SOFTWARE PROJECT PLANNING & MANAGEMENT PHASE

Planning may be the most important management activity. Without a proper plan, no real monitoring or controlling of the project is possible. The basic goal of planning is to look into the future, identify the activities that need to be done to complete the project successfully, and plan the scheduling and resources. The inputs to the planning activity are the requirements specification and the architecture description. A very detailed requirements document is not essential for planning, but for a good plan all the important requirements must be known, and it is highly desirable that architecture decisions have been taken.

System Analysis

Systems analysis is the study of a system under consideration (which may be real or imagined). Its purpose is understanding and documentation of the essential characteristics of the system being studied. Its eventual goal is to come up with a specification of the system under study. A model provides the blueprints of a system. A good model includes all the effective components in the system under study and ignores all elements that are not relevant. Modeling provides a better understanding and visualization of the system under study. Every system can be described from different aspects using different models. A model may emphasize components and their relationship to other components in the system under study, the behavior of each component in the system or the interactions among the components in the system under study. Analysts represent the data of the system under study in terms of metadata concepts and relationships. The representations of data that are commonly used in systems analysis are conceptual data modeling, like Entity Relationship Model, Object-Role Modeling, Object-Relationship Model and Unified Modeling Language (UML); knowledge representation techniques are also used like semantic networks where object-oriented concepts are used to represent knowledge making use of classes, class properties, object instances, and inheritance.

Feasibility Analysis

A major but optional activity within systems analysis is feasibility analysis. Information systems development projects are usually subjected to one or more feasibility analyses prior to and during their life. In an information systems development project context, feasibility is the measure of how beneficial the development or enhancement of an information system would be to the business. Feasibility analysis is the process by which feasibility is measured. It is an ongoing process done frequently during systems development projects. Knowing a project's current status at strategic points in time gives us and the user the opportunity to (1) continue the project as planned, (2) make changes to the project, or (3) cancel the project.

Feasibility Types: Information systems development projects are subjected to at least three interrelated feasibility types—**operational feasibility, technical feasibility,** and **economic feasibility**. **Operational feasibility** is the measure of how well particular information systems will work in a given environment (operational scope of the software to be developed). Just because XYZ Corporation's payroll clerks all have PCs that can display and allow editing of payroll data doesn't necessarily mean that ABC Corporation's payroll clerks can do the same thing. Part of the feasibility analysis study would be to assess the current capability of ABC Corporation's payroll clerks in order to determine the next best transition for them. Depending on the current situation, it might take one or more interim upgrades prior to them actually getting the PCs for display and editing of payroll data. Historically, of the three types of feasibility, operational feasibility is the one that is most often overlooked, minimized, or assumed to be okay. For example, several years ago many supermarkets installed "talking" point of sale terminals only to discover that customers did not like having people all around them hearing the names of the products they were purchasing. Nor did the cashiers like to hear all of those talking point of sale terminals because they were very distracting. Now the point of sale terminals are once again mute.

Technical feasibility is the measure of the practicality of a specific technical information system solution and the availability of technical resources. Often new technologies are solutions looking for a problem to solve. As voice recognition systems become more sophisticated, many businesses will consider this technology as a possible solution for certain information systems applications. When CASE technology was first introduced in the mid 1980s, many businesses decided it was impractical for them to adopt it for a variety of reasons, among them being the limited availability of the technical expertise in the marketplace to use it. Adoption of Smalltalk, C++, and other object oriented programming for business applications is slow for similar reasons.

Economic feasibility is the measure of the cost effectiveness of an information system solution. Without a doubt, this measure is most often the most important one of the three. Information systems are often viewed as capital

investments for the business, and, as such, should be subjected to the same type of investment analyses as other capital investments. Financial analyses such as return on investment (ROI), internal rate of return (IRR), cost/benefit, payback period, and the time value of money are utilized when considering information system development projects. Cost/benefit analysis identifies the costs of developing the information system and operating it over a specified period of time. It also identifies the benefits in financial terms in order to compare them with the costs. Economically speaking, when the benefits exceed the costs, the system has economic value to the business; just how much value is a function of management's perspective on investments.

The major issues project planning addresses are:

1) **Process planning**
2) **Effort estimation**
3) **Schedule and Resource Estimation**
4) **Quality plans**
5) **Configuration management plans**
6) **Risk management**
7) **Project monitoring plans**

1) Process planning

For a project, during planning, a key activity is to plan and specify the process that should be used in the project. This means specifying the various stages in the process, the entry criteria for each stage, the exit criteria, and the verification activities that will be done at the end of the stage. In an organization, often standard processes are defined and a project can use any of these standard processes and tailor it to suit the specific needs of the project. Hence the process planning activity mostly entails selecting a standard process and tailoring it for the project. Generally, however, based on the size, complexity and nature of the project, as well as the characteristics of the team, like the experience of the team members with the problem domain as well as the technology being used, the standard process is tailored. The common tailoring actions are modifying a step, omit a step, add a step, or change the formality with which a step is done. After tailoring, the process specification for the project is available. This process guides rest of the planning, particularly detailed scheduling where detailed tasks to be done in the project are defined and assigned to people to execute them.

2) Effort estimation

For a given set of requirements it is desirable to know how much it will cost to develop the software, and how much time the development will take. These estimates are needed before development is initiated. The primary reason for cost and schedule estimation is cost-benefit analysis, and project monitoring and control. A more practical use of these estimates is in bidding for software projects, where cost estimates must be given to a potential client for the development contract. The bulk of the cost of software development is due to the human resources needed, and therefore most cost estimation procedures focus on estimating effort in terms of person-months (PM). By properly including the "overheads" (i.e., the cost of hardware, software, office space, etc.) in the cost of a person-month, effort estimates can be converted into cost. For a software development project, effort and schedule estimates are essential prerequisites for managing the project. Effort and schedule estimates are also required to determine the staffing level for a project during different phases.

One can perform effort estimation at any point in the software life cycle. As the effort of the project depends on the nature and characteristics of the project, at any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product. Clearly, when the product is delivered, the effort can be accurately determined, as all the data about the project and the resources spent can be fully known by then. This is effort estimation with complete knowledge about the project. On the other extreme is the point when the project is being initiated or during the feasibility study. At this time, we have only some idea of the classes of data the system will get and produce and the major functionality of the system. There is a great deal of uncertainty about the actual specifications of the system. Specifications with uncertainty represent a range of possible final products, not one precisely defined product. Hence, the effort estimation based on this type of information cannot be accurate. Estimates at this phase of the project can be off by as much as a factor of four from the actual final effort. As we specify the system more fully and accurately, the uncertainties are reduced and more accurate effort estimates can be made. For example, once the requirements are completely specified, more accurate effort estimates can be made

compared to the estimates after the feasibility study. Once the design is complete, the estimates can be made still more accurately.

For actual effort estimation, estimation models or procedures have to be developed. The accuracy of the estimates will depend on the effectiveness and accuracy of the estimation procedures or models employed and the process (i.e., how predictable it is). Despite the limitations, estimation models have matured considerably and generally give fairly accurate estimates. For example, when the COCOMO model (discussed later) was checked with data from some projects, it was found that the estimates were within 20% of the actual effort 68% of the time. It should also be mentioned that achieving an estimate after the requirements have been specified within 20% is actually quite good. With such an estimate, there need not even be any cost and schedule overruns, as there is generally enough slack or free time available (recall the study mentioned earlier that found a programmer spends more than 30% of his time in personal or miscellaneous tasks) that can be used to meet the targets set for the project based on the estimates. In other words, if the estimate is within 20% of the actual, the effect of this inaccuracy will not even be reflected in the final cost and schedule.

Building Effort Estimation Models: An estimation model can be viewed as a "function" that outputs the effort estimate, clearly this estimation function will need inputs about the project, from which it can produce the estimate. The basic idea of having a model or procedure for estimation is that it reduces the problem of estimation to estimating or determining the value of the "key parameters" that characterize the project, based on which the effort can be estimated. Note that an estimation model does not, and cannot, work in a vacuum; it needs inputs to produce the effort estimate as output. At the start of a project, when the details of the software itself are not known, the hope is that the estimation model will require values of parameters that can be measured at that stage. Although the effort for a project is a function of many parameters, it is generally agreed that the primary factor that controls the effort is the size of the project, that is, the larger the project, the greater the effort requirement. One common approach therefore for estimating effort is to make it a function of *project size*, and the equation of effort is considered as:

$$\text{EFFORT} = a * \text{SIZE}^b$$

Where *a* and *b* are constants, and project size is generally in KLOC or function points. Values for these constants for a particular process are determined through regression analysis, which is applied to data about the projects that has been performed in the past.

Size estimation is often easier than direct effort estimation. For estimating size, the system is generally partitioned into components it is likely to have. Once the components of the system are known, as estimating something about a small unit is generally much easier than estimating it for a larger system, sizes of components can be generally estimated quite accurately. Once size estimates for components are available, to get the overall size estimate for the system, the estimates of all the components can be added up. Similar property does not hold for effort estimation, as effort for developing a system is not the sum of effort for developing the components (as additional effort is needed for integration and other such activities when building a system from developed components). Effort estimation models may be top down or bottom up. The approach of determining total effort from the total size is referred to as the top-down approach, as overall effort is first determined and then from this the effort for different parts are obtained. In bottom-up approach, the project is first divided into tasks and then estimates for the different tasks of the project are first obtained. From the estimates of the different tasks, the overall estimate is determined. Both the top-down and the bottom-up approaches require information about the project: size (for top-down approaches) or a list of tasks (for bottom-up approaches).

COCOMO Model

A top-down model can depend on many different factors. Instead of depending only on one variable, giving rise to multivariable models, one approach for building multivariable models is to start with an initial estimate determined by using the static single-variable model equations, which depend on size, and then adjusting the estimates based on other variables. This approach implies that size is the primary factor for cost; other factors have a lesser effect. Here we will discuss one such model called the COnstructive COst MOdel (COCOMO) developed by Boehm. This model also estimates the total effort in terms of person-months. The basic steps in this model are:

1. Obtain an initial estimate of the development effort from the estimate of thousands of delivered lines of source code (KLOC).
2. Determine a set of 15 multiplying factors from different attributes of the project.
3. Adjust the effort estimate by multiplying the initial estimate with all the multiplying factors.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single-variable models, using KLOC as the measure of size. To determine the initial effort E_i in person-months the

equation used is of the type $E_i = a * (KLOC)^b$. The value of the constants *a* and *b* depend on the project type. In COCOMO, projects are categorized into three types—organic, semidetached, and embedded. These categories roughly characterize the complexity of the project with organic projects being those that are relatively straightforward and developed by a small team, and embedded are those that are ambitious and novel, with stringent constraints from the environment and high requirements for such aspects as interfacing and reliability. The constants a and b for different systems are:

System	a	b
Organic	3.2	1.05
Semidetached	3.0	1.12
Embedded	2.8	1.20

The value of the constants for a cost model depend on the process and have to be determined from past data. COCOMO has instead provided "global" constant values. These values should be considered as values to start with until data for some projects is available. With project data, the value of the constants can be determined through regression analysis. There are 15 different attributes, called cost driver attributes that determine the multiplying factors. These factors depend on product, computer, personnel, and technology attributes (called project attributes). Examples of the attributes are required software reliability (RELY), product complexity (CPLX), analyst capability (ACAP), application experience (AEXP), use of modern tools (TOOL), and required development schedule (SCHD), Each cost driver has a rating scale, and for each rating, a multiplying factor is provided. The COCOMO approach also provides guidelines for assessing the rating for the different attributes. The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF). The final effort estimate, E, is obtained by multiplying the initial estimate by the EAF. That is, $E = EAF * E_i$ ... By this method, the overall cost of the project can be estimated. For planning and monitoring purposes, estimates of the effort required for the different phases is also desirable. In COCOMO, effort for a phase is a defined percentage of the overall effort. The percentage of total effort spent in a phase varies with the type and size of the project. The percentages for an organic software project are given in table below.

Cost Drivers	*Rating*				
	Very Low	Low	Nom-inal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Table : Effort multipliers for different cost drivers.

As an example, suppose a system for office automation has to be designed. From the requirements, it is clear that there will be four major modules in the system: data entry, data update, query, and report generator. It is also clear from the requirements that this project will fall in the organic category. The sizes for the different modules and the overall system are estimated to be:

Data Entry	0.6 KLOC
Data Update	0.6 KLOC
Query	0.8 KLOC
Reports	1.0 KLOC
TOTAL	3.0 KLOC

From the requirements, the ratings of the different cost driver attributes are assessed. These ratings, along with their multiplying factors, are:

Complexity	High	1.15
Storage	High	1.06
Experience	Low	1.13
Programmer Capability	Low	1.17

All other factors had a nominal rating. From these, the effort adjustment factor (EAF) is

$$\text{EAF} = 1.15*1.06*1.13*1.17 = 1.61.$$

The initial effort estimate for the project is obtained from the relevant equations. We have

$$E_i = 3.2 * 3^{1.05} = 10.14PM.$$

Using the EAF, the adjusted effort estimate is

$$E = 1.61 * 10.14 = 16.3PM.$$

Using the preceding table, we obtain the percentage of the total effort consumed in different phases. The office automation system's size estimate is 3 KLOC, so we will have to use interpolation to get the appropriate percentage (the two end values for interpolation will be the percentages for 2 KLOC and 8 KLOC). The percentages for the different phases are: design—16%, detailed design—25.83%, code and unit test—41.66%, and integration and testing—16.5%. With these, the effort estimates for the different phases are:

System Design	.16 * 16.3 = 2.6 PM
Detailed Design	.258 * 16.3 = 4.2 PM
Code and Unit Test	.4166 * 16.3 = 6.8 PM
Integration	.165 * 16.3 = 2.7 PM.

Using this table, the estimate of the effort required for each phase can be determined from the total effort estimate. For example, if the total effort estimate for an organic software system is 20 PM and the size estimate is 20KLOC, then the percentage effort for the coding and unit testing phase will be 40 + (38 - 40) / (32 - 8) * 20 - 39%. The estimate for the effort for this phase is 7.8 PM. In COCOMO the detailed design and code and unit testing are sometimes combined into one phase called the *programming phase*.

3) Schedule and Resource Estimation

Once the effort is estimated, various schedules (or project duration) are possible, depending on the number of resources (people) put on the project. For example, for a project whose effort estimate is 56 person-months, a total

schedule of 8 months is possible with 7 people. A schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people. As is well known, however, manpower and months are not fully interchangeable in a software project. A schedule cannot be simply obtained from the overall effort estimate by deciding on average staff size and then determining the total time requirement by dividing the total effort by the average staff size. Once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited. In a project, the scheduling activity can be broken into two sub activities: determining the overall schedule (the project duration) with major milestones, and developing the detailed schedule of the various tasks. One method to determine the normal (or nominal) overall schedule is to determine it as a function of effort. Any such function has to be determined from data from completed projects using statistical techniques like fitting a regression curve through the scatter plot obtained by plotting the effort and schedule of past projects. This curve is generally nonlinear because the schedule does not grow linearly with effort. In COCOMO, the equation for schedule for an organic type of software is

$$M = 2.5 * E^{.38}$$

It should be clear that schedule is not a function solely of effort. However, it can be used as a guideline or check of the schedules reasonableness, which might be decided based on other factors. From the macro estimate of schedule, we have to determine the schedule for the major milestones in the project. To determine the milestones, we must first understand the manpower ramp-up that usually takes place in a project. The number of people in a software project tends to follow the Rayleigh curve. That is, in the beginning and the end, few people work on the project; the peak team size (PTS) is reached somewhere near the middle of the project. This behavior occurs because only a few people are needed in the initial phases of requirements analysis and design. The human resources requirement peaks during coding and unit testing. Again, during system testing and integration, fewer people are required. Often, the staffing level is not changed continuously in a project and the required people are assigned together around the start of the project. This approach can lead to some people being unoccupied at the start and toward the end. This slack time is often used for supporting project activities like training and documentation. Given the effort estimate for a phase, we can determine the duration of the phase if we know the manpower ramp-up. Generally speaking, design requires about a quarter of the schedule, build consumes about half, and integration and system testing consume the remaining quarter. COCOMO gives 19% for design, 62% for programming, and 18% for integration.

Once the milestones and the resources are fixed, it is time to set the detailed scheduling. For detailed schedules, the major tasks fixed while planning the milestones are broken into small schedulable activities in a hierarchical manner. For example, the detailed design phase can be broken into tasks for developing the detailed design for each module, review of each detailed design, fixing of defects found, and so on. For each detailed task, the project manager estimates the time required to complete it and assigns a suitable resource so that the overall schedule is met. At each level of refinement, the project manager determines the effort for the overall task from the detailed schedule and checks it against the effort estimates. If this detailed schedule is not consistent with the overall schedule and effort estimates, the detailed schedule must be changed. Once the overall schedule is fixed, detailing for a phase may only be done at the start of that phase. For detailed scheduling, tools like Microsoft Project or a spreadsheet can be very useful. For each lowest-level activity, the project manager specifies the effort, duration, start date, end date, and resources. Dependencies between activities, due either to an inherent dependency (for example, you can conduct a unit test plan for a program only after it has been coded) or to a resource-related dependency (the same resource is assigned two tasks) may also be specified. From these tools the overall effort and schedule of higher level tasks can be determined. A detailed project schedule is never static. Changes may be needed because the actual progress in the project may be different from what was planned, because newer tasks are added in response to change requests, or because of other unforeseen situations. Changes are done as and when the need arises. The final schedule, frequently maintained using some suitable tool, is often the most "live" project plan document. During the project, if plans must be changed and additional activities must be done, after the decision is made, the changes must be reflected in the detailed schedule, as this reflects the tasks actually planned to be performed. Hence, the detailed schedule becomes the main document that tracks the activities and schedule.

Staffing and Team Structure

The number of resources is fixed when schedule is being planned. Detailed scheduling is done only after actual assignment of people has been done, as task assignment needs information about the capabilities of the team members. We have implicitly assumed that the project's team is led by a project manager, who does the planning and task assignment. This form of hierarchical team organization is fairly common, and was earlier called the Chief Programmer Team. In this hierarchical organization, the project manager is responsible for all major technical decisions of the project. He does most of the design and assigns coding of the different parts of the design to the programmers. The team typically consists of programmers, testers, a configuration controller, and possibly a

librarian for documentation. There may be other roles like database manager, network manager, backup project manager, or a backup configuration controller. It should be noted that these are all logical roles and one person may do multiple such roles.

For a small project, a one-level hierarchy suffices. For larger projects, this organization can be extended easily by partitioning the project into modules, and having module leaders who are responsible for all tasks related to their module and have a team with them for performing these tasks. A different team organization is the egoless team. Egoless teams consist of ten or fewer programmers. The goals of the group are set by consensus, and input from every member is taken for major decisions. Group leadership rotates among the group members. Due to their nature, egoless teams are sometimes called democratic teams. This structure allows input from all members, which can lead to better decisions for difficult problems. This structure is well suited for long-term research-type projects that do not have time constraints. It is not suitable for regular tasks that have time constraints; for such tasks, the communication in democratic structure is unnecessary and results in inefficiency.

In recent times, for very large product developments, another structure has emerged. This structure recognizes that there are three main task categories in software development—management related, development related, and testing related. It also recognizes that it is often desirable to have the test and development team be relatively independent, and also not to have the developers or tests report to a nontechnical manager. In this structure, consequently, there is an overall unit manager, under whom there are three small hierarchic organizations—for program management, for development, and for testing. The primary job of developers is to write code and they work under a development manager. The responsibility of the testers is to test the code and they work under a test manager. The program manager provides the specifications for what is being built, and ensure that development and testing are properly coordinated. In a large product this structure may be replicated, one for each major unit. This type of team organization is used in corporations like Microsoft.

4) Software Configuration Management (SCM) plans

The SCM plan, like other plans, has to identify the activities that must be performed, give guidelines for performing the activities, and allocate resources for them. Planning for configuration management involves identifying the configuration items and specifying the procedures to be used for controlling and implementing changes to them. The configuration controller does the CM planning when the project has been initiated and the operating environment and requirements specifications are known. The activities in this stage include:

- Identify configuration items, including customer-supplied and purchased items.
- Define a naming scheme for configuration items.
- Define the directory structure needed for CM.
- Define version management procedures, and methods for tracking changes to configuration items.
- Define access restrictions.
- Define change control procedures.
- Identify and define the responsibility of the CC.
- Identify points at which baselines will be created.
- Define a backup procedure and a reconciliation procedure, if needed.
- Define a release procedure.

The output of this phase is the CM plan.

5) Quality plans

Even though there are different dimensions of quality, in practice, quality management often revolves around defects. Hence, we use "delivered defect density"—the number of defects per unit size in the delivered software—as the definition of quality. This definition is currently the de facto industry standard. By defect we mean something in software that causes the software to behave in a manner that is inconsistent with the requirements or needs of the customer. Defect in software implies that its removal will result in some change being made to the software. To ensure that the final product is of high quality, some quality control (QC) activities must be performed throughout the development. A QC task is one whose main purpose is to identify defects. The purpose of a quality plan in a project is to specify the activities that need to be performed for identifying and removing defects, and the tools and methods that may be used for that purpose. To ensure that the delivered software is of good quality, it is essential to ensure that all work products like the requirements specification, design, and test plan are also of good quality. For this reason, a quality plan should contain quality activities throughout the project. The quality plan specifies the

tasks that need to be undertaken at different times in the project to improve the software quality by removing defects, and how they are to be managed.

Software development is a highly people-oriented activity and hence it is error-prone. Defects can be injected in software at any stage during its evolution. These injection stages are primarily the requirements specification, the high-level design, the detailed design, and coding. For high-quality software, the final product should have as few defects as possible. Hence, for delivery of high-quality software, active removal of defects through the quality control activities is necessary. The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, and acceptance testing (we do not include reviews of plan documents, although such reviews also help in improving quality of the software). With respect to quality control the terms verification and validation are often used. Verification is the process of determining whether or not the products of a given phase of software development fulfill the specifications established during the previous phase. Validation is the process of evaluating software at the end of the software development to ensure compliance with the software requirements. Clearly, for high reliability we need to perform both activities. Together they are often called V&V activities. The major V&V activities for software development are inspection and testing (both static and dynamic). The quality plan identifies the different V&V tasks for the different phases and specifies how these tasks contribute to the project V&V goals. The methods to be used for performing these V&V activities, the responsibilities and milestones for each of these activities, inputs and outputs for each V&V task, and criteria for evaluating the outputs are also specified.

The quality plan for a project is what drives the quality activities in the project. The quality plan specifies the quality control tasks that will be performed in the project. Typically, these will be schedulable tasks in the detailed schedule of the project. For example, it will specify what documents will be inspected, what parts of the code will be inspected, and what levels of testing will be performed. The plan will be considerably enhanced if some sense of defect levels that are expected to be found for the different quality control tasks are mentioned—these can then be used for monitoring the quality as the project proceeds.

6) Risk management

A software project is a complex undertaking. Unforeseen events may have an adverse impact on a projects cost, schedule, or quality. Risk management is an attempt to minimize the chances of failure caused by unplanned events.

The aim of risk management is not to avoid getting into projects that have risks but to minimize the impact of risks in the projects that are undertaken. A risk is a probabilistic event—it may or may not occur. It is defined as an exposure to the chance of injury or loss. In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule. Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal. Risk management can be considered as dealing with the possibility and actual occurrence of those events that are not "regular" or commonly expected, that is, they are probabilistic. It deals with events that are infrequent, somewhat out of the control of the project management, and which can have a major impact on the project. For example, when constructing a building, there is a risk that the building may later collapse due to an earthquake. That is, the possibility of an earthquake is a risk. If the building is a large residential complex, then the potential cost in case the earthquake risk materializes can be enormous. This risk can be reduced by shifting to a zone that is not earthquake prone. Alternatively, if this is not acceptable, then the effects of this risk materializing are minimized by suitably constructing the building (the approach taken in Japan and California). Risk management has to deal with identifying the undesirable events that can occur, the probability of their occurring, and the loss if an undesirable event does occur. Once this is known, strategies can be formulated for either reducing the probability of the risk materializing or reducing the effect of risk materializing. So the risk management revolves around risk assessment and risk control.

Risk assessment is an activity that must be undertaken during project planning. This involves identifying the risks, analyzing them, and prioritizing them on the basis of the analysis. Due to the nature of a software project, uncertainties are highest near the beginning of the project (just as for cost estimation). Due to this, although risk assessment should be done throughout the project, it is most needed in the starting phases of the project. The goal of risk assessment is to prioritize the risks so that attention and resources can be focused on the more risky items. Risk identification is the first step in risk assessment, which identifies all the different risks for a particular project. These risks are project-dependent and identifying them is an exercise in envisioning what can go wrong. Methods that can aid risk identification include checklists of possible risks, surveys, meetings and brainstorming, and reviews of plans, processes, and work products.

In risk analysis, the probability of occurrence of a risk has to be estimated, along with the loss that will occur if the risk does materialize. This is often done through discussion, using experience and understanding of the situation. However, if cost models are used for cost and schedule estimation, then the same models can be used to assess the

cost and schedule risk. Once the probabilities of risks materializing and losses due to materialization of different risks have been analyzed, they can be prioritized.
Once a project manager has identified and prioritized the risks, the top risks can be easily identified. The question then becomes what to do about them. One obvious strategy is risk avoidance, which entails taking actions that will avoid the risk altogether, like the earlier example of shifting the building site to a zone that is not earthquake-prone. For some risks, avoidance might be possible. For most risks, the strategy is to perform the actions that will either reduce the probability of the risk materializing or reduce the loss due to the risk materializing. These are called risk mitigation steps.
Risk prioritization and consequent planning are based on the risk perception at the time the risk analysis is performed. Because risks are probabilistic events that frequently depend on external factors, the risk perception may also change with time. So, in addition to monitoring the progress of the planned risk mitigation steps, a project must periodically revisit the risk perception and modify the risk mitigation plans, if needed. Risk monitoring is the activity of monitoring the status of various risks and their control activities.

7) Project monitoring plans

A project management plan is merely a document that can be used to guide the execution of a project. Even a good plan is useless unless it is properly executed. And execution cannot be properly driven by the plan unless it is monitored carefully and the actual performance is tracked against the plan. Monitoring requires measurements to be made to assess the situation of a project. If measurements are to be taken during project execution, we must plan carefully regarding what to measure, when to measure, and how to measure. Hence, measurement planning is a key element in project planning. In addition, how the measurement data will be analyzed and reported must also be planned in advance to avoid the situation of collecting data but not knowing what to do with it. For monitoring a project schedule, size, effort, and defects are the basic measurements that are needed. Schedule is one of the most important metrics because most projects are driven by schedules and deadlines. Only by monitoring the actual schedule can we properly assess if the project is on time or if there is a delay.
Effort is the main resource consumed in a software project. Consequently, tracking of effort is a key activity during monitoring & is essential for evaluating whether the project is executing within budget. For effort data some type of timesheet system is needed where each person working on the project enters the amount of time spent on the project. For better monitoring, the effort spent on various tasks should be logged separately. Generally effort is recorded through some online system (e.g. weekly activity report system), which allows a person to record the amount of time spent on a particular activity. At any point, total effort on an activity can be aggregated.
As defects have a direct relationship to software quality, tracking of defects is critical for ensuring quality. A large software project may include thousands of defects that are found by different people at different stages. Just to keep track of the defects found and their status, defects must be logged and their closure tracked. Once each defect found is logged (and later closed), analysis can focus on how many defects have been found so far, what percentage of defects are still open and other issues. Defect tracking is considered one of the best practices for managing a project.
Size is another fundamental metric because many data (for example, delivered defect density) are normalized with respect to size. The size of delivered software can be measured in terms of LOC (which can be determined through the use of regular editors and line counters) or function points. At a more gross level, just the number of modules or number of features might suffice.
The main goal of monitoring is for project managers to get visibility into the project execution so that they can determine whether any action needs to be taken to ensure that the project goals are met. Different types of monitoring might be done for a project. The three main levels of monitoring are *activity level, status reporting, and milestone analysis*. Measurements taken on the project are employed for monitoring.
Activity-level monitoring ensures that each activity in the detailed schedule has been done properly and within time. This type of monitoring may be done daily in project team meetings or by the project manager checking the status of all the tasks scheduled to be completed on that day.
Status reports are often prepared weekly to take stock of what has happened and what needs to be done. Status reports typically contain a summary of the activities successfully completed since the last status report, any activities that have been delayed, any issues in the project that need attention, and if everything is in place for the next week.
Milestone analysis is done at each milestone or every few weeks, if milestones are too far apart. Analysis of actual versus estimated for effort and schedule is often included in the milestone analysis. If the deviation is significant, it may imply that the project may run into trouble and might not meet its objectives.
A graphical method of capturing the basic progress of a project as compared to its plans is the cost-schedule-milestone graph. The graph gives the planned schedule and cost of different milestones, along with the actual cost

and schedule of achieving the milestones achieved so far. By having both the planned cost versus milestones and the actual cost versus milestones on the same graph, the progress of the project can be grasped easily.

Software Quality Assurance

Software engineering approach described so far works toward a single goal: to produce high-quality software. Software quality assurance (SQA) is an umbrella activity that is applied throughout the software process. SQA encompasses:
(1) a quality management approach,
(2) effective software engineering technology (methods and tools),
(3) formal technical reviews that are applied throughout the software process,
(4) a multi-tiered testing strategy,
(5) control of software documentation and the changes made to it,
(6) a procedure to ensure compliance with software development standards (when applicable), and
(7) measurement and reporting mechanisms.

All engineered and manufactured parts exhibit variation. Variation control is the heart of quality control. How might a software development organization need to control variation? From one project to another, we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment, and calendar time. In general, we would like to make sure our testing program covers a known percentage of the software, from one release to another. Not only do we want to minimize the number of defects that are released to the field, we'd like to ensure that the variance in the number of bugs is also minimized from one release to another. (Customers are likely to be upset if the third release of a product has ten times as many defects as the previous release.) We would like to minimize the differences in speed and accuracy of our hotline support responses to customer problems & so on.
When we examine an item based on its measurable characteristics, two kinds of quality may be encountered: *quality of design* and *quality of conformance*. Quality of design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications. Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again, the greater the degree of conformance, the higher is the level of quality of conformance. In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.
Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. This approach views quality control as part of the manufacturing process. Quality control activities may be fully automated, entirely manual, or a combination of automated tools and human interaction. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.
Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

Cost of quality includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a baseline for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms. *Quality costs* may be divided into costs associated with prevention, appraisal, and failure. Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of *appraisal costs* include

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are incurred when we detect a defect in our product prior to shipment. *Internal failure costs* include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are:

- complaint resolution
- product return and replacement
- help line support
- warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? For our purposes, software quality is defined as:
"*Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software*."
The definition serves to emphasize three important points:

1) Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2) Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3) A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group. The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. Software engineers address quality (and perform quality assurance and quality control activities) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing. The activities performed (or facilitated) by an independent SQA group are:

a) Prepares an SQA plan for a project: The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies:
   - evaluations to be performed
   - audits and reviews to be performed

- standards that are applicable to the project
- procedures for error reporting and tracking
- documents to be produced by the SQA group
- amount of feedback provided to the software project team

b) Participates in the development of the project's software process description- The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

c) Reviews software engineering activities to verify compliance with the defined software process- The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

d) Audits designated software work products to verify compliance with those defined as part of the software process- The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

e) Ensures that deviations in software work and work products are documented and handled according to a documented procedure- Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

f) Records any noncompliance and reports to senior management- Noncompliance items are tracked until they are resolved. In addition to these activities, the SQA group coordinates the control and management of change and helps to collect and analyze software metrics.

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered after the software has been released to end-users (or to another activity in the software process). The term *error* is used to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end-user (or to another activity in the software process). The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process. A number of studies reveal that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases. Defects may be amplified during the preliminary design, detail design, and coding steps of the software engineering process. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified by current work.

Formal Technical Reviews

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen. The FTR is actually a class of reviews that includes walkthroughs, inspections, round-robin reviews and other small group technical assessments of software. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing focus, the FTR has a higher likelihood of uncovering errors. The focus of the FTR is on a work product (e.g., a portion of a requirements specification, a detailed component design, a source code listing for a component). The individual who has developed the work product—the producer—informs the project leader that the work product is complete and that a review is required. The project leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day. The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the recorder; that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each. At the end of the review, all attendees of the FTR must decide whether to (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A review summary report answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties. The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report. Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

- incomplete or erroneous specifications (IES)
- misinterpretation of customer communication (MCC)
- intentional deviation from specifications (IDS)
- violation of programming standards (VPS)
- error in data representation (EDR)
- inconsistent component interface (ICI)
- error in design logic (EDL)
- incomplete or erroneous testing (IET)
- inaccurate or incomplete documentation (IID)
- error in programming language translation of design (PLT)
- ambiguous or inconsistent human/computer interface (HCI)
- miscellaneous (MIS)

Software Reliability

Reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable. Software reliability, unlike many other quality factors, can be measured directed and estimated using historical and developmental data. Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". Whenever software reliability is discussed, a pivotal question arises: What is meant by the term failure? In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures. If we consider a computer-based system, a simple measure of reliability is meantime-between-failure (MTBF), where:

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively. Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP. Stated simply, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system. In addition to a reliability measure, we must develop a measure of availability. Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

Availability = [MTTF/(MTTF + MTTR)] X 100%

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

SQA Plan

The SQA Plan provides a road map for instituting software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities that are instituted for each software project. A standard for SQA plans has been recommended by the IEEE. Initial sections describe the purpose and scope of the document and indicate those software process activities that are covered by quality assurance. All documents noted in the SQA Plan are listed and all applicable standards are noted. The **management section** of the plan describes SQA's place in the organizational structure, SQA tasks and activities and their placement throughout the software process, and the organizational roles and responsibilities relative to product quality. The **documentation section** describes (by reference) each of the work products produced as part of the software process. These include

- project documents (e.g., project plan)
- models (e.g., ERDs, class hierarchies)
- technical documents (e.g., specifications, test plans)
- user documents (e.g., help files)

In addition, this section defines the minimum set of work products that are acceptable to achieve high quality. The **standards, practices, and conventions section** lists all applicable standards and practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines). In addition, all project, process, and (in some instances) product metrics that are to be collected as part of software engineering work are listed. The **reviews and audits section** of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group, and the customer. It provides an overview of the approach for each review and audit. The **test section** references the Software Test Plan and Procedure. It also defines test record-keeping requirements. **Problem reporting and corrective action** defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities. The remainder of the SQA Plan identifies the tools and methods that support SQA activities and tasks; references software configuration management procedures for controlling change; defines a contract management approach; establishes methods for assembling, safeguarding, and maintaining all records; identifies training required to meet the needs of the plan; and defines methods for identifying, assessing, monitoring, and controlling risk.

Software Configuration Management

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. 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. 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.

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 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. 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.

Baselines

A baseline is a software configuration management concept that helps us to control change without seriously impeding justifiable change. A baseline can be defined 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*". Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change. 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. 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. 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 are followed.

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 tests. More realistically, an SCI is a document, a entire suite of test cases, or a named program component. 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. 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. The relationship may be compositional. That is, data model and component N are part of the object Design Specification. 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.

The SCM Process

The change management process has the following steps.

- Log the changes
- Perform impact analysis on the work products
- Estimate impact on effort and schedule
- Review impact with concerned stakeholders
- Rework work products

A change is initiated by a change request. A change request log is maintained to keep track of the change requests. Each entry in the log contains a change request number, a brief description of the change, the effect of the change, the status of the change request, and key dates.

Thus, 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 apprise 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.

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: 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, 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. 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:

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

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. 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.

Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. Each version of the software is a collection of SCIs (source code, documents, data), and each version may be composed of different variants. 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.

Change Control

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. 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. 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 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, performed by two different people, don't overwrite one another. 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, 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. 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. 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.

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.

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) software configuration audit.

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.

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.

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?

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 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. Configuration status reporting plays a vital role in the success of a large software development project. 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.

[E.g. of software configuration management and version control tool: Concurrent Versions System (CVS)]
http://savannah.nongnu.org/projects/cvs
