# Unit 8 Configuration Management

**Structure:**



## 8.1 Introduction

In the last unit we have discussed risk management. In this unit let's discuss configuration management, which is an important activity in project development. 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. In this unit we shall learn how to record various software changes.

**Objectives:**

After studying this unit, you should be able to:

- explain software configuration management process
- identify objects in software configuration
- describe version control and change control
- discuss configuration audit and status reporting
- list goals of software configuration management

## 8.2 Software Configuration Management (SCM)

Babich explains software configuration management as follows:

> Software *configuration management* (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM (software configuration management) activities are developed to
> (1) Identify change
> (2) Control change
> (3) Ensure that change is being properly implemented
> (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. Whereas, software configuration management is a set of activities to control change by identifying the products that are likely to change, establish 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. In short SCM is a methodology to control and manage a software development project.

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.

> 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.

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.*

*As* the software process progresses, the number *of software configuration items (*SCIs) grows rapidly. A *System Specification* spawns a *Software Project Plan* and the *Software Requirements Specification* (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information.

Change may occur at any time, for any reason. The First Law of System Engineering states: "No matter where you are in system life cycle, the system will change, and the desire to change it will span throughout the life cycle."

There are six 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 informal systems,
- Functionality delivered by products.
- Reorganization or business growth/downsizing causes changes in project priorities
- Changes in 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.

### 8.2.1 Baselines

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: (I) look at the check to determine if an error has occurred, (2) apologize to the customer (3) return to the kitchen through the IN door, (4) explain the problem, and so forth.

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.

### 8.2.2 Software configuration items (SCI)

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 be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is a document, an 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 baseline 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 8.1, 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.

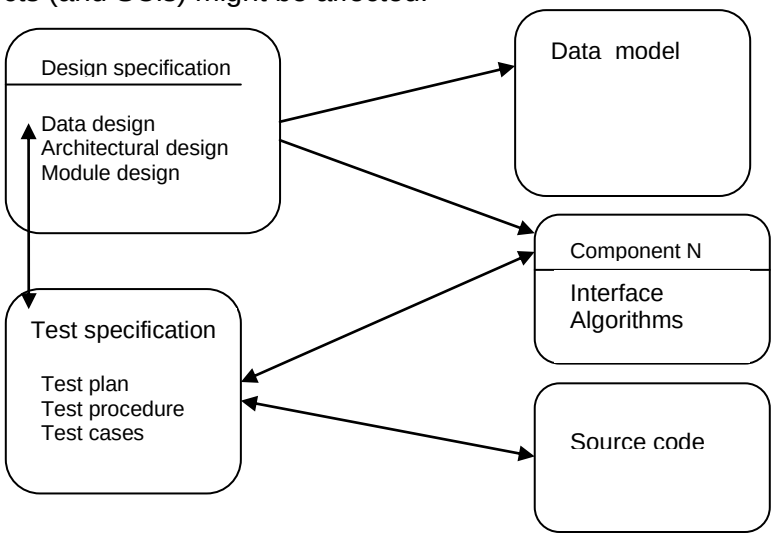


**Fig. 8.1: Configuration Objects**

**Self Assessment Questions**

1. Software Configuration Management (SCM) is an umbrella activity that is applied throughout the software process. (True / False)
2. A _____ spawns a Software Project Plan and the Software Requirements Specification.
3. A _____ has a name, attributes, and is "connected" to other objects by relationships.

## 8.3 SCM Process

Software configuration management (SCM) is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is 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 change 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 apprise others of changes that are made?

## 8.4 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 are identified: *basic objects* and *aggregate objects.* A basic object is a "unit of text" that has been created by a software engineer during analysis, design, code, and test. For example, a basic object might be a section of requirements specific to a source listing for a component, or *a* suite of test cases that are used to exercise code. An aggregate object is a collection of basic objects and other aggregate objects. Referring to figure 8.1, **Design Specification** is an aggregate object. Conceptually it can be

viewed as a named (identified) list of pointers that specify basic objects as **data model and component N.**

Each object has a set of distinct features that identify it uniquely: a name, a design, 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 the SCI type (e.g., document, program, and data) represented by the object project identifier change and/or version information. Resources are entities that are provided, processed, referenced or otherwise required by the object. 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 we can create a hierarchy of SCIs.

**E-R diagram <part-of> data model;**

**Data model <part-of> design specification;**

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). 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 baseline, 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* for any object. The evolution graph describes the change history of an object, as illustrated in figure 8.2. 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 version 2.0 or 1.0? How can we be sure that changes to the version 2.1 source are properly reflected in the corresponding design documentation? A key element in this answer to all these questions is identification.

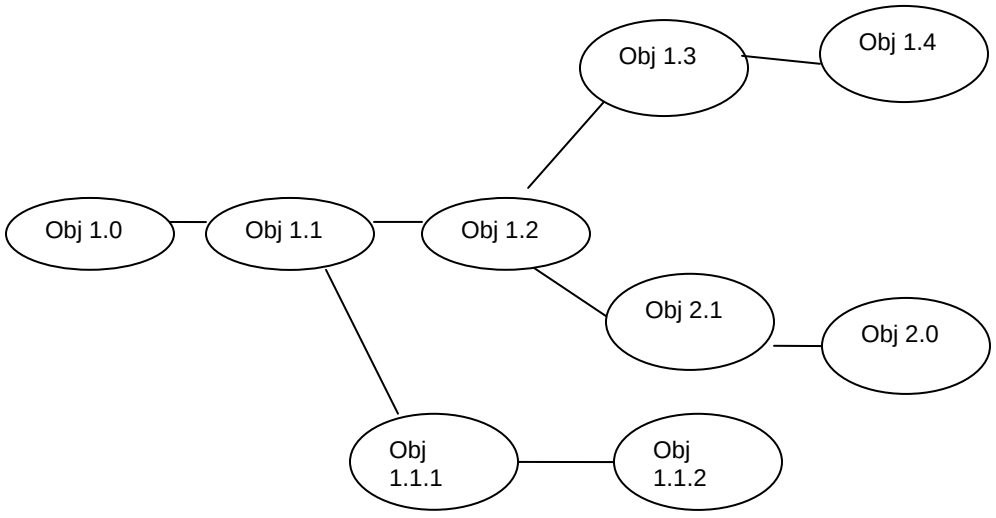


**Fig. 8.2: Evolution graph showing object's change history**

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. Scheme makes the current configuration immediately available and allows other versions to be derived easily.

## 8.5 Version control

Version *control* combines procedures and tools to manage different versions of configuration objects that are created during the software process. Clam describes version control in the context of SCM as follows:

> 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. 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) indicating specific types of functional changes that have been applied to the system.

One representation of the different versions of a system is the evolution graph presented in figure 8.2. Each node on the graph is an aggregate object, that is, a complete version of the software. Each version of the software may be composed of different variants.

To illustrate this concept, consider a version of a simple program that is composed of entities 1, 2, 3, 4, and 5. 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.

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*. Referring to

figure 8.3, 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.

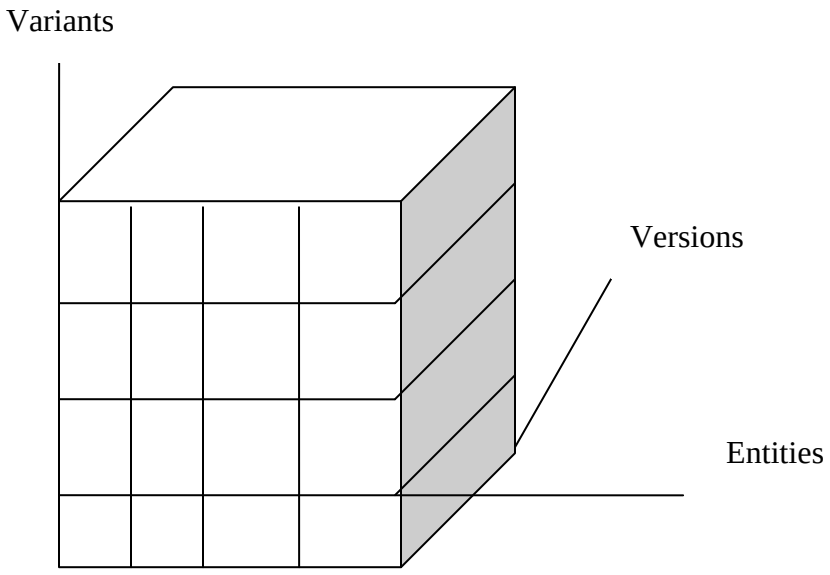


**Fig. 8.3: Object pool representation- Versions, Entities and Variants**

**Self Assessment Questions**

4. Software configuration management is not an element of software quality assurance. (True / False)
5. Two types of objects that can be identified in configuration management are _____ and _____.
6. _____ combines procedures and tools to manage different versions of configuration objects that are created during the software process.

## 8.6 Change Control

The reality of *change control* in modern software engineering context has been summed up beautifully by James Bach as follows:

> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny error 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 developer could sink the project; yet brilliant ideas originate in the minds of those people, and a burdensome change control process could effectively discourage them from doing creative work.

Bach recognizes that we face a balancing act. Too much change control creates lots of 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 contains the following steps:

1) A *change request* 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.
2) 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.
3) 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/or review and audit. The object to be changed is "die 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 are used to create the next version of the software.

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, formed by two different people, don't overwrite

one another. Access and synchronization control flow are illustrated schematically in figure 8.4. A software engineer checks out a configuration project based on the approved change request and ECO.

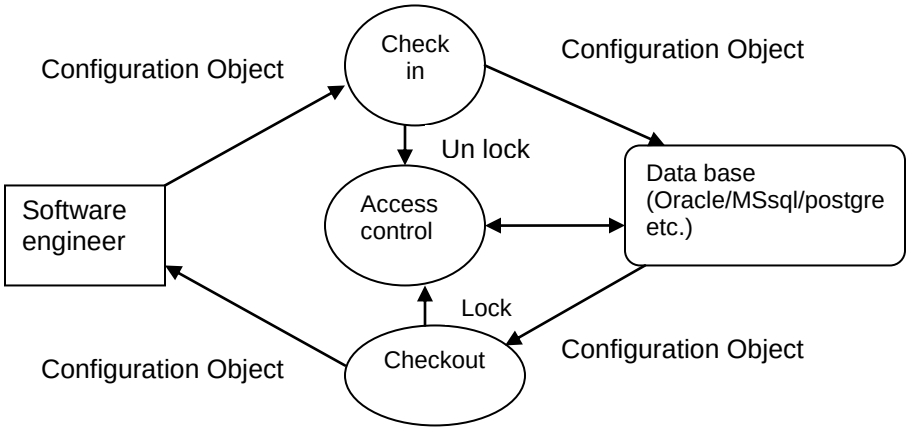


**Fig. 8.4: Flow of access and synchronization control**

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 check out version has been replaced. Note that other copies can be checked-out, but other updates cannot be made. A copy of the baseline object, called the extracted version 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.

Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately, many have one) have created a number of layers of control to help avoid the problems alluded here.

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 what ever changes are justified by project and technical requirements (as long as change not affects broader system requirements that lie outside the developer's scope of work). Once the object has, undergone formal technical review and has been approved 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.

## 8.7 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 two-fold:

(1) Formal Technical Reviews (FTR)

(2) The software 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 formal technical review 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.

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?

## 8.8 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?

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. Status reporting helps to eliminate the problems by improving communication among all people involved.

## 8.9 Goals of SCM

- Configuration Identification – What code are we working with?
- Configuration Control – Controlling the release of a product and its changes.
- Status Accounting – Recording and reporting the status of components.
- Review – Ensuring completeness and consistency among components.
- Build Management  Managing the process and tools used for builds.
- Process Management – Ensuring adherence to the organization's development process.
- Environment Management – Managing the software and hardware that host our system.
- Teamwork – Facilitate team interactions related to the process.
- Defect Tracking – making sure every defect has traceability back to the source

**Self Assessment Questions**

7. For a large software engineering project, uncontrolled change rapidly leads to chaos.  (True / False)

8. The results of the evaluation are presented as a *change report,* which is used by a _____.
9. _____ helps to eliminate the problems by improving communication among all people involved.

## 8.10 Summary

Let's summarize the important points covered in the unit:

- Software configuration management is an umbrella activity that is applied thought 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 a 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 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 baseline. Changes to a baseline object result in the creation of a new version object. The evolution of a program can be tracked by examining the revision of all configuration objects. Version control is the set of procedures, 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 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.

## 8.11 Terminal Questions

1. What is configuration management? Explain in detail.
2. Explain software configuration management process in detail.
3. Explain the goals of the SCM.
4. What is the difference between SCM audit and a formal technical review?
5. What happens to software code, when too many changes occur? Explain in your words.
6. What is version control? Explain how it helps to reduce too many changes.

## 8.12 Answers

### Self Assessment Questions

1. True
2. System Specification
3. Configuration Object
4. False
5. Base Objects, Aggregate Objects
6. Version Control
7. True
8. Change Control Authority
9. Status Reporting

### Terminal Questions

1. Software configuration management is a set of activities to control change by identifying the products that are likely to change, establish 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. (Refer Section 8.2)
2. Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is 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. (Refer Section 8.3)
3. Configuration Identification, Configuration Control, Status Accounting. (Refer Section 8.9)

4. The formal technical review 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. (Refer Section 8.7)
5. Too much change control creates lots of 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. (Refer Section 8.6)
6. Version *control* combines procedures and tools to manage different versions of configuration objects that are created during the software process. (Refer Section 8.5)