Module 5: Software Engineering Project Management

Topics

---

1. Software Project Management Activities

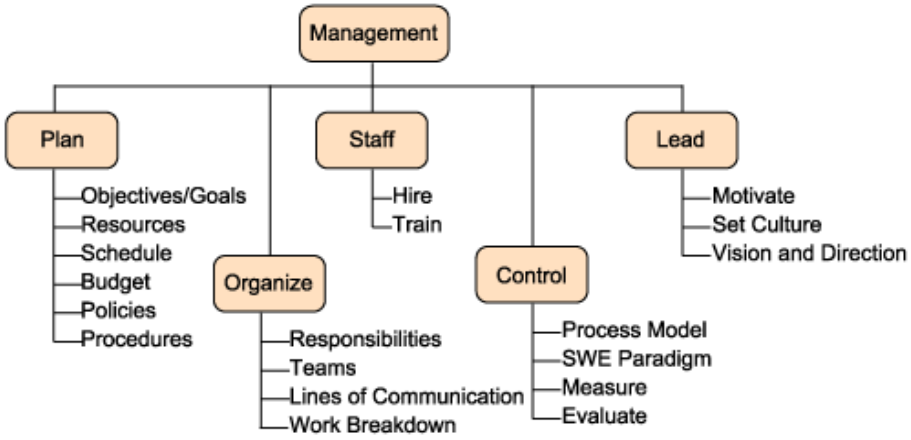
Software project management is an umbrella activity within software engineering. The software engineer plays an active role in developing the overall plan to include the activities necessary to manage the major factors that influence the software project. Pressman (2005) identifies the "four Ps" of project management as:

- people
- product
- process
- project

A project plan to manage the four Ps is compiled into a formal document and is called the *Software Project Plan* (*SPP*). Supplementary documents can also be produced and referenced by the SPP. For example, projects that contain safety-critical software require an additional document, the Software Safety Plan, to be created to specifically address the safety program-management component of the overall project-management activities. Additionally, a risk management plan can be developed to focus on the inherent risks involved in building a complex system. For larger projects, a software configuration plan can be developed to manage change. We will focus our discussion on the Software Project Plan in the next two sections of this module. In section 3, we will discuss risk management; in section 4, we will review the activities of software configuration management.

To introduce you to the activities involved in software project management, let's first examine the classic management model as portrayed by Koontz and O'Donnell (1972). The management model in figure 5.1 specifies five major management functions that address the "four Ps" of project management.

Figure 5.1
Software Project Management Activities



The expanded tree for each major management function in figure 5.1 contains the activities to be performed to accomplish the specific function. Project management is defined as the application of a systematic approach to implement all the functions necessary to manage a project.

Planning a Software Project

Software project planning commences after management and stakeholders determine that the software project is economically attractive. The product of this planning task is the Software Project Plan. Note that the SPP is not the

documentation tool for feasibility analyses, trade-off studies, cost-benefit analyses, or budgeting/cash flow projections. All of these critical economic analyses should be performed before the project plan is developed.

The project-planning process begins with making reasonable estimates of resources, costs, and schedule that are required to deliver a quality software product. To direct and implement the overall project plan, it is necessary to establish a set of policies and procedures.

Correct software project planning requires sound technical estimates, so that the software engineer will either be responsible for the SPP or will provide the technical input for management to develop the SPP. Either way, the software engineer is integral to planning. In figure 5.1 we have listed various categories of planning; we will discuss each in the following sections.

Analysis of the Problem

Although we discussed analysis of the problem in module 2, we would like to emphasize that the first step of software project planning is system analysis. You can't plan to develop anything until you know, reasonably well, how big and how complex it is. Therefore, you must first bound the problem you are attempting to solve; you accomplish the bounding with system analysis. Everything that lies within the boundary is part of your overall project. Everything outside of the boundary is unchangeable and not part of the project; it is part of the software product's environment.

Software Project Estimation

Once the problem has been bound, the software engineer must size the proposed software product in order to estimate the human effort required and the amount of time it will take to complete it. The size of the software product can be estimated using the common conventional methods for computation:

- lines of code (LOC)
- function points (FP)

The **LOC software sizing method** is dependent upon historical data. Prior projects of similar complexity can be compared with the proposed software product and scaled up or down, depending on such factors as the programming language to be used for development and the number of functions to be implemented in the new product versus the historical product.

The programming language affects product size. Higher-order programming languages such as Java and C++ can implement equivalent logic in fewer program statements than lower-order programming languages such as Assembly. The LOC method can result in a very good estimate of proposed software product size if care is taken to document all assumptions and use realistic scaling factors.

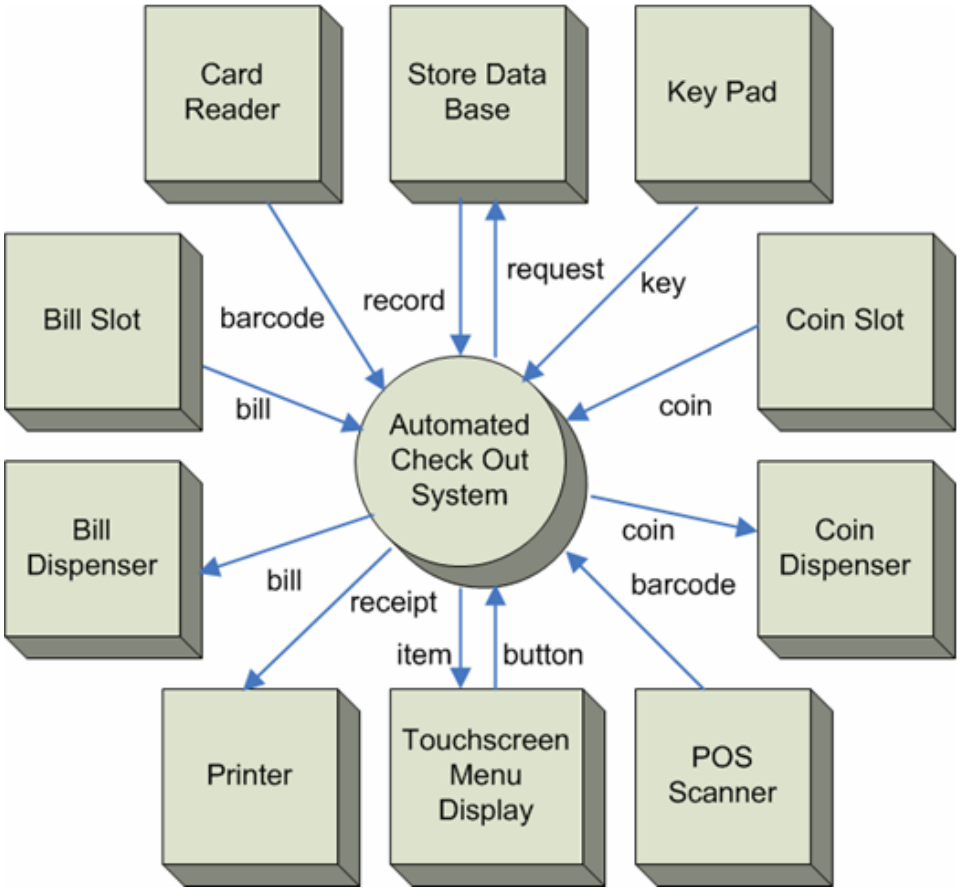
The FP software sizing method does not depend upon historical data but can be refined when historical data is available. In FP, the requirements of the proposed software product are all that is needed to estimate the software product's size. An FP value is computed that measures the complexity of the proposed software product by evaluating its requirements. There are four processing steps in computing FPs, as outlined by David Longstreet (2002) in his paper, *Fundamentals of Function Point Analysis*:

- analyze the application's domain to determine external counts
- assign a weight to each count according to the level of complexity
- grade the influence of the general characteristics of the system that will affect the execution of the application
- compute the function point

Example of Function Point Analysis

Review the architectural context diagram in figure 5.2 for the self-service checkout system and refer back to it for the following function-based metrics.

Figure 5.2
Context Diagram for Self-service Checkout System



Step 1: Analyze the application's domain to determine external counts

In the first step, we determine the *external counts* from the number of inputs, outputs, inquiries, and internal logical files and the interface files. We will identify each type of the external counts separately.

The external inputs are:

1. bill
2. coin
3. key
4. barcode

The external outputs are:

1. receipt
2. bill
3. coin

The external inquiries are:

1. button
2. request

The internal logical files are:

1. system configuration file

The external interface file can be defined as:

1. store's database

Step 2: Assign a weight to each information domain value according to the level of complexity

The second step of function-based metrics is to establish criteria for determining the *level of complexity* for each domain value. The intended software functionality and the simplicity of the entry are used as the basis for

developing the criteria. When available, historical data from similar applications can be used to assist in determining the criteria for establishing the complexity levels.

For our proposed self-service checkout system, we will assume the weighting factors as indicated in table 5.1, which are based on a scale of 1 to 10 and are grouped into three categories: less complex, average, and more complex.

Table 5.1
Weighting Factors Based on Level of Complexity

Domain Value	Less Complex	Average	More Complex
External Inputs	3	5	9
External Outputs	2	4	7
External Inquiries	3	5	8
Internal Logical Files	4	7	10
External Interface Files	3	6	8

Now we populate a table with the count values as predetermined in step 1 and multiply them by the weighting factors that we assigned in table 5.1. The count total or unadjusted function point value for our self-service checkout system is computed and shown in table 5.2.

Table 5.2
Unadjusted Function Point Computation

Information Domain Value	Count	Complexity Weighting Factor	Total
External Inputs	4	3	12
External Outputs	3	2	6
External Inquiries	2	3	6
Internal Logical Files	1	4	4
External Interface Files	1	3	3
Unadjusted function point			**31**

Step 3: Grade the influence of general characteristics of the system that will affect the application

The third step of function-based metrics is to determine the *value adjustment factor* (VAF) that is based on the general characteristics of the system that is most likely to influence the application. There are 14 possible system characteristics that need to be considered (Longstreet, 2002) in determining the VAF for a software project. You can review possible system characteristics in table 5.3.

Table 5.3
Summary of General Characteristics of a System

Number	System Characteristic	Description
1	Data communications	Number of communications facilities to transfer or exchange information
2	Distributed data processing	Handling of the distributed data and processing
3	Performance	Required response time
4	Hardware configuration	Usage of the current hardware configuration
5	Transaction rate	Frequency of transactions
6	Online data entry	Amount of data entered online
7	End-user efficiency	Amount of user interaction

8	Online update	Amount of internal logical files updated online
9	Complex processing	Amount of mathematical processing
10	Reusability	Number of intended software reuse to satisfy requirements
11	Installation	Ease of installation
12	Operations	Effectiveness of start-up, backup, and recovery procedures
13	Multiple sites	Number of sites
14	Facilitate change	Ease of change

The characteristics listed in table 5.3 will not apply to *every* system. We determine the characteristics applicable to the system that we are proposing to develop, assign an influence rating to each characteristic, and determine the value adjustment factor (Longstreet, 2002). The rating range assigned to each characteristic reflects the influence on the software development life cycle and is based on a scale from 0 (no influence) to 5 (strong influence).

The function point equation is:

$FP = \text{unadjusted FP} * [[0.65 + 0.01]] * \Sigma F_i]$

Each system characteristic is represented by the F_i, where *i* represents one particular characteristic, and the rating assigned to the characteristic. All the ratings assigned to the system's characteristics are summed together to determine the VAF, which is used in the equation to determine the FP. We will discuss the FP equation further in the step 4 discussion.

For our self-service checkout system, we will assign a rating to each system characteristic. We sum together each rating assigned for each system characteristic to determine a total grade rating or VAF. We illustrate this summation in table 5.4.

Table 5.4
Compute a Total Grade Rating for System Characteristics

Number	**System Characteristic**	**Grade Rating**
1	Data communications	2
2	Distributed data processing	3
3	Performance	4
4	Hardware configuration	4
5	Transaction rate	4
6	Online data entry	4
7	End-user efficiency	5
8	Online update	3
9	Complex processing	3
10	Reusability	1
11	Installation	2
12	Operations	3
13	Multiple sites	5
14	Facilitate change	2
Total		**45**

Step 4: Compute the function point

The final step in function-based metrics is to populate and to calculate the function point using the predetermined unadjusted function point, and the ratings assigned to the system characteristics.

The following relationship is used to determine the function point (FP):

FP = unadjusted FP $*$[[0.65 + 0.01]] $*$ Σ F*i*]

where:

- *unadjusted* FP represents the complexity level for the information domain values
- Σ F_i represents the VAF or total degree of influence of all characteristics of the system

A constant complexity multiplier is applied to the total degree of influence to determine the complexity level of the software:

[[0.65 + 0.01]] $*$ Σ*Fi*]

We can now populate the equation and perform the calculation of the function point for the software required for our self-service checkout system.

FP = 31 $*$[[0.65 + 0.01]] $*$ 45]

FP = 920.7

The FP value is the software product's size. But unlike the value resulting from an LOC calculation, the value resulting from an FP calculation has no units; it is "dimensionless." Therefore, if reliable historical data is available, it is useful at this point. By computing the FP value of past projects and comparing it to the FP value of the proposed product, the software engineer has a comparative measure of the proposed product's size.

When historical data is unavailable or unreliable, FP is often converted into LOC according to conversion tables built using historical data from other development organizations. This conversion somewhat defeats the advantages of FP because FP is programming-language independent, whereas LOC is not. FP measures intrinsic product complexity using requirements. LOC effectively measures the efficiency of programmers translating requirements into code.

Human Effort Estimation

Once the size of the proposed software product is known, the software engineer must estimate the human effort and the time to complete the project, i.e., the project duration. The de facto standard for measurement units of effort is staff hours, staff days, or staff months. There are two methods commonly used in the industry: conventional and empirical.

The **conventional method** of estimating effort and duration is based on historical data. Using similar previous projects, the software engineer can scale up or down the estimates of effort and duration for the proposed project. Historical information about the product's effort and duration in a previous project is used in much the same way as historical information was used to estimate product size in the earlier discussion.

The **empirical method** of estimating effort and duration is discussed in the following subsection.

To compute staff size, following either the conventional or empirical method, simply divide project effort by the project duration, which will yield the staff size. For example,

Step 1: Assign values for duration and effort:

project duration = 24 months

project effort = 3,000 staff months

Step 2: Calculate the project size:

size = effort / duration

3,000 / 24

staff size = 125

Empirical Estimation Models

Empirical estimation models are the alternative methods for estimating project effort and duration based on software product size. Empirical estimation models do not estimate product size; the software engineer is responsible for this estimate.

Historical data of past software projects is critical to the accuracy of any empirical estimation model. Regression analysis and curve fitting are mathematical techniques applied to historical data to transform the data into an equation or set of equations that models the behavior of the data. The mathematical equation or set of equations is the empirical estimation model. Once validated as accurate within accepted tolerances, the empirical estimation model may be used for any future software product.

To better accommodate the diversity of software products and the environment within which they are developed, modern empirical estimation models can be adjusted with parameters that are coefficients, exponents, scaling factors, or some combination of these. It is the job of the software engineer to select the best empirical estimation model and the set of adjustment parameters to estimate the effort and project duration of the software product in question. In practice, several models may be employed to factor out any one model's bias.

COnstructive COst Model (COCOMO)

The COnstructive COst Model, referred to as COCOMO, is an empirical estimation model introduced to software engineering by Barry Boehm (1981). COCOMO is one of the most commonly used and best-known effort-schedule-cost estimation models in the industry (Pressman, 2005).

The mathematical equations used in COCOMO were developed from an analysis of 63 software projects of various types dating back to the 1970s. COCOMO assumes that there are 19 productive staff days per month and 152 staff hours per staff month. These numbers are based on the average number of days (excluding vacation and holidays) and hours worked per day in the United States (Futrell et al, 2002). In order to accommodate differing development environments and complexity levels of software, Boehm defines three modes: organic, semidetached, and embedded. Table 5.5 summarizes the characteristics of these three modes:

Table 5.5
COCOMO Modes

COCOMO Mode	Project Type	Team Size	Development Environment	Constraints and Deadlines
organic	payroll, inventory	small	stable (in-house)	loose
semidetached	compilers, database, editors	medium	fluid	moderate
embedded	real-time	large	complex	severe

The basic human effort formula used in COCOMO is:

Effort (E) = a x (Size)^{b}

where

- *a* and *b* are constants derived from regression analysis that is dependent on the project type
- *Size* represents thousands of lines of code (KLOC)
- *Effort* is expressed in staff months

To accommodate the three modes defined in COCOMO, the following formulae have been defined and are used to estimate human effort:

Mode	**Formula**
Organic	E = 2.4 x **(Size)$^{1.05}$**
Semidetached	E = 3.0 x **(Size)$^{1.12}$**
Embedded	E = 3.6 x **(Size)$^{1.20}$**

The basic formula used in COCOMO to estimate project development time in staff months is:

TDEV = a x (E)^{b} months

To accommodate the three modes defined in COCOMO, the following formulae have been defined and are used to estimate project development time:

Mode	Formula
Organic	TDEV = 2.5 x $(E)^{0.38}$
Semidetached	TDEV = 2.5 x $(E)^{0.35}$
Embedded	TDEV = 2.5 x $(E)^{0.32}$

To calculate the average staff size (SS), we can use the effort and development time as follows:

Average Staff Size (SS) = Effort / TDEV

Once the average staff size is calculated, we can estimate the productivity level as follows:

Productivity (P) = Size / Effort

Using the COCOMO estimation model in software engineering has several advantages in project management. COCOMO can be easily implemented in software process improvement because it is a repeatable process that can be used for all projects. The formulae used in COCOMO were developed based on previous projects' experiences and are well documented. Another advantage to using COCOMO in project cost and schedule estimation is its ability to support different modes, as was previously defined in table 5.5.

Over the decades, COCOMO was recognized by software engineering practitioners to be deficient because it does not consider the software requirements and could not be adapted to all development models. As a result of these deficiencies, a revision of COCOMO has evolved and is referred to as COCOMO II. COCOMO II considers cost and schedule estimations for object-oriented software, software developed following the spiral and evolutionary models and applications developed with commercial off-the-shelf software. COCOMO II provides a set of tools and techniques to account for the effects of technology improvements on the development cycle and the costs of software (Futrell et al, 2002). Effort is calculated using object points and the ranges of estimates represent the standard deviation surrounding the most likely estimation. This improvement to the original COCOMO also adjusts for reuse of software and the use of re-engineering tools. You are encouraged to search the Internet for CASE tools that support COCOMO II.

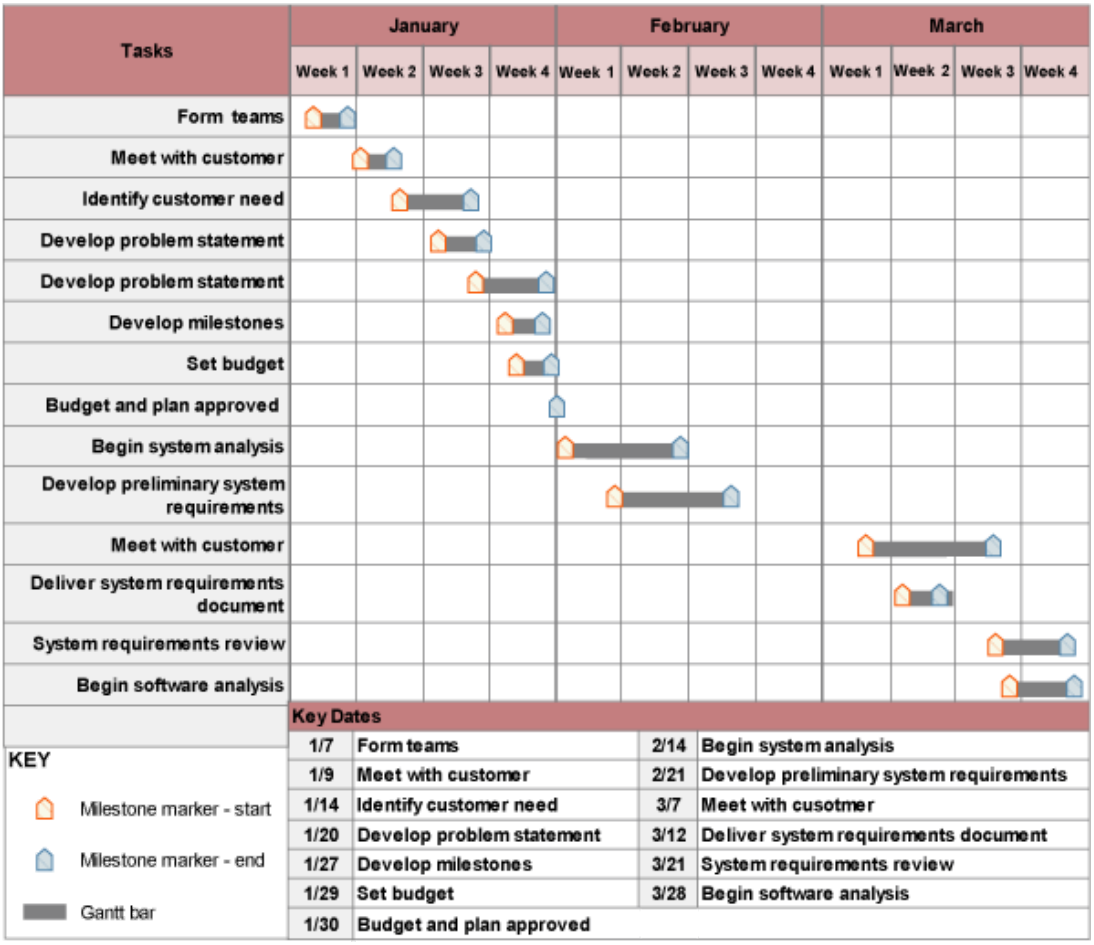
Work Breakdown and Scheduling a Software Project

The work of software development is broken down into tasks. A **task** should be limited to the amount of work that can be assigned to one software engineer or to a team of engineers to produce a product or a component of a product. The product can be any of those associated with the software development process, e.g., technical document, source code, test case, and so forth.

Based on the previous definition of *task*, the software development tasks of analysis, design, code, and test are each too large. Therefore, they should be decomposed or broken down into smaller tasks. The result of this activity is the **task breakdown** for the project. Once all the work of software development has been broken down, the tasks can be assigned to individuals or teams and allocated resources, scheduled for execution, assigned a budget, and so on. A task breakdown can be formulated into a **task network** simply by interconnecting tasks based on actual task interdependencies when they exist.

A **project schedule** is, essentially, a list of tasks with the estimated time each task should take to be performed. A popular and effective representation of a project schedule is a **timeline chart** (also referred to as a Gantt chart or Waterfall chart). Refer to figure 5.3 for an example Gantt chart timeline.

Figure 5.3
Example of Project Schedule



In figure 5.3 we can see the graphical depiction of tasks as bars that are plotted along an axis representing calendar weeks. The relationship of one task bar to another on the time axis represents the sequence in which tasks are to be executed. Tasks that are executed at the same time are aligned side by side. Tasks that occur in sequence are represented as head-to-tail.

The other axis of a timeline chart is the task axis. The task axis represents the task breakdown discussed earlier, i.e., all of the tasks of software development from analysis through test. Each task must have a timeline bar associated with it to represent its start time and its duration. Time dependencies between tasks may be represented graphically as connectors that link tasks, indicating that a change in one task's start time or duration affects the execution of the other.

Many commercially available software packages automate the development and depiction of a timeline chart. Resources, uncertainty, milestones, and products are a few of the elements that can be associated with each task in a project schedule, resulting in a comprehensive view of the project in a concise diagram.

Staffing and Organizing a Software Project

People are a major resource to a successful software project. Project managers are responsible for ensuring that adequate technical expertise exists within the organization to accomplish the job. Staffing and training technical people to develop the skill set necessary for building the software product are major activities within the project. The project manager is responsible for making sure that adequate staff exists to perform the required work.

Once the project manager selects the appropriate technical staff, strong teams can be formed to perform the outlined tasks. Effective information flow concerning the development of the project must be established between all members of the team, and with other teams and management. Lines of communication are also established between the customer and the technical staff to promote feedback of information. One integral part of the project is gathering and communicating the requirements for the desired software product to lay the foundation for the development of the correct product.

The overall project framework must be organized and managed to complete a successful project. The project framework supports the open lines of communication between all stakeholders and all disciplines involved in the project. Communication is necessary to relay information and for managing the budget and schedule, and for tracking and allocating resources within the project's framework.

Controlling and Leading the Software Project

To guide the development activities, the project manager must select a process model that is applicable to the specific type of software project. Once this process model is selected, the project manager and the development team adopt and apply the model to the organization's internal operations. We introduced process improvement models and development approaches earlier in module 1. In the next section, we will extend our discussion on the commitment to process improvement from a project management perspective.

Process Improvement Commitment

Successful process improvement requires commitment from all levels of management and technical employees within an organization. The decision to commit to a process improvement model is made by management, who must be well informed by the person making the proposal. Management must review the projected investment costs, time commitment, proposed benefits, and the return on investment that could result from making this business decision. One major disadvantage of adopting a process improvement model is that the return on investment is not seen immediately—it may take years. Managers must be made aware of this fact before they commit to a process model.

Management involvement plays an important role in the success of the organization reaching a higher maturity process level. Management must be receptive to criticism of its current processes and be willing to implement new practices. All levels of managers must be trained in the concepts of process improvement and "buy in" to these concepts for it to be successfully implemented within the organization.

Managers' roles may be affected by the adoption of new standards and practices, and as a result, new job positions and responsibilities may be created in the management structure. There is a risk that the managers of the organization will not accept the new processes.

It is beneficial to management for technical employees in all levels of the organization to be involved in the decision-making process to adopt a process improvement. Without the "buy-in" of these employees, the process improvement effort will likely be unsuccessful. The employees who actually perform the work are instrumental in defining the software processes to meet their development needs.

To summarize our discussion of process improvement, it is the process that builds the product. If the process is deficient, it will affect the quality of the end product. It is important to recognize that all the activities discussed in this section are involved in defining an effective software process. A deficiency in any activity can affect the quality of the final product.

To help you understand what is involved in software process improvement, we will discuss the Software Engineering Institute's Capability Maturity Model (SW-CMM) in the next section.

History of the Software Capability Maturity Model

The goal of the Software Engineering Institute's SW-CMM is to help organizations deliver usable high-quality software products to their customers in a timely manner and within budget. Over the years, these concerns have become more important as the demand for more reliable and cost-effective software has increased.

The SW-CMM attempts to improve the software process and its management by requiring that standards be developed and applied consistently throughout the entire organization.

Evolutionary Path to Software Process Maturity

SW-CMM offers an evolutionary improvement path for an organization from an ad hoc, immature process to a final stage of establishing mature, disciplined software processes. This path consists of five levels of process maturity. These levels are referred to as:

- Level 1 - Initial
- Level 2 - Repeatable
- Level 3 - Defined
- Level 4 - Managed
- Level 5 - Optimizing

In SW-CMM's levels two through five, there exist several key process areas that must be addressed by the organization so that it can advance to the next maturity level. These process areas are organized into sections called *common features* and are described as follows: "Common features specify the key practices which together accomplish the goals of the key process area" (Paulk et al., 1993, p. O-9).

The five maturity levels defined by the SEI's SW-CMM are graphically represented in figure 5.4. Click on the level's label to view the key process activities (Paulk et al., 1993) designated for that particular SW-CMM level of maturity.

Figure 5.4
Maturity Levels and Key Process Areas of the SW-CMM

Software organizations benefit by trying to achieve higher levels of maturity following the SW-CMM framework. The higher level of maturity places the organization in a more competitive place in the market. Government and industry also benefit from the organization's effort by being assured of higher degrees of product reliability from their software contractors.

Capability Maturity Model Integrated

In December 2001, the Software Engineering Institute released the Capability Maturity Model Integrated (CMMI). The CMMI is an upgrade from the CMM and can easily be implemented by organizations because it builds on their knowledge attained from following the original CMM (Carnegie Mellon, 2003).

The CMMI was developed to better integrate the software and system engineering disciplines. The driving force behind the CMMI was to address the dramatic increase in the importance of software as part of a system (Phillips, 2003). The transition to CMMI allows the software organization to move from the attained CMM maturity level to the corresponding maturity level in the CMMI.

It should be noted that software is just one component of a complete system. The CMMI addresses the processes of software and system engineering to merge both into one process improvement model. The success of CMMI requires a greater level of two-way communications and the cooperation of both disciplines.

The upgrade to CMMI included several improvements, as follows:

- new key process areas
- best practices noted over the years
- generic goal for each process area

One of the notable improvements seen within the CMMI framework is in the key process area for risk management, which has been added to the original CMM's defined maturity level. In the CMMI, risk management has been formalized as its own process. This feature makes the CMMI attractive for organizations that are producing the safety-critical system.

Since the introduction of the CMMI in 2001, organizations have prepared reports on the many benefits they have experienced from the upgrade. These reports are presented at annual CMMI technology conferences. To learn more about the Capability Maturity Model Integrated, you can perform a search on the Internet and visit the Carnegie Mellon Software Engineering Institute's web site.

The CMMI model has gained popularity and is being followed by many organizations in the software industry today. The benefits achieved from adopting a process improvement model are continuously being studied by organizations that have implemented the CMMI process model.

2. Software Quality Assurance

Although software quality is an abstract concept, it has concrete manifestations. The specific attributes of software that make it of "high quality" sometimes seem a bit difficult to pin down, but it is easy to tell when we are using "low quality" software: it is slow and hard to use, it doesn't operate as its manual says that it should, it freezes up on us and we lose work, and so on.

The problem is that during development, we can't actually see how the software works because it hasn't yet been built. So we have to rely on something else to help us develop a reliable product. **Software quality assurance** (SQA) indicates whether the software under construction is being built properly. As a part of SQA, we monitor two aspects of the software development:

- the **process** (the way that the developers are creating the software)
- the **product** (the actual software)

You may think that the process is the less important of these two. After all, what difference does it make *how* software is built as long as it is built well? But experience over the years of software development has clearly demonstrated that unless a sound and careful process is followed, the product will likely be of low quality.

Of course, there are limits to the amount of quality that can be built into a product. These limits arise out of having finite resources (budget and time available for development) as well as the finite capabilities of the developers themselves. With the complexity of modern systems, people often face amazingly complicated software challenges. It is only with a careful and disciplined process that we can succeed in producing high-quality systems. It is the role of software quality assurance organizations and activities to make sure that such processes are defined and followed.

Software Reviews

A **software review** (or technical review) is a meeting in which the participants present and discuss issues relating to the software under development. Reviews are a fundamental tool not only for software developers but also for quality assurance staff. Why are reviews so valuable? The answer is that they facilitate the free exchange of ideas and viewpoints, and allow many minds to tackle the problems at hand. As Pressman (2001) points out, reviews are "filters," that when applied at various times during development, help the development team to uncover problems and devise solutions. They "filter" out the problems and help to "purify" the products.

Many of you have probably participated in reviews at one time or another and have formed some opinions about their effectiveness. A review has to be very carefully planned and conducted to realize the maximum benefit. Yet, even with careful planning, the rewards may be limited. For example, a review cannot easily explore very detailed and complicated issues, because having a group of people in a review tends to limit the amount of detail that can be discussed.

Reviews, however, do provide some benefits that cannot be achieved through other means. By having many minds focusing on the same problem, fresh approaches to problems can be created, and in a group, dialog can help to clarify or determine important decisions. Overall, the main goal of a review is to find and remove software defects that, if left undiscovered, can be expensive to find and fix later in the life cycle of the project.

There are two main types of software reviews: formal and informal. **Informal reviews** provide an important service to any project and are based on discussions of technical issues, with a loose format and protocol. **Formal reviews**, in contrast, are strictly organized and conducted, with very specific goals.

Measures of Software

The idea of *measuring* software may seem a bit strange at first. You can't weigh software and you can't see how long it is in feet. As we have seen in the previous section of this module, there are many characteristics of software that can be measured. For example, you can count the number of source-code lines, the function points, and the number of defects that have been found. There are many additional valuable and important

measurements you can make on the software you're developing. We can measure the cyclomatic complexity and the cohesion. More importantly, these measurements can help you manage and improve the software as you proceed through development.

A critical concept that you need to understand is the difference between direct measures and indirect measures (Pressman, 2001). Direct measures have a well-understood meaning and relevance. When you apply a **direct measure**, you are directly counting a specific attribute. For example, you can tell if the software executes within a certain time period or if the software occupies a certain amount of memory.

Indirect measures, however, have a more abstract relationship to the characteristic that you are trying to measure. When you apply a quality measure, for example, you are not directly measuring quality. Instead you are measuring certain attributes that you believe *imply quality or are related to* quality. Although you can't prove it, heuristically you have confidence in what the measure is telling you. During development, both types of measures are important.

One important use of direct and indirect measures is to incorporate them as a part of what Pressman (2005) calls "statistical software quality assurance." With this idea, developers apply various measures throughout the development and collect and use the results as feedback to improve the processes. Over time, the development organization will show improvement in its ability to produce high-quality software on time and within budget. Factories that produce hardware products use this same idea to improve their production lines.

Many different measures have been defined for use in software development. In fact, there are so many that it is often hard to select which ones to use. By and large, when selecting measures, you should keep in mind the basic goals and requirements of the software itself. If the product is to be highly reliable, for example, then a set of reliability measures should be applied. If reliability is not a prime goal, then use of these measures should be minimized.

Pressman (2001) introduced a set of measures for each of the major development phases, from specifications through testing and maintenance. When planning a project, consider which measures you should use across the life cycle. The descriptions provided by Pressman are a good starting point for making your selections. Table 5.6 lists the sample measures that are useful in each phase of the life cycle.

Table 5.6
Sample Measures Across the Life Cycle

Phase	Sample Measure	Purpose
Requirements Analysis	Number of requirements	Size of system
Top-level Design	Number of top-level components	Size of system architecture
Detailed Design	Number of components with complete detailed design	Design progress
Code and Unit Test	Source lines of code written	Size of each code unit
Integration and Test	Number of units completed test	Completion progress for unit testing
System Integration and Test	Number of failed system component tests	Quality of components
Qualification Test	Number of requirements passed testing	Acceptability of final product

Software Quality Assurance Plan

Software quality assurance is so important to software development that a supplementary document to the SPP is created to address software quality measures to be used during the development of the software product. This document is referred to as the Software Quality Assurance (SQA) Plan and the IEEE provides an outline document, *IEEE Standard for Software Quality Assurance Plans, Std 730-1998*. The IEEE provides standards to identify the scope of software quality, the boundary of the product to be measured, and any tools to be used for measurement, roles of responsible parties, and auditing policy and procedure.

3. Risk Assessment and Management

Managing risks in a software development project requires a disciplined environment for learning and decision making. Different types of risk can exist in a software project, technical risks and project risks. These types of risks require a special focus and must be continually monitored for anything that could potentially go wrong.

Risk management is a continuous project management function. New risks can occur at any time during the software development cycle.

Risk management is defined by the Carnegie Mellon Software Engineering Institute (SEI) as a "software engineering practice with processes, methods, and tools for managing risks in a project" (2003c).

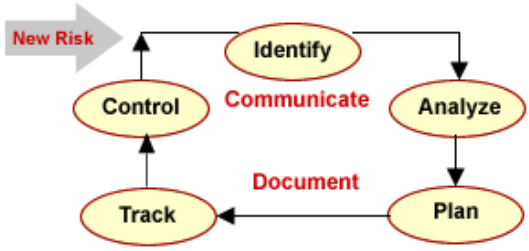
If we read the following definition provided by Van Scoy (1992, p. 3), we can see that risk can also be viewed as an opportunity:

Risk in itself is not bad; risk is essential to progress, and failure is often a key part of learning. But we must learn to balance the possible negative consequences of risk against the potential benefits of its associated opportunity.

The Software Engineering Institute defines a risk management model that defines a basic set of functions that are identified as continuous activities throughout the development life cycle of a software project. The SEI describes the communication function as an integral part of this model. The SEI's risk management model has been modified by a working group at NASA to include an additional function: document. Document has been added to the model to reflect the importance of tracking information that results from each function performed on each risk.

Move your mouse over each of the seven labels in figure 5.5 for each phase in the risk management model to read a description of the activity that should occur.

Figure 5.5
Risk Management Model



Risk Assessment

The only things certain in life are—you guessed it—death and taxes. Anything else may or may not happen, i.e., will be uncertain, a risk. We can measure uncertainty with a probability estimate. A probability estimate has a value between zero and one (0 and 1) and is intended to represent the likelihood of an undesirable event's occurrence.

Risk assessment is part of the software project planning activity because the plan must identify the obstacles to successful project completion. It's not humanly possible to respond to every possible risk that might deter the completion of a project, so we must rank or assess risks by how severely or critically they threaten the successful completion of the project. Using historical risk data from similar projects can be helpful in risk assessment.

Risk assessment can be measured in cost ($) or on a relative scale (e.g., 1 = low, through 5 = high). Typically, it is difficult to estimate the cost of a risk to a project, but cost is still the more effective way to communicate a risk's impact to a project.

Risks should be ranked quantitatively, i.e., a risk should be given a numeric value that can be directly compared to the value of other risks. It is incorrect to simply compare the *probability estimates* of each risk. Nor is it correct to just compare the *impacts* of each risk to a project. But the *product* of the probability and the impact of a risk can be directly compared to that of other risks. In effect, we "weight" the impact of a risk by its probability of occurrence, i.e., we multiply the impact by the probability:

Risk Impact x Probability of Occurrence = Weighted Impact

For example:

Risk #1	Risk #2
Impact: $10,000	Impact: $75,000
Probability: 0.9	Probability: 0.1
Weighted impact: $9,000	Weighted impact: $7,500

Risk #1 has a weighted impact that is $1,500 more than Risk #2. Therefore, Risk #1 is a more critical risk than Risk #2.

To see if you understand the concept of assessing risk, try this risk assessment exercise.

Risk Management Plan

Another supplementary document to the SPP can be created to address potential project risks during the development of the software product. This document is referred to as the Software Risk Management Plan, and the IEEE provides an outline document, *IEEE Standard for Software Life Cycle Processes-Risk Management, Std 1540-2001*. The IEEE provides standards to identify and manage risks, and to identify roles of responsible parties, and auditing policy and review procedure.

4. Software Configuration Management

The purpose of Software Configuration Management is to manage changes to software. Specifically, SCM:

- identifies changes that are being made to software products
- controls these changes to minimize confusion
- ensures that the changes are made according to plan
- documents and reports the changes

There are two basic types of software development activities as described by Pressman (2005): phased activities and umbrella activities. SCM is an umbrella activity because it is needed throughout software development and even during the software maintenance phase.

Why do we want to manage change? After all, we are always changing our software, all the way from design through coding. The problem is that sometimes the changes run out of control. In large projects, if different development teams implement changes without coordination among them, the changes are likely to be incompatible. Then, when the software engineers try to fix the problems, they each insert additional changes that also may be incompatible and may make the situation even worse. The purpose of SCM —to manage change—is illustrated in our example. The goal of SCM is to ensure that all parts of a system are being built or changed according to the same plan, like a chorus all singing from the same sheet of music.

SCM manages changes to a variety of items that are used or produced during software development. These configuration items include: source and executable code, documents, databases, and programming tools. The activities that SCM performs include:

- **identification**—specifying the items that need to be under SCM
- **version control**—ensuring that specific configurations and variants of the product are preserved and managed
- **change control**—ensuring that changes to baselines are implemented in a coordinated and managed way
- **auditing**—evaluating the various versions of the product to verify that they are managed
- **reporting**—reporting on the findings of the auditing

A key concept is that of a **baseline**. A baseline is a specific version of one of the SCIs, resulting from the SCM version control. Changes to baselines are carefully managed.

Identification of Software Configuration Items (SCIs)

The first step in performing SCM is to identify the specific SCIs that will be controlled. Pressman (2005) refers to two types of items that can be identified:

- **basic objects**—items that are large enough to be identifiable and stand on their own, but small enough so that if they were broken up, they would not be identifiable
- **aggregate objects**—collections of basic objects and other aggregate objects

A basic object may be a section of a requirements specification. If you try to define a single paragraph as a basic object, you may find that it does not make sense on its own. When you include the rest of the section, however, the section becomes a basic object and can stand by itself. Aggregate objects would include other sections of the specification, other documents, the code, and essentially, every other item that, when combined, form the total system package.

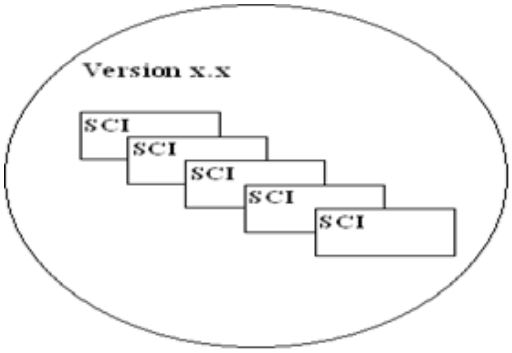
It is important to define these different levels of items because they have a direct relationship to how changes need to be managed. Because basic objects have a meaning or identity by themselves, they can be managed as a unit. Aggregate objects can then be managed as collections or configurations of specific basic objects.

What does *identification* of a software configuration item mean? For commercially purchased hardware and software, it means the tracking of information such as name, manufacturer, version, service-pack release, protocols, and standards. For custom-built application software, it means the tracking of information such as programming language version and manufacturer, operating system version and manufacturer, development tools, and shared subroutine libraries and versions.

Version Control and Change Control

At the foundation of SCM are the concepts of version control and change control. A **version** is a specific collection of system parts, each of which is uniquely identified in some way. This version is used as the baseline, indicating that it is specifically identified and managed and, therefore, can be recreated from its pieces in the configuration library or repository. In figure 5.6 we illustrate multiple SCIs grouped together to form a version.

Figure 5.6
Grouping SCIs to Make a Version



Change control is performed when new versions or new models are produced. There are two aspects of change control for an SCI:

- **access control**—where SCM controls who may check out and make a change to the SCI
- **synchronization control**—where SCM coordinates multiple changes to SCIs to ensure that conflicts are not created

Your first impression may be that you don't want someone snooping into your software development practices. Such control may seem to be "bean counting" and excessive. After all, who likes red tape? The counter to this position is that, first, it doesn't have to be onerous, and second, it is truly necessary to ensure that changes do not get out of control. Just because everyone in a choir is reading from the same sheet of music, it doesn't mean that they are being controlled too much. A certain amount of coordination is important.

Configuration Audits

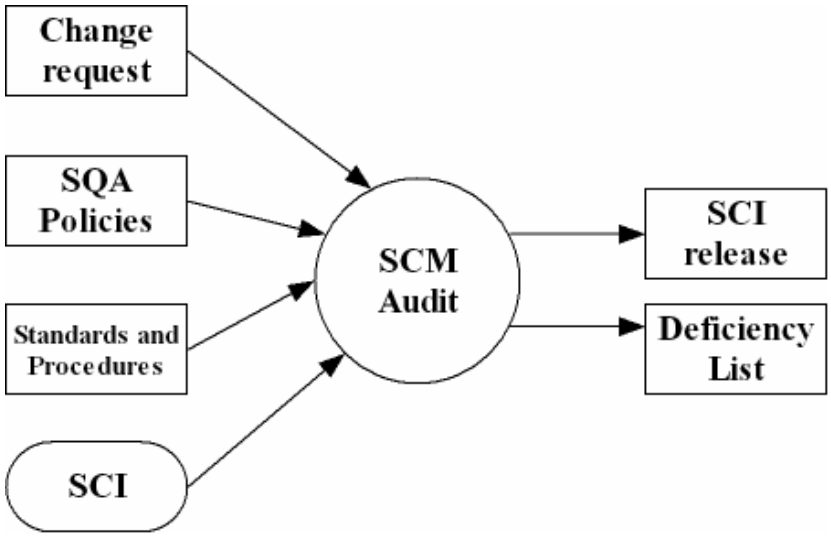
SCM must be systematically enforced throughout software development. Configuration audits are conducted by SCM personnel to verify that:

- the change process has identified all affected SCIs for each proposed change
- no unanticipated changes have surfaced in the baseline configuration
- authorized changes have been completely and consistently implemented for all affected configuration items

We summarize the role of the software configuration audit in figure 5.7. Using such audits, project personnel can assess how well the construction of the system has been coordinated across the software development

project.

Figure 5.7
Summary of Role of Software Configuration Audit



Software Configuration Management Plan

Another supplementary document to the SPP is the software configuration plan. The *IEEE Standard for Software Configuration Management Plans, Std 828-1998* can be used for guidance.

References

Boehm, B. *Software Engineering Economics*. Prentice-Hall, 1981.

Carnegie Mellon, Software Engineering Institute (2003a), Capability Maturity Model (SW-CMM) for Software (online). Available at: http://www.sei.cmu.edu/cmm/.

Carnegie Mellon, Software Engineering Institute (2003b), Upgrading from SW-CMM to CMMI, Pittsburgh, PA.(online). Available at: http://www.sei.cmu.edu/cmm/.

Carnegie Mellon, Software Engineering Institute (2003c), Risk Management Overview (online). Available at: http://www.sei.cmu.edu/risk/overview.html.

Carnegie Mellon, Software Engineering Institute (2003d), Risk Management Paradigm (online). Available at: http://www.sei.cmu.edu/risk/paradigm.html.

Futrell, Robert T., Donald F. Shafer, and Linda I. Shafer. *Quality Software Project Management*. New Jersey: Prentice Hall, 2002.

Koontz, H., and O'Donnell, C. (1972). Principles of Management: An Analysis of Managerial Functions, 5th edition, McGraw-Hill, New York.

Longstreet (2002). Fundamentals of Function Point Analysis (online). Available at www.SoftwareMetrics.com, retrieved 11/12/06.

Paulk, M.C., C.V. Weber, S.M. Garcia, M.B. Chrissis, and M. Bush. *Key practices of the capability maturity model, version 1.1* (Technical Report CMU/SEI-93-TR-25). Pittsburgh, PA: Carnegie Mellon University, 1993.

Pressman, Roger S. *Software engineering: A practitioner's approach* (fifth edition). New York: McGraw-Hill, 2001.

Pressman, Roger S. *Software engineering: A practitioner's approach* (sixth edition). New York: McGraw-Hill, 2005.

Van Scoy, Roger L. (1992). *Software development risk: Opportunity, not problem.* Software Engineering Institute (CMU/SEI-92-TR-30, ADA 258743) (online). Available at: http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.030.html.