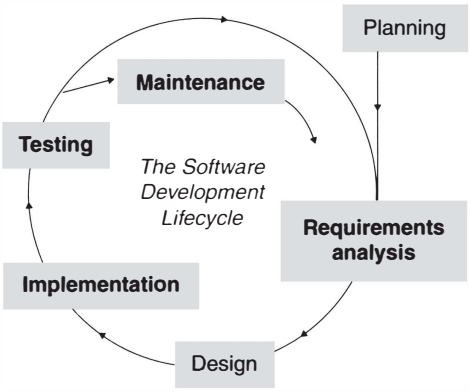
6

Software Configuration Management



- What is the purpose of software configuration management?
- What activities does it consist of?
- How do you plan for configuration management?
- What tools are available to support it?
- How is configuration management handled in large projects, in practice?

Figure 6.1 The context and learning goals for this chapter

Many artifacts are produced in the course of developing a software product, such as specifications (e.g., requirements, design), source and executable code, test plans and test data, user documentation, and supporting software (e.g., compilers, editors). Each undergoes numerous revisions, and keeping track of the various versions needs to be managed in a reliable and consistent manner. *Software Configuration Management* (SCM) is the process of identifying, tracking, and storing all the artifacts on a project. In the context of SCM, each of these artifacts is referred to as a Configuration Item (CI).

SCM contributes to overall software quality in that it supports a reliable way to control a project's artifacts. For example, the SCM process ensures that the proper source files are included when building the software system and that the correct project documentation is retrieved when required.

Many activities contribute to configuration management, including identification of artifacts as configuration items, storage of artifacts in a repository, managing changes to artifacts, tracking and reporting these changes, auditing the SCM process to ensure it's being implemented correctly, and managing software builds and releases. Many of these activities are labor intensive, and SCM systems help automate the process.

6.1 SOFTWARE CONFIGURATION MANAGEMENT GOALS

We first define the overall goals of software configuration management: *baseline safety, overwrite safety, reversion, and disaster recovery*.

Baseline safety is a process of accepting new or changed CIs for the current version of the developing product (the baseline), and safely storing them in a common repository so that they can be retrieved when needed later in a project.

Overwrite safety means that team members can safely work on CIs simultaneously, and changes can be applied so they do not overwrite each other. Overwrite safety is needed when the following kind of sequence occurs.

1. A. Engineer Alan works on a copy of CI X from the common repository.

   B. Brenda simultaneously works on an identical copy of X.

2. Alan makes changes to X and puts the modified X back in the repository.

3. Brenda also makes changes to X and wants to replace the version of X in the common repository with the new version.

Overwrite safety assures that Brenda's changes don't simply replace Alan's, but instead are added correctly to Alan's.

Reversion occurs when a team needs to revert to an earlier version of a CI. This is typically required when mistakes transition a project to a bad state—for example, when it is found that a new version of a CI turns out to cause so many problems that it is preferable to revert to a previous version. Reversion requires knowing which version of each CI makes up a previous version of the project.

Disaster recovery is a stronger form of reversion—it is the process of retaining older versions of an application for future use in case a disaster wipes out a newer version.

These four goals, fundamental for configuration management, are summarized in Figure 6.2.

6.2 SCM ACTIVITIES

There are several SCM activities and best practices that are implemented to successfully meet the goals just described. They are mainly the following:

1. Configuration identification.
2. Baseline control.
3. Change control.
4. Version control.

- **Baseline Safety**

  Ensure that new or changed CIs are safely stored in a repository and can be retrieved when necessary.

- **Overwrite Safety**

  Ensure that engineer's changes to the same CI are applied correctly.

- **Reversion**

  Ensure ability to revert to earlier version.

- **Disaster Recovery**

  Retain backup copy in case of disaster.

Figure 6.2 Major goals of configuration management

5. Configuration auditing.
6. Configuration status reporting.
7. Release management and delivery.

Each of these is described in the following sections.

6.2.1 Configuration Identification

The first step in configuration management is to identify a project's artifacts, or configuration items (CI), that are to be controlled for the project. As described in the introduction, candidate CIs include source and object code, project specifications, user documentation, test plans and data, and supporting software such as compilers, editors, and so on. Any artifact that will undergo modification or need to be retrieved at some time after its creation is a candidate for becoming a CI. Individual files and documents are usually CIs and so are classes. Individual methods may also be CIs, but this not usually the case. A CI may consist of other CIs.

CIs are too large when we can't keep track of individual items that we need to and we are forced to continually lump them with other items. CIs are too small when the size forces us to keep track of items whose history is not relevant enough to record separately.

Once selected, a CI is attached with identifying information that stays with it for its lifetime. A unique identifier, name, date, author, revision history, and status are typical pieces of information associated with a CI.

6.2.2 Baselines

While an artifact such as source code or a document is under development, it undergoes frequent and informal changes. Once it has been formally reviewed and approved, it forms the basis for further development, and subsequent changes come under control of configuration management policies. Such an approved artifact is called a *baseline*. IEEE Std 1042 defines a baseline as a "specification or product that has been formally reviewed and agreed to by responsible management, that thereafter serves as the basis for further development, and can be changed only through formal change control procedures."

Baselines not only refer to individual CIs but also to collections of CIs at key project milestones. They are created by recording the version number of all the CIs at that time and applying a version label to uniquely identify it. Milestones can occur when software is internally released to a testing organization, or when a

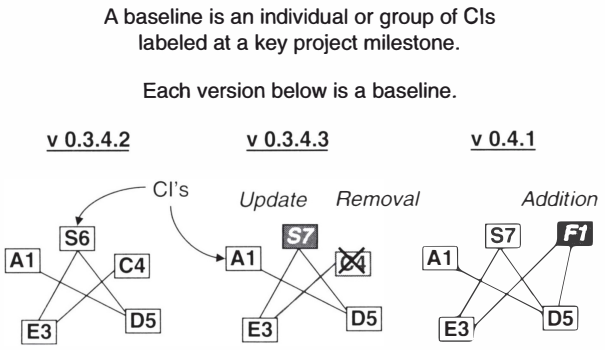


Figure 6.3 Transitioning from one baseline to the next

version of software is packaged for release to a customer. For example, a new software version is created and a set of source files and documentation that constitute software release 1.0 are grouped together, given a unique label such as "version 1.0," and are recorded as belonging to the same baseline. This allows all the files constituting software release 1.0, and their correct versions, to always be correctly identified and grouped together. If files belonging to a baseline are added, deleted, or modified, an updated baseline is created with a new version number. Figure 6.3 illustrates this concept.

Once baselines are created and labeled, they are utilized in a project for three primary reasons [1]:

1. Reproducibility
2. Traceability
3. Reporting

These are explained next.

1. **Reproducibility** means that you can reproduce a particular software version or set of documentation when necessary. This is required during product development as well as during maintenance. For example, software is shipped to customers, and different customers may use different versions. In the meantime, the project team is developing a new software release, and as a result is updating many of the CIs used in previous software releases. If a problem is reported by a customer running an older software version, because it was saved as a baseline the older version can by resurrected by the project team and used to identify and repair the source of the problem.
2. **Traceability** means that relationships between various project artifacts can be established and recognized. For example, test cases can be tied to requirements, and requirements to design.
3. **Reporting** means that all the elements of a baseline can be determined and the contents of various baselines can be compared. This capability can be utilized when trying to identify problems in a new release of software. Instead of performing a lengthy debugging effort, differences in source files between the previous and current software versions can be analyzed. This may be enough to identify the source of the problem. If not, knowing exactly what changes occurred can point to potential root causes, speeding up the debugging effort and leading to faster and easier problem resolution. Baselines are also useful to ensure that the correct files are contained in the executable file of a software version. This is used during configuration audits, and is covered in Section 6.2.5.

6.2.3 Change Control

Configuration items undergo change throughout the course of development and maintenance, as a result of error correction and enhancement. Defects discovered by testing organizations and customers necessitate repair. Releases of software include enhancements to existing functionality or the addition of new functionality. *Change control*, also known as configuration control, includes activities to request, evaluate, approve or disapprove, and implement these changes to baselined CIs [2].

The formality of these activities varies greatly from project to project. For example, requesting a change can range from the most informal—no direct oversight for changing a CI—to a formal process—filling out a form or request indicating the CI and reason for change, and having the request approved by an independent group of people before it can be released. Regardless of the formality of the process, change control involves identification, documentation, analysis, evaluation, approval, verification, implementation, and release as described next [2].

Identification and documentation

The CI in need of change is identified, and documentation is produced that includes information such as the following:

- Name of requester
- Description and extent of the change
- Reason for the change (e.g., defects fixed)
- Urgency
- Amount of time required to complete change
- Impact on other CIs or impact on the system

Analysis and evaluation

Once a CI is baselined, proposed changes are analyzed and evaluated for correctness, as well as the potential impact on the rest of the system. The level of analysis depends of the stage of a project. During earlier stages of development, it is customary for a small group of peer developers and/or the project manager to review changes. The closer a project gets to a release milestone, the more closely proposed changes are scrutinized, as there is less time to recover if it is incorrect or causes unintended side effects. At this latter stage, the evaluation is often conducted by a change control board (CCB), which consists of experts who are qualified to make these types of decisions. The CCB is typically comprised of a cross-functional group that can assess the impact of the proposed change. Groups represented include project management, marketing, QA, and development. Issues to consider during the evaluation are as follows:

- Reason for the change (e.g., bug fix, performance improvement, cosmetic)
- Number of lines of code changed
- Complexity
- Other source files affected
- Amount of independent testing required to validate the change

Changes are either accepted into the current release, rejected outright, or deferred to a subsequent release.

Approval or disapproval

Once a change request is evaluated, a decision is made to either approve or disapprove the request. Changes that are technically sound may still be disapproved or deferred. For example, if software is very close to being released to a customer, and a proposed change to a source file requires many complex modifications, a decision to defer repair may be in order so as to not destabilize the software base. The change may be scheduled for a future release.

Verification, implementation, and release

Once a change is approved and implemented, it must be verified for correctness and released.

6.2.4 Version Control

Version control supports the management and storage of CIs as they are created and modified throughout the software development life cycle. It supports the ability to reproduce the precise state of a CI at any point in time. A configuration management system (also known as a version control system) automates much of this process and provides a repository for storing versioned CIs. It also allows team members to work on artifacts concurrently, flagging potential conflicts and applying updates correctly.

A good version control system supports the following capabilities:

1. Repository
2. Checkout/Checkin
3. Branching and merging
4. Builds
5. Version labeling

These are explained in the following sections.

6.2.4.1 Repository

At the heart of a version control system is the repository, which is a centralized database that stores all the artifacts of a project and keeps track of their various versions. Repositories must support the ability to locate any version of any artifact quickly and reliably.

6.2.4.2 Checkout and Checkin

Whenever a user needs to work on an artifact, they request access (also known as *checkout*) from the repository, perform their work, and store the new version (also known as *checkin*) back in the repository. The repository is responsible for automatically assigning a new version number to the artifact.

Files can usually be checked out either *locked* or *unlocked*. When checking out locked, the file is held exclusively by the requester and only that person can make modifications to the file and check in those changes. Others may check out the file unlocked, which means it is read-only and they only receive a copy. Concurrent write privileges can be achieved by branching, which is explained in the next section.

When checking in a file, version control systems record the user making the change and ask the person to include an explanation of why the file was changed and the nature of the changes. This makes it easier to see how a file has evolved over time, who has made the changes, and why they were made.

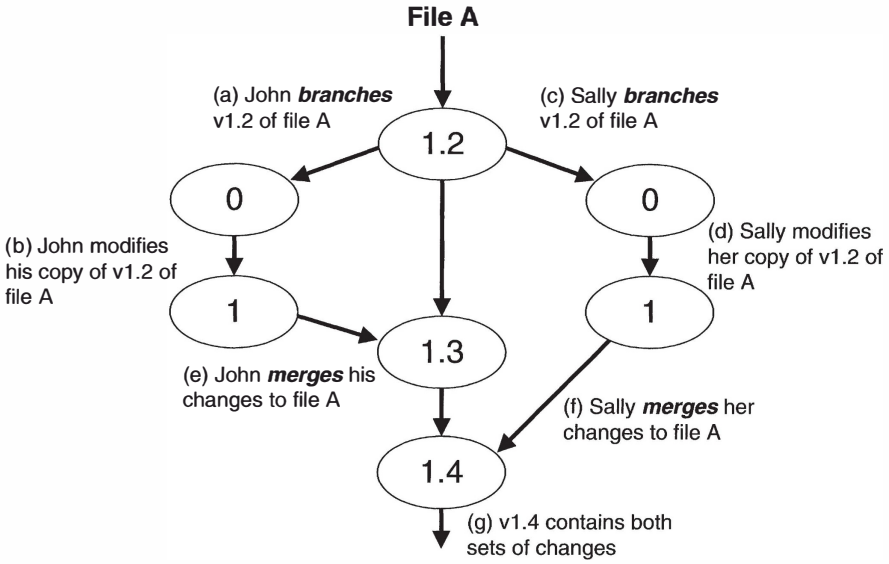


Figure 6.4 Branching and merging changes

6.2.4.3 Branching and Merging

Projects are most often developed by teams of people. Version control systems provide mechanisms to allow team members to work on the same set of files concurrently (*branching*) and correctly apply changes to common files so that none are lost or overwritten (*merging*). This is also known as overwrite safety.

Figure 6.4 depicts an example, in which John creates a *branch* by checking out version 1.2 of file *A*. A branch is a private work area in a version control system that allows you to make changes to baselined files without conflicting with other team members. John maintains a private copy of the file on his branch and makes his changes. Concurrently, Sally also needs to work on file *A*, so she creates her own branch and checks out version 1.2. Her copy starts out the same as John's since he has not yet checked in his updates. Once John finishes his changes, he checks in his files to the repository, and the file is given a new version number 1.3. Later, Sally finishes her modifications to *A* and wants to check in the file. If the version control system allowed her to just replace the current copy of file *A* (version 1.3, which now includes John's changes) with her copy, John's changes would be lost. Instead, the version control system supports *merging*, which intelligently applies Sally's changes to version 1.3 and creates a new version 1.4 with both of their changes applied correctly. If the set of changes are made to various parts of the same file, the system can usually perform the merge operation automatically. If the changes conflict with each other, as when the same lines of code are changed, the system will ask the user to merge the changes manually. In this case the system will show the user which lines overlap so that person can apply the changes correctly.

6.2.4.4 Builds

Version control systems provide support so as to reliably and reproducibly compile and build the latest version of software files into an executable, usable file. A user can specify which branch of code to build from, allowing concurrent development. In addition to the executable file, builds produce logs containing the file versions comprising the build.

6.2.4.5 Version Labeling

Versions are created by applying a label to all files comprising a software build. The label is usually a version number, such as "version 1.0". This allows software versions to easily be referenced and reconstructed if necessary, and supports the construction of baselines.

6.2.5 Configuration Audits

As defined by IEEE 1028-2008 [3], a software audit is "an independent examination of a software product, software process, or set of software processes performed by a third party to assess compliance with specifications, standards, contractual agreements, or other criteria." In the context of configuration management, the goals of a configuration audit are to accomplish the following:

- Verify that proper procedures are being followed, such as formal technical reviews.
- Verify that SCM policies, such as those defined by change control, are followed.
- Determine whether a software baseline is comprised of the correct configuration item. For example, are there extra items included? Are there items missing? Are the versions of individual items correct?

Configuration audits are typically conducted by the quality assurance group.

6.2.6 Configuration Status Reporting

Configuration status reporting supports the development process by providing the necessary information concerning the software configuration. Other parts of the configuration process, such as change and version control, provide the raw data. Configuration status reporting includes the extraction, arrangement, and formation of reports according to the requests of users [4]. Configuration reports include such information as the following:

- Name and version of CIs
- Approval history of changed CIs
- Software release contents and comparison between releases
- Number of changes per CI
- Average time taken to change a CI

6.2.7 Release Management and Delivery

Release management and delivery define how software products and documentation are formally controlled. As defined in IEEE 12207-1998 [5], "master copies of code and documentation shall be maintained for the life of the software product. The code and documentation that contain safety or security critical functions shall be handled, stored, packaged and delivered in accordance with the policies of the organizations involved." In other words, policies must be implemented to ensure that once software and documentation is released it must be archived safely and reliably, and can always be retrieved for future use.

Figure 6.5 IEEE 828-2005 Software Configuration Management Plan table of contents

Source: IEEE Std 828–2005.

6.3 CONFIGURATION MANAGEMENT PLANS

To specify how the software configuration is to be managed on a project, it is not sufficient merely to point to the configuration management tool that will be used. There is more to the process, such as what activities are to be done, how they are to be implemented, who is responsible for implementing them, when they will be completed, and what resources are required—both human and machine [2]. The IEEE has developed a standard for software configuration management plans, IEEE 828-2005. This can be very useful in making sure that all bases have been covered in the process of CM. Figure 6.5 shows the relevant contents of this standard, which are included in Chapter 3 of the plan.

The topics to be specified in Section 3.3 of the SCMP are largely covered in Section 6.2 of this chapter. Section 3.3.3 of the SCMP documents the means by which the status of SCM is to be communicated (e.g., in writing, once a week). Section 3.3.6 applies if a CM tool is used or if configuration management is handled by a subcontractor. The IEEE standard describes the purpose of each section of the above outline in detail. IEEE 828-2005 is used in the Encounter case study later in this chapter.

6.4 CONFIGURATION MANAGEMENT SYSTEMS

For all but the most trivial application, a configuration management system (also known as a version control system) is indispensable for managing all the artifacts on a project. Microsoft's SourceSafe™ is in common use. CVS is a common environment, as well as Subversion and others.

6.4.1 Concurrent Version System (CVS)

The Concurrent Version System (CVS) is a commonly used, free, open source configuration management system. CVS implements a client-server architecture, with users running client CVS software on their

- **Up-to-date**

  Is identical to the latest revision in the repository.

- **Locally Modified**

  File has been edited but has not replaced latest revision.

- **Needs a Patch**

  Someone else has committed a newer revision to the repository.

- **Needs Merge**

  You have modified the file but someone else has committed a newer revision to the repository.

- **Unknown**

  Is a temporary file or never added.

Figure 6.6 File status possibilities in CVS

machines, and a CVS server storing a repository of project files. Users check out projects (using the *checkout* command); make changes; check in (using the *commit* command); and merge with the changes of others who checked out at the same time (using the *update* command). CVS automatically increments the version number of files or directories (e.g., from 2.3.6 to 2.3.7). The status possibilities for a file are shown in Figure 6.6.

6.5 CASE STUDY: ENCOUNTER VIDEO GAME

What follows is a configuration management plan for Encounter. The Software Configuration Management Plan (SCMP) for Encounter is based on IEEE Std 828-2005 Standard for Software Configuration Management Plans. The table of contents of the relevant sections is outlined in Figure 6.8, which is Chapter 3 of the IEEE specification.

ENCOUNTER SOFTWARE CONFIGURATION MANAGEMENT PLAN

APPROVALS

Title	Signature	Date
Engineering Manager	*P. Jones*	*6/15/04*
QA Manager	*L Wilenz*	6/11/04
Project Manager	*A. Pruitt*	*6/7/04*
Author	*E. Braude*	*6/1/04*

Note to the Student:
It is a good idea to have each team member sign off on the physical document. This process focuses their attention on the fact that they are accountable for its contents, and they will be more likely to ensure that it is the document they intend.

Revision History

This assumes the existence of a method whereby revision numbers of documents are assigned.

Version 1

1.0.0 E. Braude: Created first draft 5/1/98

1.1.0 R. Bostwick: Reviewed 1/10/99

1.1.1 E. Braude: Expanded 3.2 1/18/99

1.2.0 E. Braude: Reviewed for release5/18/99

1.2.1 E. Braude: Final editing4/30/99

Version 2

2.0.0 E. Braude: significant edits of section xx 5/2/99

2.0.1 E. Braude: edits 5/13/04

The table of contents of this SCMP follows that of IEEE standard 828-2005.

3.1. Introduction

This Software Configuration Management Plan (SCMP) describes how the artifacts for the Encounter video game project are to be managed.

3.1.1 Definitions

Approved CIs: CIs signed off by project management

Artifact: A final or interim product of the project (e.g., a document, source code, object code, test result)

Master file: A particular designated file for this project, defined in Section 3.3.1.2

3.1.2 Acronyms

CI: configuration item—an item tracked by the configuration system

CM: configuration management—the process of maintaining the relevant versions of the project

SCMP: the Software Configuration Management Plan (this document)

3.2. SCM Management

3.2.1 Organization

> State how this is to be managed. Supply role (s), but no names or responsibilities. Names are supplied in a later section.

A specific engineer, provided by the QA organization, will be designated as the "configuration leader" for the duration of the project.

3.2.2 SCM Responsibilities

> State the tasks that each role must carry out. If this is not stated, essential activities will not be done, and some activities will be done by more than one team member. Include backup responsibilities in case the main individual is incapacitated.

3.2.2.1 Configuration Leader

> "Responsible" does not necessarily imply that the individual does all of the work—merely that he or she organizes the work and sees to it that the work is done.

The configuration leader shall be responsible for organizing and managing configuration management (CM). Whenever possible, the configuration leader shall discuss CM plans with the development team prior to implementation. He or she will maintain this document (the SCMP). The configuration leader is responsible for the installation and maintenance of the configuration management tool(s) specified in Section 3.2.3. Archiving is to be performed in accordance with department policies 12345.

The SCM leader shall be responsible for acquiring, maintaining, and backing up the configuration tools used. He or she shall also develop a plan of action if tools become unsupported (e.g., by discontinuance of the vendor). Additional responsibilities of the configuration leader are stated in Sections 3.1-3.6.

3.2.2.2 Project Leader The project leader and his or her manager will take over the configuration leader's function only under exceptional circumstances. They are responsible for knowing all the

relevant means of access to documents throughout the life of the project. The project leader shall ensure that archiving is performed in accordance with the policies in Section 3.2.3 below.

Additional responsibilities of the managers are stated in Sections 3.3.3 and 3.3.4.

3.2.2.3 Engineers It is the responsibility of each engineer to abide by the CM rules that the configuration leader publishes. Engineers are also referred to "Standard Engineering Responsibilities," document 56789.

Additional responsibilities of the engineers are stated in Section 3.3 below.

3.2.3 Applicable Policies, Directives, and Procedures

> Activities such as CM are generally conducted in accordance with group or corporate guidelines. Student teams should identify and list their policies in this section. Policy 3 should be included.

1. Configuration management for this project shall be carried out in accordance with the corporate guidelines for configuration management, corporate document 7890 version 6 (8/15/98).
2. In accordance with division software improvement policies, midstream and post-project review sessions are required, where improvements to these guidelines are to be documented for the benefit of the organization. These sessions are required to help prepare the division for level 5 CMM certification. The self-assessment results are to be sent to the manager of Software Self-Assessment within three weeks of the assessment session. All "room for improvement" sections are to contain substantive material, with specific examples.
3. All current and previously released versions of CIs will be retained.
4. The master file (defined in Section 3.3.1.2) can be accessed only by the configuration leader and, in his or her absence, the department manager.
5. CM passwords should be changed in accordance with corporate security practices, with the following addition: No password shall be changed until the project leader, his manager, and the manager of QA have all been notified and have acknowledged the notification.
6. The project leader and department manager are to have complete access to all documents under configuration at all times. Access verification form www.ultracorp.division3.accessVerification is to be submitted every two weeks by the project leader to his or her manager.
7. The Encounter project will use SuperCMTool release 3.4, a configuration management product by SuperCMTool.

> These are fictitious names.

8. Archiving is to be performed in accordance with department policies 123456.

3.3. SCM Activities

3.3.1 Configuration Identification

> This section states how configuration items (CIs) come into being and how they get their names. Without such procedures being stated and followed, chaos results.

3.3.1.1 Identifying Configuration Items The project leader shall be responsible for identifying all CIs. Engineers wishing to propose CIs shall secure his or her agreement, via e-mail or otherwise. If the project leader is unavailable for one business day following the engineer's e-mailed proposal for inclusion, the configuration leader shall have the authority to accept the proposed item.

3.3.1.2 Naming Configuration Items The configuration leader shall have the responsibility for

labeling all CIs. The file conventions shall be as follows:

Root directory: Encounter

Subdirectory: SRS or SDD or . . .

File N-N-N.xxx corresponding to version N.N.N

For example, version 2.4.8 of the SRS will be on file Encounter/SRS/2_4_8.txt.

The text file Master in the root directory states the versions of the CIs that comprise the current and prior states of the project. For example, Master could include information such as:

The current version of Encounter is 3.7.1. It comprises version 2.4.8 of the SRS, version 1.4 of the SDD.

The previous version of Encounter was 3.6.11. It comprised version 2.4.8 of the SRS, version 1.3 of the SDD.

This information shall be maintained in a table of the following form.

Encounter SRS version SDD version. . . .

Release

3.3.1.3 Acquiring Configuration Items

In specifying this section, imagine the most stressful part of the project, which is the implementation phase, involving several people in parallel. The process has to be very orderly, but it also has to allow engineers reasonable access to the parts of the project so that they can start work quickly.

Engineers requiring CIs for modification shall check them out using SuperCMTool's checkout procedure. Note that SuperCMTool prompts the user with a form requesting an estimate of how long the checkout is anticipated, and stores this information for all requesters of the CI. Anyone requiring a CI that is currently checked out should negotiate with the current owner of the CI to transfer control through SuperCMTool. A read-only version of the CI is available to all engineers. Under no circumstances may an engineer transfer a CI directly to anyone.

3.3.2 Configuration Control

This section spells out the process whereby configuration items are changed. This process should be flexible enough to allow quick changes, but controlled enough to keep changes very orderly so that they improve the application, not damage it.

3.3.2.1 Requesting Changes As specified in the Software Project Management Plan (see Part III), the team will designate an "inspector" engineer who is allocated to each team member. Before requesting a change, engineers must obtain an inspection of the proposed change from an inspection team or, if this is not possible, from their inspector engineer. To request the incorporation of a changed CI into the baseline, form www.ultracorp.division3.Encounter .submitCI must be submitted to the configuration leader and the project leader, along with the changed CI and the original CI.

3.3.2.2 Evaluating Changes

For larger projects, a group of people, often called the Change Control Board, evaluates and approves changes. Student teams must make this process reasonably simple.

The project leader or designee will evaluate all proposed changes. The project leader must also specify the required quality standards for incorporation.

3.3.2.3 Approving or Disapproving Changes

The project leader must approve proposed changes. If the project leader is unavailable for three business days following the submission of a proposed change, the configuration leader shall have the authority to approve changes.

3.3.2.4 Implementing Changes

> To avoid chaos, it is natural to give to the CM leader the responsibility for incorporating changes; this can create a bottleneck at implementation time, however. Before this "crunch" occurs, the CM leader should find ways to remove this bottleneck by distributing as much work as feasible to the engineers making the changes.

Once a CI is approved for incorporation into the baseline, the configuration leader shall be responsible for coordinating the testing and integration of the changed CI. This should be performed in accordance with the regression test documentation described in the Software Test Documentation. In particular, the configuration leader shall coordinate the building of a version for testing. Version releases must be cleared with the project leader, or with the manager if the project leader is absent.

3.3.3 Configuration Status Accounting

The configuration leader shall update the configuration summary at least once a week on the project configuration Web site www.ultracorp.division3/Encounter/Configuration. SuperCMTool's status report will be a sufficient format for the summary.

3.3.4 Configuration Audits and Reviews

> In industry, random audits are often employed. They are not commonly conducted by student teams due to a lack of resources, although some teams have carried them out successfully. Periodic reviews, as part of the regular team meetings, do not take much time, and they are recommended.

The project manager shall schedule a review by the CM leader of the configuration at least once every two weeks, preferably as an agenda item for a regularly scheduled weekly project meeting. The CM leader shall review CM status, and report on the proposed detailed procedures to be followed at code and integration time.

Configuration efforts will be subject to random audits throughout the project's life cycle by the IV&V team.

3.3.5 Interface Control

The CM system interfaces with the project Web site. This interface shall be managed by the configuration leader.

3.3.6 Subcontractor/Vendor Control

The configuration leader shall track upgrades and bug reports of SuperCMTool. He or she should be prepared with a backup plan in case the maintenance of SuperCMTool is discontinued. This plan is to be sent to the project leader within a month of the project's inception.

3.4 SCM Schedules

> The SCM schedule can be provided here, or combined with the project schedule in the SPMP. In the latter case, this section would not repeat the schedule, but would merely point to the SPMP.

The schedule for configuration management reporting, archiving, and upgrading is shown in Figure 6.7.

3.5 SCM Resources

The configuration leader will require an estimated average of six hours a week to maintain the system configuration for the first half of the project, and twelve hours a week for the second half. We have chosen not to call out separately the time spent by the other team members on configuration management.

3.6 SCM Plan Maintenance

> All project documents undergo change throughout the duration of the project. The SCMP is especially sensitive to change, however, because it controls change itself.

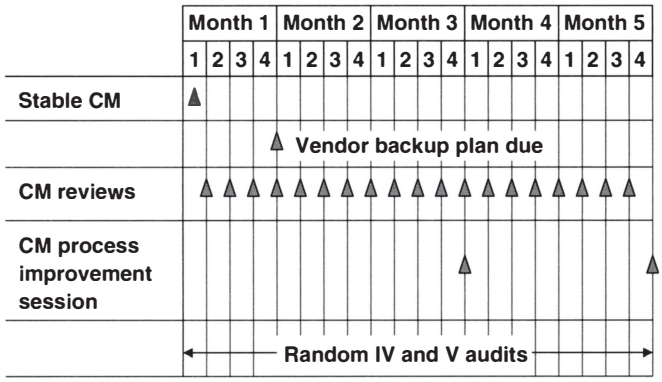


Figure 6.7 Configuration management schedule

Due to the importance of a stable SCM plan, all changes to this document must be approved by the entire CM team.

In view of the software development organization's goal to attain CMM level 5, the configuration leader will do the following for the CM process improvement sessions:

- Review the effectiveness of this plan
- Quantify losses due to defects in this plan
- Review the effectiveness of Super CMTool
- Investigate the literature for new CM methods; quantify the costs and benefits of improvements
- Investigate new CM tools
- Suggest specific improvements to this CM process
- List the benefits of improvements
- Provide cost estimates on effecting the improvements
- Prioritize the cost/benefit ratios of all the suggested changes

6.6 CASE STUDY: ECLIPSE

CONFIGURATION MANAGEMENT IN ECLIPSE[1]

Open source projects such as Eclipse are developed by geographically dispersed engineers who have different source code access requirements. Most are developing their own plug-ins, which are extensions to existing Eclipse code and functionality, and only require access to Eclipse source code for debugging their work. Others may want to change existing Eclipse source code if they are fixing a bug, or add code if they are developing a feature. These developers require write-access to Eclipse code.

Source code is managed in Eclipse using CVS (Concurrent Versioning System), which is the de facto version control system for open source projects. Eclipse integrates a built-in CVS GUI, making version control very easy to use. CVS implements a client-server architecture, with the Eclipse repository housed on a central server at dev.eclipse.org that stores all the Eclipse source code.

In general there are two classes of Eclipse users as follows:

1. Those that want to read and/or modify Eclipse code but don't have write access to the CVS repository.
2. Those that do have write permission for the Eclipse source code and can modify and update the CVS repository. These people are call *committers*.

The following is quoted from [7] and describes these two classes of users.

[1] This section is based on information from [6].

Anonymous CVS

For people who actually want to change Eclipse code but who do not have the required commit rights in that area, all elements of the Eclipse project are available via anonymous access to the development CVS repository. Using anonymous access you can checkout code, modify it locally, but cannot write it back to the repository. This is handy if you would like to fix a bug or add a feature. Get the code via anonymous access, do your work, and then pass the work on to a committer for inclusion in the repository. . . .

All committers must use SSH (Secure SHell) to access the CVS repository if they wish to use their user id and password (i.e., if they want to write to the repository).

Full CVS

Developers with commit rights have individual user ids and passwords in the Eclipse project development repository. As a committer you can use SSH (Secure SHell) to connect to the CVS repository as follows. . . . . Once your information is authenticated, you can browse the repository and add projects to your workspace. If you do some changes that you'd like to contribute, after testing and ensuring that you have followed the contribution guidelines, you are free to release your changes to the repository. Of course, you can only release changes to projects for which you have commit rights.

Note that you can use the SSH protocol and your Eclipse user id to access projects for which you are not a committer, but you will not be able to release changes.

These points are summarized in Figure 6.8.

VERSION NUMBERING

Eclipse plug-ins use a version-numbering scheme that captures the nature of the changes implemented by the plug-in. Version numbers are composed of four parts as follows:

Major, minor, service, and qualifer. *Major, minor,* and *service* are integers, and *qualifier* is a string.

- Major—this number is incremented each time there is a breakage in the API
- Minor—this number is incremented for "externally visible" changes
- Service—this indicates a bug fix or other change not visible through the API
- Qualifier—this indicates a particular build

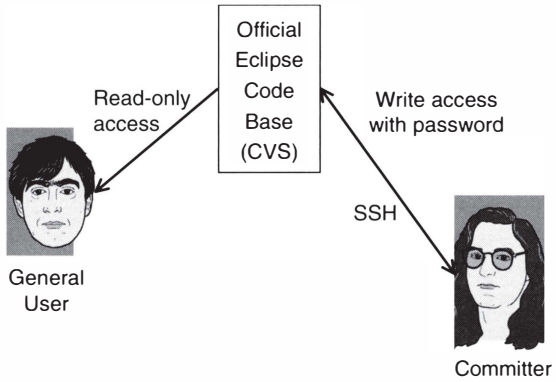


Figure 6.8 Eclipse configuration management

First development stream

1.0.0

Second development stream

1.0.100 (indicates a bug fix)
1.1.0 (a new API has been introduced)
The plug-in ships as 1.1.0

Third development stream

1.1.100 (indicates a bug fix)
2.0.0 (indicates a breaking change)

The plug-in ships as 2.0.0

Maintenance stream after 1.1.0

1.1.1

The plug-in ships as 1.1.1

Figure 6.9 Eclipse plug-in version numbering 1 of 2
Source: Eclipse Wiki, http://wiki.eclipse.org/Version_Numbering.

The example shown in Figure 6.9 and 6.10, taken from http://wiki.eclipse.org, shows how the version number changes as a result of plug-in development. The description shows that bug fixes, new application programming interfaces, and shipping to the community are reflected in the numbering in particular ways.

6.7 STUDENT TEAM GUIDANCE: CONFIGURATION MANAGEMENT

Student teams should create a Software Configuration Management Plan (SCMP) for their project using IEEE Std 828-2005 as a template and the Encounter SCMP in Section 6.5 guidance. The rest of this section describes how to organize for this task.

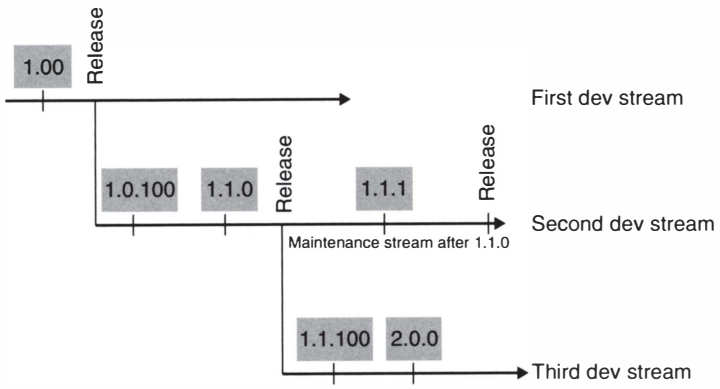


Figure 6.10 Eclipse plug-in version numbering 2 of 2
Source: Eclipse Wiki, http://wiki.eclipse.org/Version_Numbering.

1. Roughly sketch out your SCMP
   - Determine procedures for making changes.
   - Omit tool references unless already identified one.
   - See the case study for an example.
2. Specify what you need from a CM tool
   - For class use, maybe only locking and backup.
3. Evaluate affordable tools against needs and budget
   - Commercial tools are in wide use.
   - For class use, try free document storage Web sites; simple method of checking out, e.g., renaming, can be too simple.
4. Finalize your SCMP

Figure 6.11 Planning configuration management

Figure 6.11 shows how teams can go about deciding their configuration management methods.

When there is insufficient time to learn a CM environment, teams have succeeded reasonably well with simple Web sites, such as www.yahoogroups.com, that allow document storage. A simple checkout system is needed, one of which is to change the document type. For example, when the SQAP is checked out to Joe, the file is changed from *sqap.txt* to *sqap.joe*. Although configuration management applies to both documents and source code, the file-naming convention usually has to be planned separately. For example, we cannot change *myClass.java* to *myClass.joe* without disrupting compilation. Some groups maintain two directories. One contains the current baseline, which cannot be changed without a formal process. The other directory contains versions that are currently being worked on.

Trial CM tools or free ones such as CVS and Subversion are available. Google supports free document hosting with Google Docs and has support for version control. Be sure that your process does not rely on excessive manual intervention, and that it does not result in a bottleneck where one person is overloaded. If you are considering using a tool, be sure that the length of the learning curve justifies its use. There are many other software engineering aspects to learn besides using a particular CM tool. Whatever system you select, try it out first on an imagined implementation. Make sure that the process is smooth. You do not want to worry about your CM process during the implementation phase, when time is limited. In the work world, however, professional CM tools are a necessity.

6.8 SUMMARY

Software configuration management (SCM) is a process for managing all the artifacts produced on a software project, including source code, specifications, and supporting software. Planning for SCM starts early in a project, starting with the Software Configuration Management Plan. It specifies the SCM activities to be implemented throughout development and software maintenance, such as identification, change control, version control, audits, status reporting, and release management.

Early in a project, artifacts are identified as configuration items (CI) to be stored in a repository and managed. After a CI is reviewed and approved it becomes part of a baseline and is officially managed by SCM policies. Artifacts go through inevitable change, being newly created or modified due to either error correction or enhancement. SCM defines activities to request, evaluate, approve or disapprove, and implement changes to baselined CIs.

A key requirement of SCM is the ability to reproduce the precise state of all CIs at any point in time. A version control system (also known as a configuration management system) automates much of this process and provides a repository for storing versioned CIs. It also allows team members to work on artifacts concurrently, flagging potential conflicts and applying updates correctly. Version control systems include functionality for checking in and checking out files, branching and merging, building the software, and creating software versions.

Configuration audits, which are typically conducted by the quality assurance group, are used to ensure that agreed upon SCM procedures and policies are being followed, and that software being produced is comprised of the correct components.

Configuration status reports support the audit process by providing a detailed history for each CI including when it was created and modified, how it was modified, and by whom.

6.9 EXERCISES

1. In your own words, define the term *configuration item* and describe its purpose.
2. Why is it necessary for compilers to be identified as configuration items? Describe a scenario that illustrates when this is necessary.
3. Describe the four goals of configuration management in your own words, and explain why each is important.
4. If you are developing a software application using an incremental process, at what points would you minimally create baselines?
5. Explain the difference between change control and version control.
6. Agile processes promote continuous integration, which results in branching and merging at very frequent intervals (as often as every few hours). Describe one advantage and disadvantage of branching and merging so frequently. Describe one advantage and disadvantage of branching and merging less frequently.
7. As mentioned in the Team Guidance section (Section 6.6), for small teams it may not be necessary to utilize an automated configuration management system. Describe the process you would follow to perform branching and merging without such a system.
8. Research a configuration management system such as Subversion or CVS. Describe how it implements the seven SCM activities listed in Section 6.2.

TEAM EXERCISE

For the team exercise, consider as a group how you will perform it, check the hints below, and then carry out the assignment.

CM Plan

Produce a software configuration management plan for your team project using IEEE standard 828-1990. The case study should guide your team, but your document will be more specific, reflecting the

particular resources available to you. Do not include material unless it contributes to the document's goals. Avoid bottlenecks and unnecessary procedures. Include procedures for what to do if people cannot be reached and deadlines loom.

Before you begin, estimate the number of defects per page the team thinks it will discover during the final review. Keep track of and report the time spent on this effort by individual members and by total team effort. State the actual defect density (average number of defects per page). Assess your team's effectiveness in each stage on a scale of 0 to 10. Summarize the results using the numerical results, and state how the team's process could have been improved.

Criteria:

1. Practicality: How well does the plan ensure that documents and their versions will be secure, coordinated, and available? (A = plan very likely to ensure coordination and availability)
2. Specifics: How specific is the plan in terms of suitably naming places and participants? (A = no doubt as to what engineers must do)
3. Process assessment and improvement: To what degree did the team understand the strengths and weaknesses of its process, and how specific are its plans for improvement? (A = full, quantitative understanding, with plans for improvement very specific and realistic)

Hints for Team Exercise

Do not fill in sections that are as yet unknown; add only parts that you are confident of being able to implement within the semester. Make your plan realistic. For example, don't state that "all other team members will review each contributor's work" unless you are reasonably sure that this can be done within the probable constraints of your project. It is too early to make any assumptions about the architecture and design of the application.

BIBLIOGRAPHY

1. Bellagio, David, and T. Milligan, "Software Configuration Management Strategies and IBM Rational Clearcase: A Practical Introduction," IBM Press/Pearson plc, 2005, p. 7.
2. "IEEE Standard for Software Configuration Management Plans," *IEEE Std 828-2005*, August 2005.
3. "IEEE Standard for Software Reviews and Audits," *IEEE Std 1028-2008*, August 2008.
4. Hass, Anne M. J., *"Configuration Management Principles and Practice,"* Addison-Wesley, 2003, p. 25.
5. "Systems and software engineering—Software life cycle processes," *IEEE Std 12207-2008* Second edition, January 2008, p. 69.
6. Eclipse, http://www.eclipse.org [accessed August 13, 2009].
7. Eclipse, http://www.eclipse.org/eclipse/ [accessed August 13, 2009].