Similarly, a configuration management tool must be able to operate within standalone and n-tiered environments, as well as within a geographically distributed computing environment.

Capability 50: Permit replicated repository and server updates to occur as a single distributed transaction or through a delayed delivery mechanism. Replication supports distributed development and also helps to provide high availability of the artifacts under configuration management.

Capability 51: Provide an Application Programmer's Interface (API). Organizations should be able to interface to a configuration management tool through an API to satisfy its unique needs.

Capability 52: Provide a command line interface (CLI). Some users prefer CLIs to GUIs and their needs should be accommodated.

Capability 53: The API, CLI, and GUI should be consistent. If a user learns how to use one interface, he should be able to immediately use that knowledge for another interface.

Capability 54: Allow the importation and exportation of artifacts using existing standards, including delimited text documents. At a minimum, a configuration management tool should be able to export and import arbitrary data in XML according to a defined format. For requirements and code, it is reasonable to expect a tool to import and export and import data in a delimited or fixed-field textual format. In addition, software designs should be able to be imported and exported in the Unified Modeling Language exchange format.

6 Summary and Conclusions

This paper identified ten principles and twenty-three practices applicable to configuration management. This paper differs from other *best practices* documents in several ways. First, it describes the rationales for each practice. These rationales primarily discuss the benefits of each practice. Second, the practices are traced to underlying principles, which have greater acceptance than practices. Third, the traceability matrix of principles and practices permits one to focus on those categories of greatest concern. Fourth, the traceability matrix also allows one to easily determine the coverage that selected practices have relative to principles. Thus, the provided information should permit the software professional to analyze these principles and practices in light of specific constraints to develop an optimal economic solution. That is, it is clear that economic issues dominate software development in the real world. Thus, the primary issue is determining what practices one can select that provide the greatest reward-risk ratio given the existing constraints.

This paper also identified fifty-four capabilities that a configuration management tool should have to support a configuration management system that adheres to the identified principles and practices. These capabilities can be used to evaluate existing configuration tools or they can be used as a basis for the requirements of a future configuration management tool. This was actually the original goal of the author, who thinks that most, if not all, configuration management tools do not adequately support software development. Two specific examples of this follow. First, most configuration management tools are file-based and do not capture artifacts at the level of granularity that is truly useful – individual requirements, design concepts, classes, functions, test cases, and so on. Second, most configuration management systems do not interoperate within an enterprise – the industry needs a configuration management tool that operates in every environment in the same manner.

In sum, this paper has identified principles and practices that differ from other sources. Most source documents, such as the CMM [5], CMMI [12], and ISO [13], [14] standards, requirements, and guidelines usually provide very general advice about the practices that an organization needs to perform. This paper, on the other hand, gives very detailed recommendations about what an organization needs to do. About half of these practices are discussed by these other sources, whereas the remaining half represents new ideas or at least ones that are not widely discussed. They can provide a useful foundation for a process improvement effort for the configuration management process area.

Acknowledgments

The author would like to acknowledge the competent and comprehensive comments made on an early draft of this paper by Bill Pardee.

References

1. Rochkind, Marc J.: The Source Code Control System. *IEEE Transactions on Software Engineering SE-1* (1975) 364-370.
2. Tichy, Walter F.: Design, implementation, and evaluation of a Revision Control System. *Proceedings of the Sixth International Conference on Software Engineering* (1982) 58-67.
3. Krause, Ralph: CVS: an introduction. *Linux Journal* **2001** (2001).
4. Babich, Wayne A.: *Software configuration management: coordination for team productivity*. Addison-Wesley (1986).
5. Paulk, Mark C., Weber, Charles V., Curtis, Bill, Chrissis, Mary Beth: *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley (1994).
6. Jones, Capers: *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley (2000).
7. Bersoff, Edward H.: *Software configuration management, an investment in product integrity*. Prentice-Hall (1980).
8. Whitgift, David: *Methods and Tools for Software Configuration Management*. Wiley (1991).
9. Berlack, H. Ronald: *Software configuration management*. Wiley (1992).
10. Buckley, Fletcher J.: *Implementing configuration management: hardware, software, and firmware*, 2nd ed. IEEE Computer Society Press (1996).
11. Rigg, William, Burrows, Clive, and Ingram, Pat: *Ovum Evaluates: Configuration Management Tools*. Ovum Limited (1995).
12. CMMI Product Team: *Capability Maturity Model Integration, Version 1.1, Staged Representation*. CMU/SEI-2002-TR-012 (2002).
13. International Organization for Standardization: *Quality management systems – Requirements*, ISO 9001 (2000).
14. International Organization for Standardization: *Quality management systems – Guidelines for performance improvements*. ISO 9004 (2000).