Untangling Configuration Management

To Deliver Quality Software

A Foundation Document

Author: Richard Wallace
Date: April 7, 2000
Contact: Richard Wallace, Quantum Solutions
richard.wallace@emendo-ex-erratum.org

Forward

There is considerable diversity in the Configuration Management (CM) methodology needs of today's software teams. The CM methods used by a team is a function of technical, social and corporate constraints that define how the project team must design, construct, test and deliver software.

Most commercial and academic CM systems/products created to date support particular CM methodologies. Some created specific mechanisms to support their methodology, while others simply support the methods that work given the constraints and limitations of their mechanisms.

If CM systems are to be applicable to the broad spectrum of software development teams, the methodologies must be separated from the mechanisms. The mechanisms must be distilled into a flexible set of widely applicable capabilities, and the usage of methodologies using these mechanisms must be defined.

This paper will first discuss the requirements of the mechanisms, then describe a range of methodologies that can be defined and supported using these mechanisms.

Introduction

Today's software teams span a broad range of languages, team sizes, development and delivery platforms, quality requirements and localization requirements. They often simultaneously support a number of prior releases, as well as new releases under development. Any given team has a unique set of environmental requirements that influence the optimal software development process and the CM process in particular.

Those working on software development team support have been investigating methods for enabling these teams and addressing the problems, they face. From these experiments and products have emerged several useful CM mechanisms and methodologies [3, 4, 5, 7, 8]. However, these systems have had limited success in four important respects. First, while a particular CM system might be a good match for one project team, the project teams down the hall often have substantially different requirements and are forced to change the way they operate; look elsewhere for a CM system; or develop something themselves. The lesson for those who design CM systems is to provide systems that support a wide range of needs, not just a particular methodology that is thought to have special value.

Most CM systems provide extremely limited automated methodology support [5]. They provide mechanisms that are useful when followed up with manual procedures and team discipline, but there is precious little support in the system itself. Perhaps the extreme example of this occurs when project teams assemble their own CM system by writing programs and shell scripts on top of standard version control systems. Any methodological support they put in is a level above the CM mechanisms, and therefore the mechanisms are not able to provide the automated support for these methods. The lesson here is that the CM system must consider the automated support of methodological requirements as an important mechanism.

The problem has to do with systems that just store the project files but do not manage the process of creating deliverable objects based on the source, or provide automated ways to construct configurations. This "build" process requires a considerably more robust notion of a configuration. While this may be the difficult aspect of implementing CM systems, this aspect is critical.

Additionally the problem of CM systems has been standing the test of time. While the project team may find a match with a CM system as they begin their project, their needs can change over time. The CM system can quickly become an obstacle in accommodating new requirements. This is often the case when the team grows in size or quality or delivery requirements change. A clear capability of modern CM systems that emerged through this experience is the ability to evolve the methodological support over time.

The remainder of this report will list the core CM mechanisms that are believed by the industry to be required to address these issues.

CM Mechanisms

CM mechanisms are defined as those capabilities supporting the implementation of CM systems.

Objects

In order to characterize what should be placed under control of a CM system, one must decide which activities will be covered by CM. A broad variety of activities can sensibly and beneficially participate in the CM process, including preparation of requirements and design documents, project planning and scheduling, task assignment and tracking, program coding and testing, the quality assurance process, defect and change tracking, and finally release generation.

From the above list we can derive an even longer list of artifacts which one may want to control, including requirements documents, design data (drawings), PERT charts, schedules, task descriptions, source files, program build procedures, object files, libraries and executable programs, test plans, test data, test results, quality measurements, user manuals, release descriptions, configurations, and product releases. These types of data include virtually every form of data that one would normally encounter in a modern computing environment including free-form and structured text, structured binary data, image data, databases, executable images, and various structured and unstructured collections of data of the preceding types.

This is not the entire picture either. All of the items have other properties (also referred to as attributes) which need to be stored and referenced, such as who created them, when they were created and last modified, how they are intended to be used, what level of approval they have reached, and what kinds of protections should apply to them. This information is often called "metadata," and may actually cover the same range of forms as the primary data.

In addition, each item may be related in various ways to other items under control, and these relationships should be stored.

In order to define a notion of "object" which will be general and flexible enough to encompass all of these different kinds of information, the following requirements are set forth.

An object and its metadata should be considered atomic with respect to overall configuration management. This means that to create, delete, copy, move, or otherwise manipulate an object generally includes the metadata as well.

Objects need not be atomic entities, but rather may be homogeneous or heterogeneous collections of parts. Further, each part may take on any of the forms available to object data.

Each object may fit into a classification such that common properties and behavior specified for the class can be used to perform operations on the individual object. The means by which an object acquires behavior from its class is called "inheritance", for consistency with Object-Oriented Programming concepts.

In spite of the above, each object instance may have unique character of its own, which overrides the class behavior. Further, new attributes of any type may be added to a given object at any time, unless purposeful access restrictions specifically prohibit this. Unlike popular database concepts used in traditional application development, this means the CM database should not be ruled by a fixed schema.

As is suggested above, a key requirement of objects (or more generally, their classes) is that they describe how to perform most generic or abstract operations upon themselves. Since many users, activities, and processes (programs) will interact with an object during its lifetime, it is important that the means of performing routine functions (such as creation, deletion, modification, changes of

state, or retrieval of primary or metadata) be transparent to the user or encoded into every application. Thus the notion that behavior be storable and associated with the class is fundamental to our object definition. For the sake of uniformity, stored behavior can also be considered as another type of metadata, and we refer to this kind of metadata as "methods".

While objects themselves are classified and have behaviors and properties derived from their classes, data members also have a data type and properties leading from that type, such as storage format (integer, string, text, binary) and location (the CM database, local file-system, remote repository).

Objects may participate in various "relationships" with other objects or with structured or unstructured groups (binary and n-ary relationships). Some relationships may be treated as properties (metadata) of one of the affected objects. For example, the "is a" relationship between an object and its class is a (static) property of the object. Other relationships may not be considered properties of the participants. An example of this is the participation of an object in a configuration. The configuration "contains" the object only by reference, since the same object may be used in a variety of places.

Objects need to be distinguishable and identifiable by some subset of their properties. Thus, we will require a name as a fundamental meta datum of all objects. This name will be unique within some bounds. One such bound should be the object class, since two similarly named objects of different classes are normally distinguishable. In order to keep names short and meaningful, the notion of a namespace will constitute the other boundary. As will be detailed shortly, the version is another meta datum that must be known to uniquely identify an object (in the absence of configuration context), and so a universal naming convention arises: namespace:class:name:version. Objects must be persistent.

Versioned Objects

We next turn to defining what is meant by the term "version", and how this relates to objects. We then examine the relationships that inherently exist among versions, in order to create a basis for discussing CM processes.

In informal discussions, we could talk about objects as being the same as versions of class instances (object = version), or we can consider objects to exist as one or more versions (object = set of related versions). For simplicity, the latter usage will be followed. In fact, we will not make a formal definition of objects, but instead only of versions. The pattern by which versions belong to the same object will be a matter of convention. As examples, having identical namespace, class, and name could be considered to identify all versions of the same object, or all versions leading from a particular one via lineage (the closure of successor and predecessor relationships) could serve this purpose.

It is a clear purpose of CM systems to store the version history of an object. However, what is not so clear is which of several kinds of relationships between versions should be involved in this history. Thus, a CM system should be able to show users all relationships of which it has knowledge. Which relationships are kept in the CM database is a matter of methodology.

Aside from the occasions when an initial version of a new object is created, there exist at least two important implicit properties of a version being created. The other version(s) from which the new one is derived; and the intended use of the new version.

The former is called the "derivation" relationship of the version, and one may say the new version is "derived from" the preceding one (its predecessor). If a merger process is involved in creating the new version, then this version may be considered as a derivation from more than one predecessor. It is important to track this relationship for two important reasons. First, it is needed for computing history of changes to the succession of versions leading to a particular one; and second, it allows storage compression mechanisms to operate more efficiently.

The "usage" of a derived version is more interesting from a methodological perspective. This relationship connects the new version to one or more previous versions that it is either intended or known to replace (in the sense of participation in configurations). In other words, we create a new version to "succeed" a previous one (usually because it contains a correction or enhancement to the predecessor), or to "replace" or "supersede" it (if the new one is considerably different, and not a change to the previous). In the latter case the new version may not actually be considered to belong to the same object, but still constitutes part of the predecessors' history.

Configuration Objects

Another fundamental requirement of a CM system is to provide the means to construct, manage, and control "configurations". The term configuration will be used rather loosely, but there are a few assumptions that we will make about the properties these objects must have.

There may be several types of configurations, and all will be first-class objects in the sense of possessing member data, behavior (inherited from the class or defined locally), and version history.

In departure from normal object properties, configurations will permit the aggregation of objects (possibly by containment, but certainly by reference), and will permit structural relationships among objects to be stored and navigated.

Configurations may form hierarchies such that one configuration "contains", or organizes (again by reference), a set of others just as it can do so for other types of objects.

The reason referential grouping is emphasized is that although containment is an intuitively attractive concept; the referential mode appears to support more capabilities required for CM purposes. This should become clearer in later sections that examine release management methodologies, but we will mention one key point here. It is the very nature of the CM (and software development) process that objects are continually reused and reorganized in ways that were not anticipated initially. Thus objects which have long been non-modifiable (for control reasons) may need to be included in new configurations, and of course in multiple configurations. Thus containment in a single configuration tends to be a less useful concept.

The question now arises as to what kinds of structures the configurations may need to represent. Some examples of configuration structures that naturally arise are:

- Sets (e.g. the set of all program-versions participating in a release)
- Groupings (e.g. the program versions a user maintains)
- Trees (e.g. the section/chapter/paragraph structure of a document)
- Directed graphs (e.g. the "make" dependency structure of a program)
- A network (e.g. the call graph among a class collection)

Depending on work habits, methodologies, or development tools, other structures may need to be represented. Thus, a good CM system should provide a rich set of configuration types and/or a general way of implementing new structures.

Considering the above list, a very general type of structure that may be able to represent most of the cases routinely arising in CM is the directed graph. We will shortly define a type of configuration which supports directed graph structures but can also be used to represent many other commonly needed object organizations.

At this point, we will further clarify why it is so crucial that a CM system support configurations as first-class objects.

To start with, a primary function of a CM system is to provide the means to define and control releases of a product (we use this term in the abstract). Consider the case of a software development shop. The goal of such an organization is to deliver software to a set of clients or

customers. Over time many versions of this software must typically be delivered. The software shop is expected to deliver versions with known functionality and quality.

This software is usually not a singular entity, but rather a collection of programs, data, documents, and other objects provided on some medium such as a CD disk. So a primary task of the software shop is to generate the CD disk with known, correct content. Thus they immediately need to store in their CM database a representation of this content, and this representation must be constructed in some orderly, reliable and reproducible fashion. Further, since some of the contents of this "CD disk configuration" consist of constructed objects, each of these constructed objects should also have their component structure stored in the CM database.

Additionally, the constructed CD disk image must correspond to these configuration structures, which means that these structures ought to be functional and not just documentation. To clarify, the configuration should be constructed to represent the required product, and used to drive the product construction process.

As time passes, other uses for this configuration will arise. If the software shop sells additional copies of the product to other customers, and if they also continue to maintain and enhance the software, multiple versions will inevitably be in concurrent use in the field. Thus they will require (at a minimum for support reasons) that they keep a permanent record of each configuration which is delivered, and the CM database must be able to store and control (i.e. prevent modification) saved configurations.

Versioned Object Management

Now that we have covered the properties and relationships of object versions, we must examine issues such as how and why versions are created, and what happens to these versions over time. Rules or guidelines concerning creation of versions are certainly a very fundamental aspect of a CM methodology, but since the objective is to provide CM mechanisms that are as methodology-independent as possible, we must carefully separate methodological constraints from those CM principles which are widely applicable. Here the case is made that traditional views on versioning are much too limiting, and that a CM system should support a much broader range of possible versioning models.

First we will consider the question of when a new version should be (considered to have been) created. The following points are relevant.

Anytime the appropriate copy(s) of an object are non-modifiable, yet a change must be made to the object, a new version must be created.

On the other hand, a user may wish to start with a copy of a modifiable version (even someone else's) yielding two concurrently modifiable versions. There is nothing inherently wrong with this concept.

There are diverse reasons why such creation may occur, not all of which have to do with creating a successor (in the sense of usage) to the version from which the newer one is derived. Thus, the actual time of creation may be less interesting for CM purposes than the (later) point in time when the new version becomes usable for some purpose.

Many revisions (in the sense of edits or other manipulations) to an object may occur between creation and eventual usage, and it may be the case (depending on methodology) that these intermediate revisions are of no significance to overall CM matters (release generation). In other words, only revisions, which are considered to be “milestones” in some sense, are of importance in creating configurations. We call these potentially numerous and small checkpoint revisions the "micro-versions", while those versions which are considered CM level we call "macro-versions".

It is generally useful for versions created as in [1] to be known to other users, though it is especially useful when the new version is intended to succeed an existing one. The intentions of the version

creator are more important than the simple existence of the version, since development decisions may not be able to be based on the latter.

Versions, which initially are considered to be useful for some purpose (e.g. an integration test), may later be promoted to other uses (e.g. an actual release), or be discarded as unusable. This evolution from creation to initial uses, to eventual uses, and/or obsolescence, can be thought of as the "lifecycle" of the object.

From the above points, we can conclude that in the absence of a more limiting methodology, a generalized versioning model should provide the following.

All versions of objects must be known to the CM system without constraining them to a particular use.

The intended future use of the object may be known — if and only if the user supplied this information explicitly, or the methodology provides it implicitly — but is not necessarily implied simply by the existence or derivation of the version.

The CM system must provide some way for the user to convey when a version is suitable and ready for some intended use.

The CM system may or may not provide for tracking of micro-versions, but obviously must provide for full management of macro-versions.

An important tracking responsibility of a CM system is to know the state of every version within its lifecycle although this lifecycle is defined by users chosen methodology, and may depend on the class of object.

Note that any dependency between versions of different, or the same, objects exists in the absence of specific methodologies. Thus we might say that all objects should be thought of as being versioned independently of, or orthogonal to, each other and any interdependence arises as a result of added operational rules.

Note also that it has not been concluded that any notion of locking is relevant to the general issue of versioning. The idea of locking will arise only when considering strategies for handling concurrency during parallel development, not in the context of simple versioning.

For this reason certain well-established beliefs about CM are rejected. For example, it has long been thought that the "check-in/check-out" (CICO) model is a universal property of CM systems. The "long transaction" model is essentially equivalent to CICO in this regard, though it suggests an attractive correlation between CM models of concurrency with those used in general database theories [5]. This analogy has little practical applicability. It is suggested that more effective methodologies will be far less trivial than these. A CM system must support a more general class of techniques.

The locking aspect of the CICO model is just one example out of myriad examples of potential rules for maintaining some methodologically required conditions on a set of versions of an object. A lock is just a conveyance of a binary property over one (or a set of) version(s). Certainly more information could be conveyed, and with more information perhaps a more efficient overall process can result.

Similarly, the privacy and isolation components of the CICO model are arbitrary (and actually very poor) techniques for achieving insulation between separate concurrent activities. Shortly we will explore the insulation concept more fully to show that more general and effective approaches exist. The CICO model is a two-state lifecycle, which is far too trivial for managing most types of data in a quality-focused, team environment.

Insulation

Another basic problem that CM systems must solve is that of interference between various elements of a project team. Since many versions of objects are constantly being created during development, each group or individual must use some set of these versions in order to get work done. However, in the interest of working with the "right" set, or in particular an "up-to-date" set, techniques are often employed which frequently cause developers to end up with incompatible, nonfunctional, or otherwise unusable configurations. The prevention of these problems termed "insulation."

Here again it can be stated that traditional CM methods overly trivialize the issue. It must be emphasized that insulation cannot be trivialized to isolating developers until such time as they submit work as completed, after which all such work is immediately shared. Modern CM systems should not be limited to the "sharing versus isolation" model anymore. Today's software development organizations are usually collections of a number of different teams under some management structure, and each team has different responsibilities and relationships with other teams. For example, a software project may have one team developing core service libraries with others developing application functions, graphical user interfaces, and generating multiple releases. Each team can be seen to have different relationships with other teams with regard to communicating results and status or work in progress. Within each team there are still further subtleties in relationships and communication paths. All these factors indicate that results of work must flow among diverse elements in a non-trivial way. This multi-dimensional and potentially complex aspect of the development process may need to be served by a variety of forms of insulation.

Another way to look at the issue of insulation is to consider it as a matter of controlling the "visibility" of objects. That is, if a particular piece of (complete or incomplete) work is not visible to some individual (or team), then it could be said that the individual is insulated from the source of that work. Further, there is no abstract difference between insulating two developers from each other and insulating new revisions from end-users or customers. This is also just a matter of limiting delivery of changes to customers until such time as the changes are fully tested and approved.

A further refinement of this concept is that visibility should not imply that work that is not ready for some particular use cannot be known ahead of time to that user. What is really necessary is for the existence and purpose of the work to be known, yet unusable due to methodological restrictions that are supported in the environment. Thus the challenge here is really to accomplish insulation without having to resort to isolation.

Object Lifecycles

Given that the concept of visibility control as a mechanism for achieving insulation between diverse project elements has been adopted, and that it is understood that the level of control over objects may vary along with the changes in visibility, some way to model the evolution of an object's state is necessary. A general technique for doing this is called "lifecycle modeling". The lifecycle of an object is the set of visibility and control states that it moves through between the time of its creation and its eventual obsolescence, together with the rules governing which state-transitions are permitted and what side effects or pre-conditions occur when making each transition.

No single lifecycle model can be expected to apply to all objects in a system, and lifecycles may be specific to (groups of) classes, or may be particular to certain usage of the objects, rather than to the class of the object. Since these lifecycle models are at the core of a CM methodology, a CM system must be very flexible as to the lifecycles it supports.

Consider the following examples of typical lifecycles. The first is a sample lifecycle for a software component (e.g. a source file) which typically proceeds from a developmental state (working), through an integration test stage (test), and then through a quality control process (QA), and finally

to a deliverable (released) state. The initial testing as well as quality assurance procedures can result in the module being passed back to the developer (test → working), or being rejected altogether (sqa → obsolete). Although it is clearly much more complicated than the two-state CICO model, this model is actually fairly simple, once the idea of an object lifecycle is clear.

To illustrate just how different the lifecycles for two object classes may be, consider a lifecycle for a document object, which rather than unit and integration testing, has more of an approval and sign-off based lifecycle.

When lifecycle models such as these are used to represent (part of) a software development methodology, the act of moving an object across the transitions, termed "promotion", is one of the most important operations in the CM system. This follows from the use of these transitions to actually effect changes in control and visibility. In addition to changing the visibility of the object, the promotion operation must ensure that the object meets all required properties defined for objects in the destination state. For example, a source object might not be promoted from a development state into a quality assurance state until it passes some minimum level of coverage testing. Thus, the "promote" operation can be one of the primary actuators of workflow in the CM system. This theory will be shown in practical use in later methodology examples.

Assembling Configurations

In addition to the obvious need to be able to explicitly include an object in a configuration, we believe that modern CM systems require the power and flexibility of rule-based configuration assembly.

In rule-based configuration assembly, the configuration object contains a method that determines the set of objects and specific object versions that should be included in the configuration, and to actually build the structure, if any, is required.

Common rules include "select the latest versions of the files except those that are private and not owned by me", or "select all the files in Release 2 plus the fixes for Patch 9."

This is a key mechanism for supporting insulation and object lifecycles. This mechanism has proven useful in a wide variety of methodologies and in dealing with large amounts of data where explicit specification is not feasible [5, 7].

The configuration member selection rules themselves are a key component of the methodology, in that they provide automated support for the methodology. Specifically, they support the particular means of insulation desired.

An important aspect of rule-based configuration specification is that since this mechanism is used to update the contents of the configuration, the configuration update process must be under the control of the user due to insulation considerations. Otherwise, a particular user's configuration could change without action on his part, a clear violation of the insulation requirement.

Security and Access Control

In addition to inducing information flow in the development environment, promotion can also be used to operate the changes in the control state of an object. This is accomplished by defining some set of access types (privileges), and defining the set of privileges that are allowed for each state in the lifecycle. Consider the following example.

A three-state lifecycle could be defined in which the states are working, integrate, and released. The working state could allow full privileges to its creator, but only read access to others. Others could not, for example, change a module in this state, or even use it in their own configurations. Once the work on the object is initially completed, the object is promoted to integrate, which then allows the object to be used in test configurations, and perhaps in working configurations of other

team members. At this point it may be non-modifiable (or only modifiable by its owner), though possibly retrievable back to the working state by its creator. After sufficient tests have been passed, it might then be promoted to released (but only by a QA manager), which would qualify the object for use in deliverable configurations, and would require that the object be permanently non-modifiable, and not dependent on any other modifiable object.

The access rights required for the above scenario can be summarized as read, write, bind (use in a configuration), and promote. We can also add that perhaps a different set of these applies to the creator or owner of an object than to other team members.

Of course, additional "natural" access rights can easily be thought of, and artificial ones can even be devised to serve purposes of specific methodologies. For example, the right to "sign off" a document might be reserved for certain users of certain roles.

Another behavior, which may usefully be tied to lifecycle, is that of storage control or accessibility control. This behavior deals with the fact that an object currently in use must be readily available, while those, which are older and possibly obsolete, may be archived until the need for them arises. In some methodologies, both short-term archival for space compression and long-term archival for historical purposes are required.

Objects and Methodology

Mechanisms for describing object classes, which include both stored data and stored behavior, and how an object's data and behavior is inherited from its class, and its superclass(es) have been discussed. Additionally, it has been explained that data or behavior can be specialized for each instance, and that objects are manipulated through their methods.

In the object-oriented information repository that has been described, all project entities, both physical (representing configurations, files, libraries, executables) and logical (representing groupings, installation instructions) are represented as objects and stored in the database.

Using this object-oriented repository, a software CM methodology is represented by creating the appropriate objects, with the desired behavior captured in the object's methods, and the project data encapsulated in the object's data attributes. This collection of classes (methods and data) is referred to as the "CM Development Model".

The development model supports the desired methodology. Because the methodology will evolve, so must the development model. The development model is viewed as a key project object, and should be configuration managed along with the other project information. In fact, the products that are built under the CM system's control actually have the development model as a dependency.

Because the classes and their methods are completely dynamic and user extensible, this mechanism can be used to support a wide range of behavior.

There is a good deal more to be said about representing CM process using an object-oriented repository. [1].

Methodologies that can be supported using these mechanisms will now be explored.

CM Methodology

CM methodology is defined as the use of the mechanisms to provide automated support for a particular approach to CM. The CM methodology is sometimes referred to as the "CM process" [6] or "CM model" [5]. The methodology describes how software developers, testers, writers, project managers and release managers use the system, how software moves through its lifecycle, how software is managed, and how changes to the software are controlled.

Some of the methodologies described below are actually compatible with other methodologies in the sense that a team could employ both without conflict. In other cases, the methodologies are mutually exclusive.

The Ladder Methodology

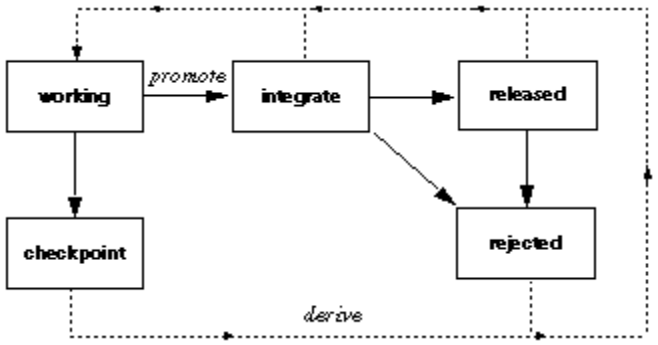
Managing change is fundamental to software configuration management. This may be as simple as checking in and out files, or as involved as the marriage of Problem Tracking and CM. This area, perhaps more so than any other, represents the diversity of methodology in CM systems.

We will describe two methods for creating, managing, and propagating changes through a project team. Both methods make use of many of the mechanisms described earlier, but this is an especially useful example of the value and flexibility of rule-based configuration specification.

The term "Ladder Methodology" refers to the promotion of software versions up through the various "rungs" of the project team. The general notion is that each team member has a personal configuration for each variant of the release they are working on. In addition, there are configurations for each primary state of the source object lifecycle. Each step the software goes through increases its visibility and its reliability (software "climbs the ladder").

The methodology is a function of the desired lifecycle. In this example, the company uses the three-stage lifecycle for their source code objects shown in Figure 1. The "ladder" begins at working, and the software climbs through integrate and released.

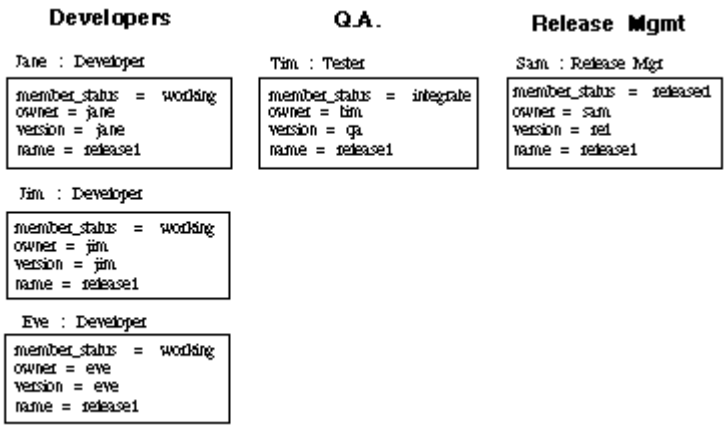
The working state is defined as private, modifiable only by the owner (i.e. the one who checked out the file). The integrate state is used to indicate that the software is ready for others on the team to test with, including QA The released state indicates that QA has approved the software for wider release.



• Figure 1 Lifecycle of Source Objects

There are several security and access control aspects also defined in the methodology which will not be described here, but these supporting details are much as would be expected.

To support this lifecycle, the project team has configurations set up as shown in Figure 2.



• Figure 2 Configurations Defined For Ladder Methodology

The team members and their roles are indicated in the diagram. Each configuration shown is a version of the "release1" configuration. Each team member has a configuration where they do their work. These configurations are guaranteed insulated from changes by others. In this methodology, each person is in complete control over when new versions are incorporated into their configuration.

To understand how this method works, consider what happens when Eve, one of the software developers, decides to make a change to a source file. She derives a new version of a file (let us say main.c). Let us also say the new version is version 3, and the other members of her team are running with version 2. Eve modifies the new version, builds, and tests in her environment. When she believes the new version is ready for testing, (both by QA and her teammates in this example), she promotes the version 3 from the working state (which was the initial state when she created the new version), to the integrate state. It is important to note that in this methodology, simply promoting the file does not affect any of the other versions of the configuration. The only thing that happens is that version 3 in the object pool is now a new state.

One especially important example of where this is critical is for the QA tester, Tim. Tim may have updated his configuration and started the execution of a lengthy regression test suite. It is imperative for the integrity of his test results that the contents of the configuration do not change until Tim says it should. Imagine the confusion and lost time if Tim's tests fail, and before he has analyzed the results, the erroneous version is updated. Or consider the case where during test execution a file changes which causes a long-running test to crash.

Depending on local customizations, the team may or may not have set the environment to notify other team members and/or QA of the new file version being available.

Now consider the system from the QA perspective. Tim is in charge of developing and running automated tests against "release1". He has set up his configuration to update in the middle of the night, build, and automatically run his test suite, storing the results so that he can review them in the morning. That night, the changes that Eve has made and promoted are included into her "qa" configuration, and then tested against them. Note that the member_status attribute is set to integrate in Tim's configuration. This is so that no personal working versions will be included in the tests, even if they are his own. This attribute indicates to the rule-based configuration definition that no version of a state lower (i.e. less visible) than integrate is ever selected for inclusion in the configuration. Note that Sam, the release manager, has a configuration with a member_status of released, because he only wants to include versions that have been approved by QA as ready for release.

If Tim determines through his test suite that the new version of main.c has a problem, he can "reject" the new version by promoting the object to the rejected state. A rejected item is never selected by the configuration member selection rules.

This description has been simplified slightly, and many organizations have made enhancements to this method to support their particular requirements, but this is one real life example of a methodology built entirely on the mechanisms described earlier.

The Change Set Methodology

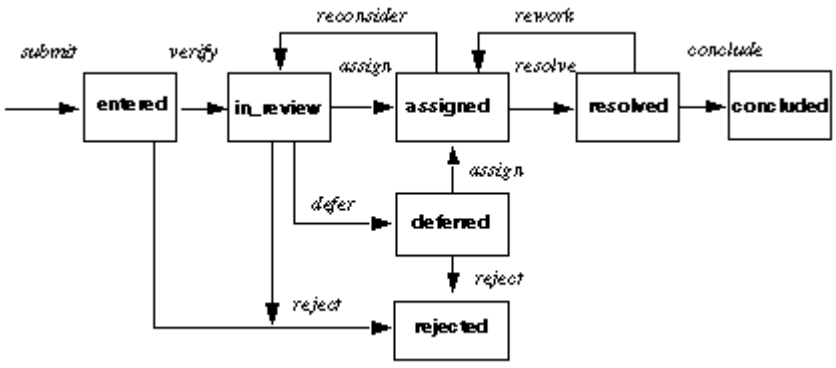
Another very different methodology for managing change is known as the "change set" method. The basic notion here is that rather than checking in and out individual files, you group your changes under a higher level object, which is often referred to as a "change set". The change set represents all of the changes to all of the files to make a given functional change. So "change 3" may actually be composed of changes to ten source files, three header files and a chapter in a user's guide.

Some of the convenient capabilities of change sets are that they can be combined with baselines to form new configurations with good assurance of consistency (which the Ladder methodology does not guarantee). For example, you can create a new "release 2" which is defined as "release 1" plus change sets 3, 12 and 13. The CM methodology and mechanism must determine if change set 12 requires other change sets in order to form the closure of all required changes for delivery of a consistent configuration.

There are actually many variants of the change set methodology. The simple method described above has proven overly simplistic for many teams, primarily due to its lack of support for the concepts of visibility and insulation.

The Change Tracking Methodology

There is a variant on the Change Set Methodology which is a hybrid of the Ladder and Change Set methods, which is referred to as the Change Tracking Methodology.



• Figure 3 Lifecycle of Change Request Object

A full implementation of the change tracking methodology provides a very deep integration of problem tracking functionality with source level configuration management.

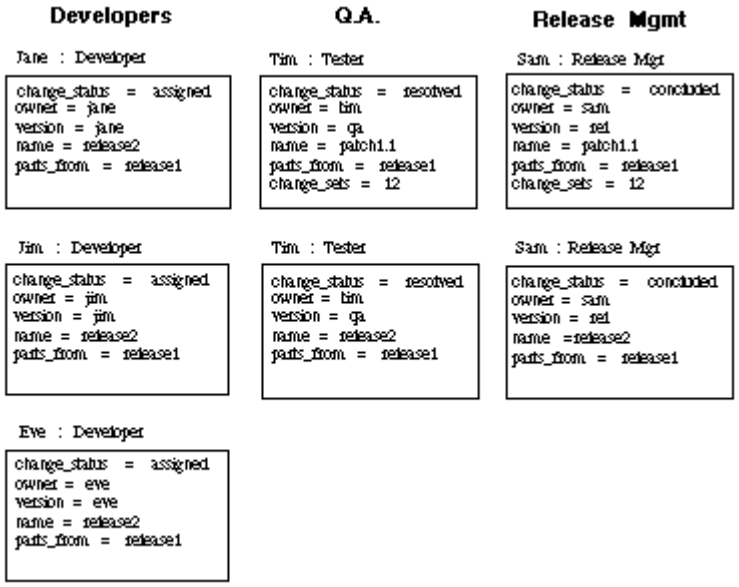
First a lifecycle of the change request object will be defined, as illustrated in Figure 3.

A change set represents zero or more actual changes to the project components (e.g. source code, designs, documents or test cases). A change set is initiated by submitting a change request,

which is then subject to approval. Once approved, the developers (and/or writers if documentation is involved) make changes in the context of a change request, thus accumulating a change set. The CM system, using this methodology, records which objects are added, modified or deleted in the course of implementing the change.

Once the changes are tested, the developer promotes the change set (rather than the individual members) and the software moves through its lifecycle as a result of its association with the change set.

Project members define their configurations through the use of a combination of rule-based configuration definition and reference to baselines. Figure 4 shows configuration specifications for our example project team.



• Figure 4 Configurations Defined for Change Tracking methodology

This example assumes that release1 exists, and that the work in progress is both maintenance to Release 1, and new development on Release 2 (which is also based on Release 1).

The configuration specification for each developer indicates the baseline they are working against, and the change-set(s) that they will see, by default, are all of those which are assigned to them (change_status = assigned). The QA person, Tim, has two configurations: one for a patch release he is working on (patch1.1, which is release1 plus change set 12); and the other configuration including all change sets that have been resolved, indicating that they have been promoted by the developer, yet not verified as corrected.

The Release Manager, Sam, has two configurations also corresponding to the QA configurations, but his rules indicate that only those problems that have been verified fixed (concluded) are to have their components included in his configurations.

Again, this description has been simplified somewhat. Even more so than the ladder methodology, this method is often enhanced to meet local requirements. This basic methodology has proven a good match for several project teams.

One particularly useful variant of the Change Tracking Methodology decomposes change requests into task objects, which provides direct methodological and tool support for multi-person changes.

Models of Parallel Development

Most modern CM systems support parallel development. However, the precise form of parallel development can vary widely. We view parallel development as a spectrum, with purely serialized development on a single line of descent at one extreme, and the "optimistic" scheme made popular by Sun's NSE [3] at the other. Most techniques are somewhere in between. All can be supported using the mechanisms we have described.

It is useful to first define what we mean by parallel development, as this definition varies widely as well.

Parallel development occurs at two levels of granularity. Multiple changes may be made in parallel either by multiple people or by one person working on multiple tasks, to a single source object such as a C source file, or a design document, and simultaneous changes may be made to a configuration. For example, one person may add the new file "foo.c", and another adds the new file "bar.c").

Parallel development involves access control to both the source objects and the configuration, notification (keeping people informed of the parallel activity, when desired), comparison (identifying the differences between parallel versions) and merging (combining parallel versions into a new, common version).

One extreme method for dealing with the problems of parallel development is to enforce serialization. This simply says that parallel versions are prevented. The first person derives a new version and begins work. If another person asks to derive from the same ancestor, then that derive fails, with a reference to whoever owns the new version. This of course makes the person that currently owns the object a bottleneck (which all too often leads to circumvention of the system), which is why many organizations prefer to support at least one of the forms of parallel development.

The most common form of parallel development involves making an explicit branch in the main line of descent, and later (when and if desired) doing an explicit merge of the parallel version(s). Even with this technique the details vary, especially with respect to how the various owners of the parallel versions are kept informed of each other's activities. All owners of parallel versions may be kept informed of all activity with regard to these objects (such as when they are created, promoted or merged); however, most of the common version control and CM systems support this scheme without activity notification.

Another approach to parallel development, referred to as the "optimistic" scheme [4], creates parallel versions implicitly (the creator does not know it is a parallel version at the time of version derivation). Only on promotion is the user informed of a conflict, if one exists; at which point the user must enter a reconciliation process.

The purpose here is not to critique any of these methods, but rather to emphasize the range of styles that can be supported with the general mechanisms described. All forms of parallel development described here are useful in appropriate circumstances.

Instrumentation for the Process

Everything we have discussed under methodology is a part of the overall software development process. However, there are other forms of methodological support that can be applied to nearly any methodology to further specialize the CM system to support the local process requirements. In this section we will discuss some examples of this.

Pre-conditions and Notifications

A very common form of process control is to enforce pre and/or post-conditions on object state transitions or method invocations. For example, certain audit information may be enforced as a pre-condition for promoting an object to a released state. Or, QA may be notified each time a new object is ready for testing, or require that a test suite has run before a configuration object can be released.

State transitions are not the only situations that are appropriate for pre-conditions and notifications. You may also want to be notified or log in an audit trail when source objects have been compiled, or when design diagrams have been edited. The key lesson learned over the past few years from work in control integration was that it is very difficult to predict in advance the sorts of operations that project teams may need to trigger on and what actions they wish to trigger, and that the best policy is to support customizable pre-conditions and notifications on all operations on all object classes [2].

Metrics Collection

It should be clear at this point that definition of the CM process and supporting its continuous improvement is crucial to the software development process. Moreover, the key to continuous process improvement lies in measuring the process to identify where problems lie [6].

Since modern CM systems manage the project data, manage access to that data, and manage change to that data, they are an ideal place to collect information on the process.

There is an abundance of useful raw data stored when using the methodologies described here. For example, the CM system knows how long it takes from the time a change is requested to the time it is approved for release. It knows who has been doing what operations to what data, and when it was done. It knows how many unassigned change requests are in the system, and how long it takes until they are assigned.

Furthermore, since the CM mechanisms described here provide for the dynamic addition of user-defined attributes on any object class, it is simple to add an attribute (such as "essential complexity") to the C source class, and instrument the "build" or the "promote" method to run complexity analysis tools and store the results on the object. Or code stability could be measured just as easily.

The point here is not the value of any particular metric, or set of metrics, but rather to point out both the opportunity for easily collecting project data, and that this data can and should be analyzed in order to identify the process improvements necessary to improve your CM methodology.

Hopefully a useful sampling of the range of CM methods that can be supported with a well-defined set of very general CM mechanisms has been provided. The purpose here is not to endorse any particular methodology. The examples are meant to illustrate the broad range in methodologies.

Conclusion

Process methodology is fundamental to software configuration management. Furthermore, it is the role of CM systems to support the CM process, not just simply provide the mechanisms. The CM process must be able to evolve over time and the system should in fact facilitate this evolution and certainly not hinder it.

The CM mechanisms described here are able to support all four of the commercial CM models described in [5], under the terms "Check-out/Check-in Model", "Composition Model", "Long Transaction Model", and "Change Set Model".

References

[1] Cagan, Martin, and Wright, Alan, "Requirements for a Modern CM System", CaseWare Technical Report CWI-TR-91-02.

[2] Cagan, Martin, "An Architecture for a New Generation of Software Tools", Hewlett-Packard Journal, June 1990.

[3] Courington, William, "The Network Software Environment", Technical Report, Sun Microsystems, Inc. 1989.

[4] Feiler, Peter, "Tool Version Management Technology: A Case Study", SEI Technical Report SEI-90-TR-25, 1990.

[5] Feiler, Peter, "Configuration Management Models in Commercial Environments", SEI Technical Report SEI-90-TR-7, 1990.

[6] Humphreys, Watts, "Managing the Software Process", Addison-Wesley, 1989.

[7] David B. Leblang, Robert P. Chase, Jr., and Gordon D. McLean, Jr. The DOMAIN software engineering environment for large-scale software development efforts. In Proceedings of the 1st International Conference on Computer Workstations, pages 266-280, San Jose, CA, November

[8] Wiebe, Douglas, "Generic Software Configuration Management: Theory and Design", University of Washington, Department of Computer Science, Technical Report 90-07-03.