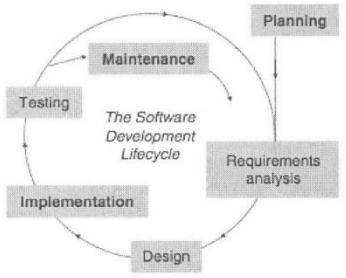
Source: Braude, M.J. & Bernstein, M.E. (2016). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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).

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

- **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

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

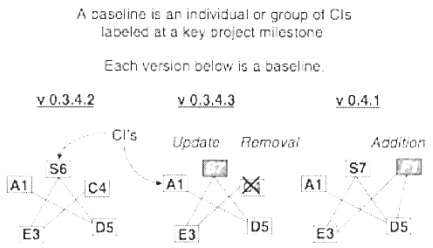


Figure 6.3 Transitioning from one baseline to the next

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

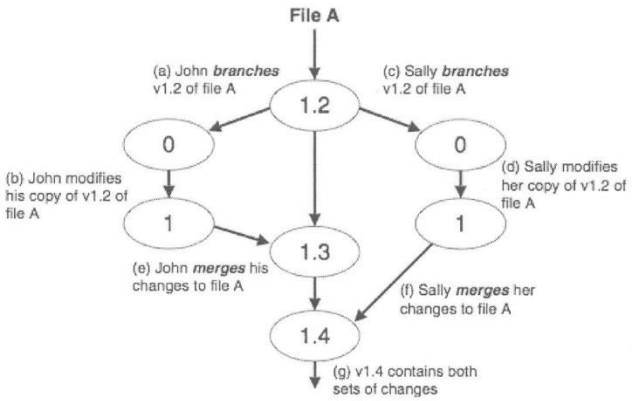


Figure 6.4 Branching and merging changes

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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.

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

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

Source: Braude, M.J. & Bernstein, M.E. (2011). *Software Engineering: Modern Approaches (2nd ed.)*. John Wiley & Sons.

- **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