CHAPTER

9 SOFTWARE CONFIGURATION MANAGEMENT

KEY CONCEPTS

Change is inevitable when computer software is built. And change increases the level of confusion among software engineers who are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error. Babich [BAB86] discusses this when he states:

> The art of coordinating software development to minimize . . . confusion is called *configuration management.* Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put

QUICK LOOK

What is it? When you build computer software, change happens. And because it happens, you need to control it effectively. Software configuration management (SCM) is a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Who does it? Everyone involved in the software engineering process is involved with SCM to some extent, but specialized support positions are sometimes created to manage the SCM process.

Why is it important? If you don't control change, it controls you. And that's never good. It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. For that reason, SCM is an essential part of good project management and solid software engineering practice.

What are the steps? Because many work products are produced when software is built, each must be uniquely identified. Once this is accomplished, mechanisms for version and change control can be established. To ensure that quality is maintained as changes are made, the process is audited; and to ensure that those with a need to know are informed about changes, reporting is conducted.

QUICK LOOK

What is the work product? The Software Configuration Management Plan defines the project strategy for SCM. In addition, when formal SCM is invoked, the change control process produces software change requests and reports and engineering change orders.

How do I ensure that I've done it right? When every work product can be accounted for, traced, and controlled; when every change can be tracked and analyzed; when everyone who needs to know about a change has been informed—you've done it right.

into operation. Software configuration management is a set of tracking and control activities that begin when a software engineering project begins and terminate only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter, we discuss the specific activities that enable us to manage change.

9.1 SOFTWARE CONFIGURATION MANAGEMENT

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms); (2) documents that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration.*

Quote:

"There is nothing permanent except change."

Heraclitus
500 B.C.

As the software process progresses, the number of *software configuration items* (SCIs) grows rapidly. A *System Specification* spawns a *Software Project Plan* and *Software Requirements Specification* (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change.* Change may occur at any time, for any reason. In fact, the First Law of System Engineering [BER80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.

- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

9.1.1 Baselines

Most software changes are justified. Don't bemoan changes. Rather, be certain that you have mechanisms in place to handle them.

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A *baseline* is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

One way to describe a baseline is through analogy:

> Consider the doors to the kitchen in a large restaurant. One door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction.
>
> If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.
>
> If, however, he leaves the kitchen, gives the customer the dish and then is informed of his error, he must follow a set procedure: (1) look at the check to determine if an error has occurred, (2) apologize profusely, (3) return to the kitchen through the IN door, (4) explain the problem, and so forth.

A software engineering work product becomes a baseline only after it has been reviewed and approved.

A baseline is analogous to the kitchen doors in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, we figuratively pass through a swinging one-way door. Changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

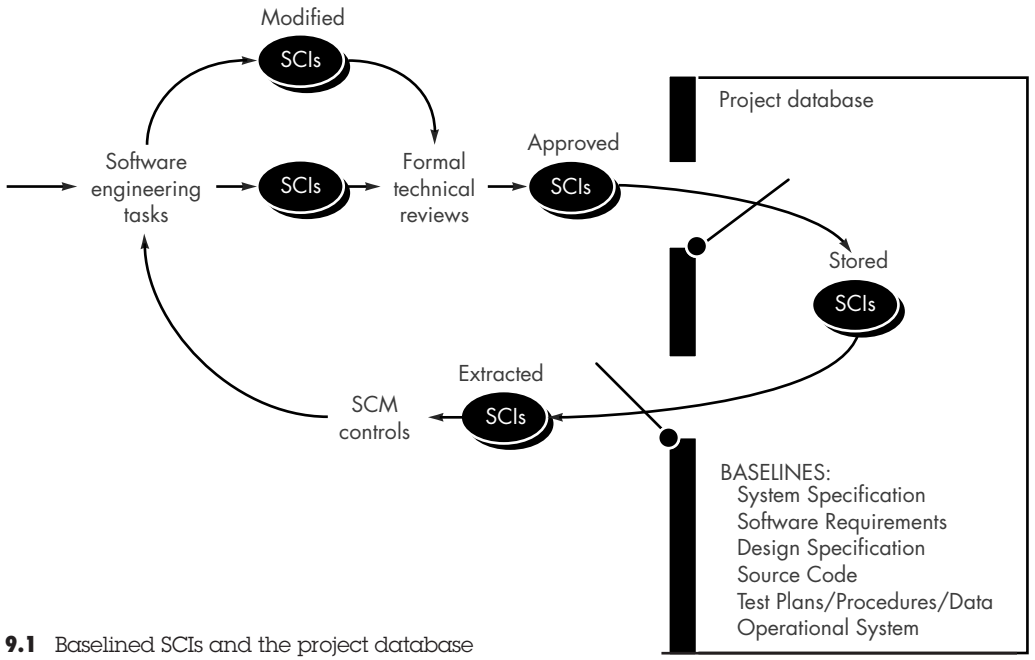


FIGURE 9.1 Baselined SCIs and the project database

In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review (Chapter 8). For example, the elements of a *Design Specification* have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed, corrected and then approved, the *Design Specification* becomes a baseline. Further changes to the program architecture (documented in the *Design Specification*) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 9.1.

Be sure that the project database is maintained in a centralized, controlled location.

The progression of events that lead to a baseline is also illustrated in Figure 9.1. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository*). When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private work space. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 9.1 illustrate the modification path for a baselined SCI.

9.1.2 Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of a large specification or one test case in a large suite of

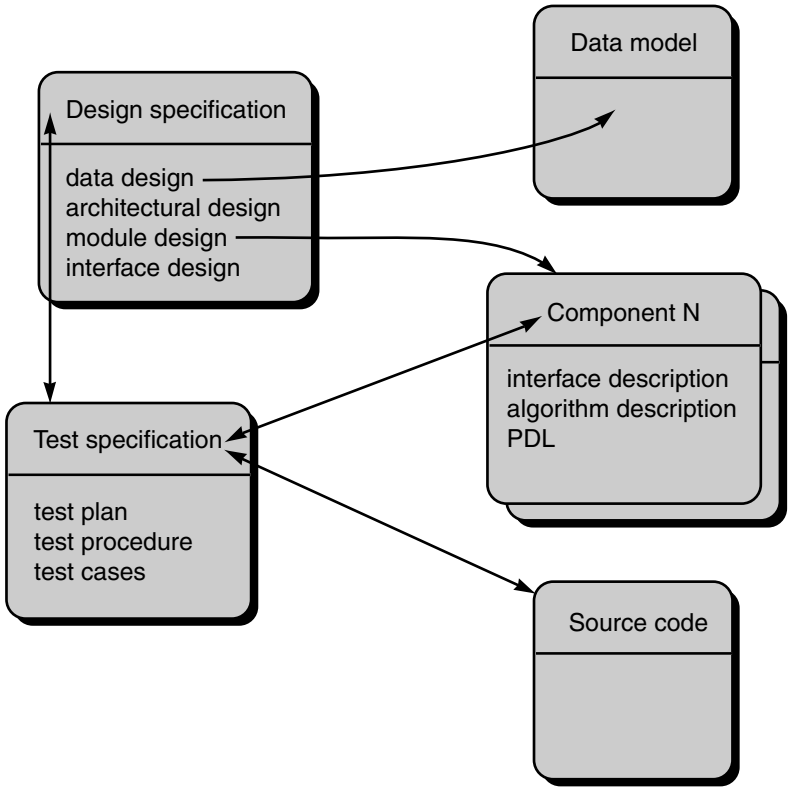


FIGURE 9.2
Configuration objects

Software Configuration Items

tests. More realistically, an SCI is a document, a entire suite of test cases, or a named program component (e.g., a C++ function or an Ada package).

In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form *configuration objects* that may be cataloged in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 9.2, the configuration objects, **Design Specification, data model, component N, source code** and **Test Specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a *compositional relation.* That is, **data model** and **component N** are part of the object **Design Specification.** A double-headed straight arrow indicates an interrelationship.

If a change were made to the source code object, the interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.[1]

9.2 THE SCM PROCESS

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software, the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration.

Any discussion of SCM introduces a set of complex questions:

- How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

These questions lead us to the definition of five SCM tasks: *identification, version control, change control, configuration auditing,* and *reporting.*

WebRef

The ***Configuration Management Yellow Pages*** contains the most comprehensive listing of SCM resources on the Web at **www.cs.colorado.edu/users/andre/configuration_management.html**

9.3 IDENTIFICATION OF OBJECTS IN THE SOFTWARE CONFIGURATION

To control and manage software configuration items, each must be separately named and then organized using an object-oriented approach. Two types of objects can be identified [CHO89]: *basic objects* and *aggregate objects.*[2] A basic object is a "unit of text" that has been created by a software engineer during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, a source listing for a component, or a suite of test cases that are used to exercise the code. An aggregate object is a collection of basic objects and other aggregate objects. Referring to Figure 9.2, **Design Specification** is an aggregate object. Conceptually, it can be viewed as a named (identified) list of pointers that specify basic objects such as **data model** and **component N**.

Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify

1 These relationships are defined within the database. The structure of the project database will be discussed in greater detail in Chapter 31.

2 The concept of an aggregate object [GUS89] has been proposed as a mechanism for representing a complete version of a software configuration.

- the SCI type (e.g., document, program, data) represented by the object
- a project identifier
- change and/or version information

KEY POINT

The interrelationships established for configuration objects allow a software engineer to assess the impact of change.

Resources are "entities that are provided, processed, referenced or otherwise required by the object [CHO89]." For example, data types, specific functions, or even variable names may be considered to be object resources. The realization is a pointer to the "unit of text" for a basic object and null for an aggregate object.

Configuration object identification must also consider the relationships that exist between named objects. An object can be identified as **<part-of>** an aggregate object. The relationship **<part-of>** defines a hierarchy of objects. For example, using the simple notation

E-R diagram 1.4 <part-of> data model;
data model <part-of> design specification;

we create a hierarchy of SCIs.

XRef

Data models and data flow diagrams are discussed in Chapter 12.

It is unrealistic to assume that the only relationships among objects in an object hierarchy are along direct paths of the hierarchical tree. In many cases, objects are interrelated across branches of the object hierarchy. For example, a data model is interrelated to data flow diagrams (assuming the use of structured analysis) and also interrelated to a set of test cases for a specific equivalence class. These cross structural relationships can be represented in the following manner:

data model <interrelated> data flow model;
data model <interrelated> test case class m;

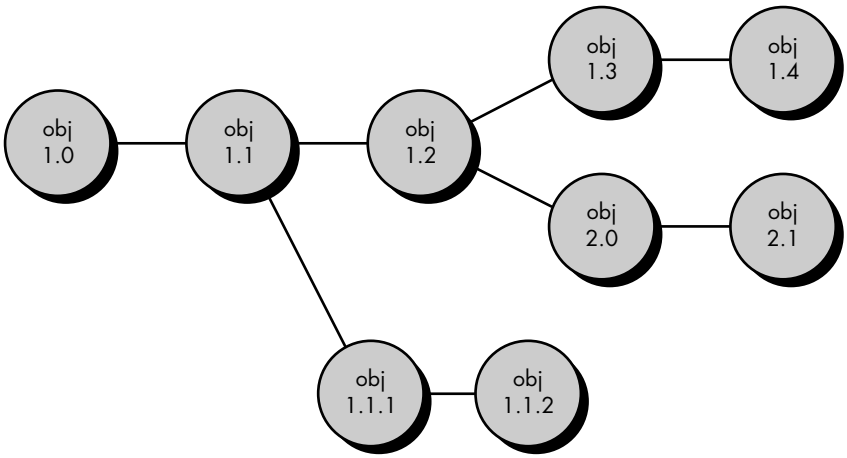
In the first case, the interrelationship is between a composite object, while the second relationship is between an aggregate object (**data model**) and a basic object (**test case class m**).

The interrelationships between configuration objects can be represented with a *module interconnection language* (MIL) [NAR87]. A MIL describes the interdependencies among configuration objects and enables any version of a system to be constructed automatically.

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent. It is possible to create an *evolution graph* [GUS89] for any object. The evolution graph describes the change history of an object, as illustrated in Figure 9.3. Configuration object 1.0 undergoes revision and becomes object 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The evolution of object 1.0 continues through 1.3 and 1.4, but at the same time, a major modification to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.

Changes may be made to any version, but not necessarily to all versions. How does the developer reference all components, documents, and test cases for version 1.4? How does the marketing department know what customers currently have

FIGURE 9.3
Evolution graph



version 2.1? How can we be sure that changes to the version 2.1 source code are properly reflected in the corresponding design documentation? A key element in the answer to all these questions is identification.

A variety of automated SCM tools has been developed to aid in identification (and other SCM) tasks. In some cases, a tool is designed to maintain full copies of only the most recent version. To achieve earlier versions (of documents or programs) changes (cataloged by the tool) are "subtracted" from the most recent version [TIC82]. This scheme makes the current configuration immediately available and allows other versions to be derived easily.

CASE Tools—SCM

9.4 VERSION CONTROL

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. Clemm [CLE89] describes version control in the context of SCM:

> Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes.

The naming scheme you establish for SCIs should incorporate the version number.

These "attributes" mentioned can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system [LIE89].

One representation of the different versions of a system is the evolution graph presented in Figure 9.3. Each node on the graph is an aggregate object, that is, a complete version of the software. Each version of the software is a collection of SCIs (source code, documents, data), and each version may be composed of different variants. To illustrate this concept, consider a version of a simple program that is com-

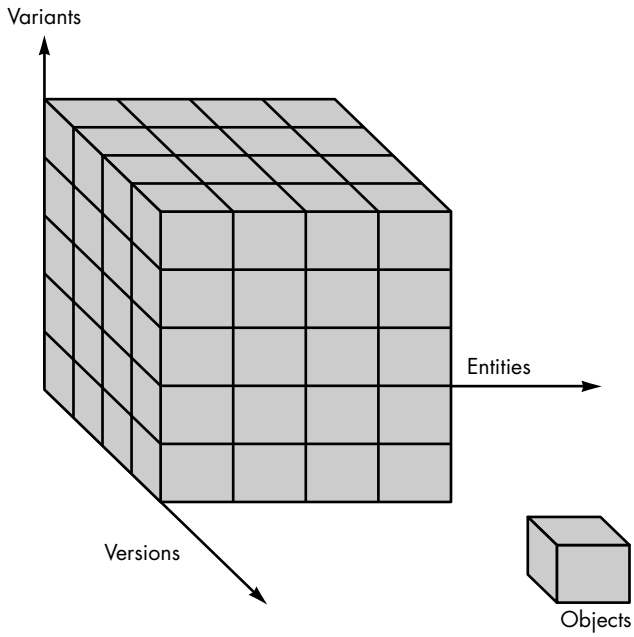


FIGURE 9.4
Object pool representation of components, variants, and versions [REI89]

posed of entities 1, 2, 3, 4, and 5.[3] Entity 4 is used only when the software is implemented using color displays. Entity 5 is implemented when monochrome displays are available. Therefore, two variants of the version can be defined: (1) entities 1, 2, 3, and 4; (2) entities 1, 2, 3, and 5.

Quote:
"Any change, even a change for the better, is always accompanied by drawbacks and discomforts."
Arnold Bennett

To construct the appropriate *variant* of a given version of a program, each entity can be assigned an "attribute-tuple"—a list of features that will define whether the entity should be used when a particular variant of a software version is to be constructed. One or more attributes is assigned for each variant. For example, a color attribute could be used to define which entity should be included when color displays are to be supported.

Another way to conceptualize the relationship between entities, variants and versions (revisions) is to represent them as an *object pool* [REI89]. Referring to Figure 9.4, the relationship between configuration objects and entities, variants and versions can be represented in a three-dimensional space. An entity is composed of a collection of objects at the same revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants. A new version is defined when major changes are made to one or more objects.

A number of different automated approaches to version control have been proposed over the past decade. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

---

3 In this context, the term *entity* refers to all composite objects and basic objects that exist for a baselined SCI. For example, an "input" entity might be constructed with six different software components, each responsible for an input subfunction.

9.5 CHANGE CONTROL

The reality of *change control* in a modern software engineering context has been summed up beautifully by James Bach [BAC98]:

> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

Quote:

"The art of progress is to preserve order amid change and to preserve change amid order."

Alfred North Whitehead

Bach recognizes that we face a balancing act. Too much change control and we create problems. Too little, and we create other problems.

For a large software engineering project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 9.5. A *change request*[4] is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group who makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and appropriate SQA activities are applied. The object is then "checked in" to the database and appropriate version control mechanisms (Section 9.4) are used to create the next version of the software.

ADVICE

Confusion leads to errors—some of them very serious. Access and synchronization control avoid confusion. Implement them both, even if your approach has to be simplified to accommodate your development culture.

The "check-in" and "check-out" process implements two important elements of change control—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another [HAR89].

Access and synchronization control flow are illustrated schematically in Figure 9.6. Based on an approved change request and ECO, a software engineer *checks out* a configuration object. An access control function ensures that the software engineer has authority to check out the object, and synchronization control *locks* the object in the project database so that no updates can be made to it until the currently checked-out version has been replaced. Note that other copies can be checked-out, but other updates cannot be made. A copy of the baselined object, called the *extracted version,*

4 Although many change requests are submitted during the software support phase, we take a broader view in this discussion. A request for change can occur at any time during the software process.

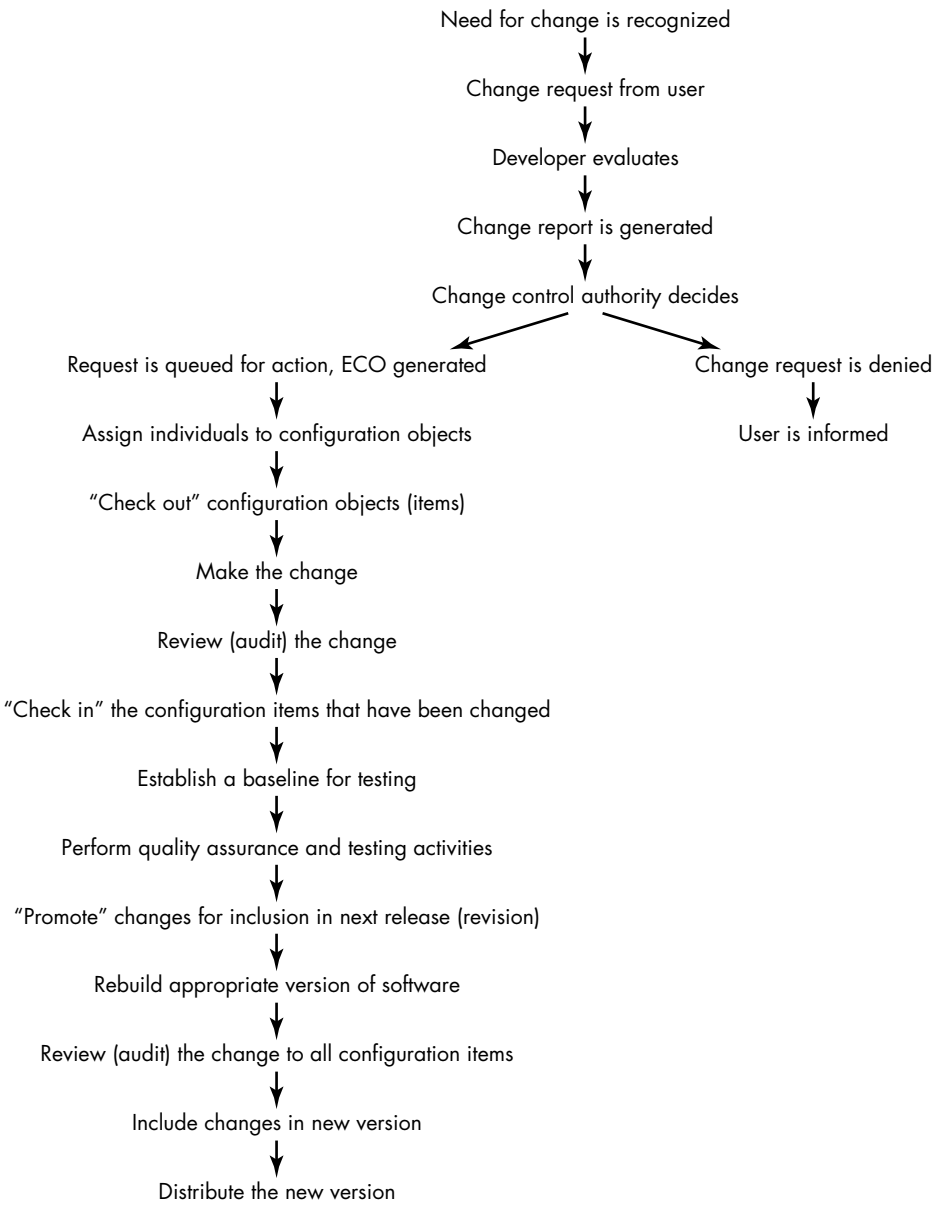


FIGURE 9.5
The change control process

is modified by the software engineer. After appropriate SQA and testing, the modified version of the object is *checked in* and the new baseline object is unlocked.

Some readers may begin to feel uncomfortable with the level of bureaucracy implied by the change control process description. This feeling is not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately,

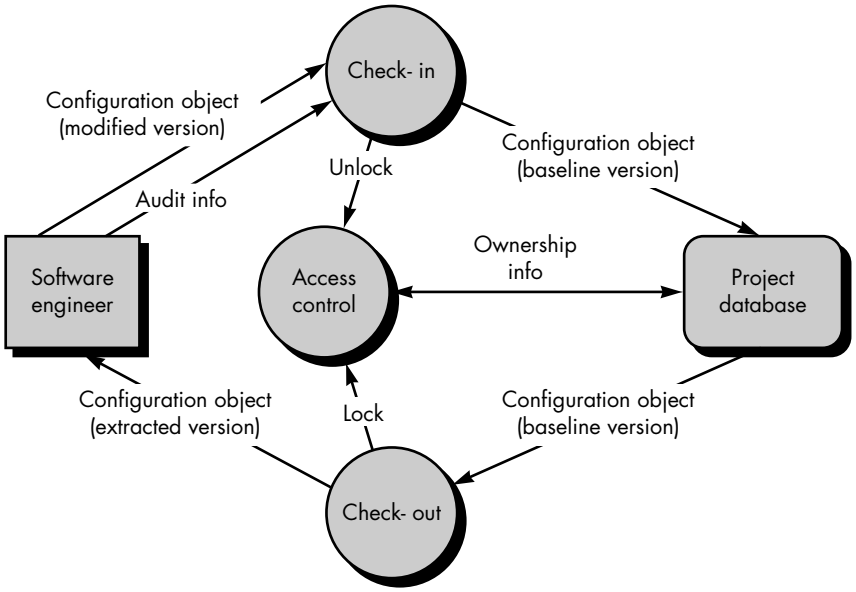


FIGURE 9.6
Access and synchronization control

many have none) have created a number of layers of control to help avoid the problems alluded to here.

Opt for a bit more change control than you think you'll need. It's likely that "too much" will be the right amount.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone formal technical review and has been approved, a baseline is created. Once an SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, formal generation of change requests, change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 9.5.

"Change is inevitable, except from vending machines."

Bumper sticker

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify customer's perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

9.6 CONFIGURATION AUDIT

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review (presented in detail in Chapter 8) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A formal technical review should be conducted for all but the most trivial changes.

? What are the primary questions that we ask during a configuration audit?

A *software configuration audit* complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group.

9.7 STATUS REPORTING

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 9.5. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported

as part of the CSR task. Output from CSR may be placed in an on-line database [TAY85], so that software developers or maintainers can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

Develop a "need to know list" for every SCI and keep it up-to-date. When a change is made, be sure that everyone on the list is informed.

Configuration status reporting plays a vital role in the success of a large software development project. When many people are involved, it is likely that "the left hand not knowing what the right hand is doing" syndrome will occur. Two developers may attempt to modify the same SCI with different and conflicting intents. A software engineering team may spend months of effort building software to an obsolete hardware specification. The person who would recognize serious side effects for a proposed change is not aware that the change is being made. CSR helps to eliminate these problems by improving communication among all people involved.

9.8 SCM STANDARDS

Over the past two decades a number of software configuration management standards have been proposed. Many early SCM standards, such as MIL-STD-483, DOD-STD-480A and MIL-STD-1521A, focused on software developed for military applications. However, more recent ANSI/IEEE standards, such as ANSI/IEEE Stds. No. 828-1983, No. 1042-1987, and Std. No. 1028-1988 [IEE94], are applicable for non-military software and are recommended for both large and small software engineering organizations.

9.9 SUMMARY

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of software engineering becomes part of a software configuration. The configuration is organized in a manner that enables orderly control of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items, that are produced as a result of some software engineering activity. In addition to documents, programs, and data, the development environment that is used to create software can also be placed under configuration control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and composite objects form an object pool from which variants and versions are created. Version control is the set of procedures and tools for managing the use of these objects.

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

REFERENCES

[BAB86] Babich, W.A., *Software Configuration Management,* Addison-Wesley, 1986.
[BAC98] Bach, J., "The Highs and Lows of Change Control," *Computer,* vol. 31, no. 8, August 1998, pp. 113–115.
[BER80] Bersoff, E.H., V.D. Henderson, and S.G. Siegel, *Software Configuration Management,* Prentice-Hall, 1980.
[CHO89] Choi, S.C. and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 66–75.
[CLE89] Clemm, G.M., "Replacing Version Control with Job Control," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 162–169.
[GUS89] Gustavsson, A., "Maintaining the Evoluation of Software Objects in an Integrated Environment," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 114–117.
[HAR89] Harter, R., "Configuration Management," *HP Professional,* vol. 3, no. 6, June 1989.
[IEE94] *Software Engineering Standards,* 1994 edition, IEEE Computer Society, 1994.
[LIE89] Lie, A. et al., "Change Oriented Versioning in a Software Engineering Database," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October, 1989, pp. 56–65.
[NAR87] Narayanaswamy, K. and W. Scacchi, "Maintaining Configurations of Evolving Software Systems," *IEEE Trans. Software Engineering,* vol. SE-13, no. 3, March 1987, pp. 324–334.
[REI89] Reichenberger, C., "Orthogonal Version Management," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 137–140.
[TAY85] Taylor, B., "A Database Approach to Configuration Management for Large Projects," *Proc. Conf. Software Maintenance—1985,* IEEE, November 1985, pp. 15–23.
[TIC82] Tichy, W.F., "Design, Implementation and Evaluation of a Revision Control System," *Proc. 6th Intl. Conf. Software Engineering,* IEEE, Tokyo, September 1982, pp. 58–67.

PROBLEMS AND POINTS TO PONDER

9.1. Why is the First Law of System Engineering true? How does it affect our perception of software engineering paradigms.

9.2. Discuss the reasons for baselines in your own words.

9.3. Assume that you're the manager of a small project. What baselines would you define for the project and how would you control them?

9.4. Design a project database system that would enable a software engineer to store, cross reference, trace, update, change, and so forth all important software configuration items. How would the database handle different versions of the same program? Would source code be handled differently than documentation? How will two developers be precluded from making different changes to the same SCI at the same time?

9.5. Do some research on object-oriented databases and write a paper that describes how they can be used in the context of SCM.

9.6. Use an E-R model (Chapter 12) to describe the interrelationships among the SCIs (objects) listed in Section 9.1.2.

9.7. Research an existing SCM tool and describe how it implements control for versions, variants, and configuration objects in general.

9.8. The relations **<part-of>** and **<interrelated>** represent simple relationships between configuration objects. Describe five additional relationships that might be useful in the context of a project database.

9.9. Research an existing SCM tool and describe how it implements the mechanics of version control. Alternatively, read two or three of the papers on SCM and describe the different data structures and referencing mechanisms that are used for version control.

9.10. Using Figure 9.5 as a guide, develop an even more detailed work breakdown for change control. Describe the role of the CCA and suggest formats for the change request, the change report, and the ECO.

9.11. Develop a checklist for use during configuration audits.

9.12. What is the difference between an SCM audit and a formal technical review? Can their function be folded into one review? What are the pros and cons?

FURTHER READINGS AND INFORMATION SOURCES

One of the few books that have been written about SCM in recent years is by Brown, et al. (*AntiPatterns and Patterns in Software Configuration Management,* Wiley, 1999). The authors discuss the things not to do (antipatterns) when implementing an SCM process and then consider their remedies.

Lyon (*Practical CM: Best Configuration Management Practices for the 21st Century,* Raven Publishing, 1999) and Mikkelsen and Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook,* Allyn & Bacon, 1997) provide pragmatic tutorials on important SCM practices. Ben-Menachem (*Software Configuration Management Guidebook,* McGraw-Hill, 1994), Vacca (*Implementing a Successful Configuration Change Management Program,* I. S. Management Group, 1993), and Ayer and Patrinnostro (*Software Configuration Management,* McGraw-Hill, 1992) present good overviews for those who need further introduction to the subject. Berlack (*Soft-*

ware Configuration Management, Wiley, 1992) presents a useful survey of SCM concepts, emphasizing the importance of the repository and tools in the management of change. Babich [BAB86] provides an abbreviated, yet effective, treatment of pragmatic issues in software configuration management.

Buckley (*Implementing Configuration Management,* IEEE Computer Society Press, 1993) considers configuration management approaches for all system elements—hardware, software, and firmware—with detailed discussions of major CM activities. Rawlings *(SCM for Network Development Environments,* McGraw-Hill, 1994) is the first SCM book to address the subject with a specific emphasis on software development in a networked environment. Whitgift (*Methods and Tools for Software Configuration Management,* Wiley, 1991) contains reasonable coverage of all important SCM topics, but is distinguished by discussion of repository and CASE environment issues. Arnold and Bohner (*Software Change Impact Analysis,* IEEE Computer Society Press, 1996) have edited an anthology that discusses how to analyze the impact of change within complex software-based systems.

Because SCM identifies and controls software engineering documents, books by Nagle (*Handbook for Preparing Engineering Documents: From Concept to Completion,* IEEE, 1996), Watts (*Engineering Documentation Control Handbook: Configuration Management for Industry,* Noyes Publications, 1993), Ayer and Patrinnostro (*Documenting the Software Process,* McGraw-Hill, 1992) provide a complement to more-focused SCM texts. The March 1999 edition of *Crosstalk* contains a number of useful articles on SCM.

A wide variety of information sources on software configuration management and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to SCM can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/scm.mhtml