SOFTWARE MAINTENANCE AND PROJECT MANAGEMENT

Software as an evolution entity

Lehman and Belady have studied the characteristics of the evolution of several software products [1980]. They have expressed their observations in the form of laws. Their important laws are given below.

1. **Lehman's first law:** A software product must change continually or become progressively less useful.
2. **Lehman's second law:** The structure of a program tends to degrade as more and more maintenance is carried out on it.
3. **Lehman's third law:** Over a program's lifetime, its rate of development is approximately constant.

SOFTWARE-CONFIGURATION MANAGEMENT ACTIVITIES

Configuration management is carried out through three principal activities:

1. Configuration identification,

2. Configuration control, and

3. Configuration accounting.

Briefly, these three activities can be described as:

Configuration Identification: Which parts of the system must be kept track of?

Configuration Control: Ensures that changes to a component happen smoothly.

Configuration Accounting: Keeps track of what has been changed, when, and why.

1. Configuration Identification.

The project manager normally classifies the objects associated with a software's development into three main categories: controlled, pre-controlled, and uncontrolled.

Controlled objects are those that are already put under configuration control. Pre-controlled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subject to configuration control. Controllable objects include both controlled and pre-controlled objects.

Typical controllable objects include:

- Requirement specifications documents
- Designing documents
- Tools used to build the system, such as compliers, linkers, lexical analyzers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports
- The configuration-management plan written during the project-planning phase lists all controlled objects.

2. **Configuration Control.**

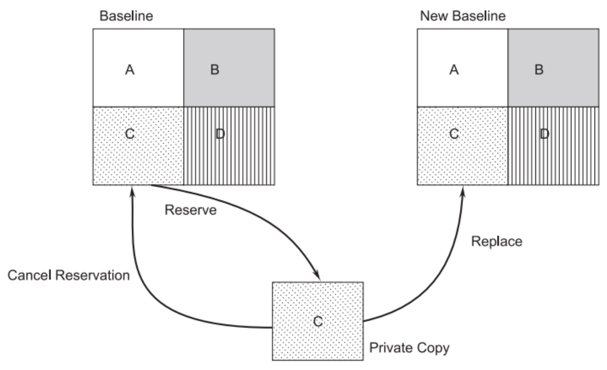
Configuration control is the process of managing changes to controlled objects. The configuration control system prevents unauthorized changes to any controlled object.

In order to change a controlled object, such as a module, a developer can get a private copy of the module from a reserve operation.

Configuration management tools allow only one person to reserve a module at any time. Once an object is reserved it does not allow anyone else to reserve this module until the reserved module is restored. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

The CCB is a group of people responsible for configuration management. The CCB evaluates the request based on its effect on the project, and the benefit due to change. An important reason for configuration control is people need a stable environment to develop a software product.

Suppose you are trying to integrate module A, with modules B and C. You cannot make progress if the developer of module C keeps changing it; this is especially frustrating if a change to module C forces you to recompile A. As soon as a document under configuration control is updated, the updated version is frozen and is called a baseline, as shown in Figure.



Configuration Control

3. Configuration Accounting.

Configuration accounting can be explained with two concepts:

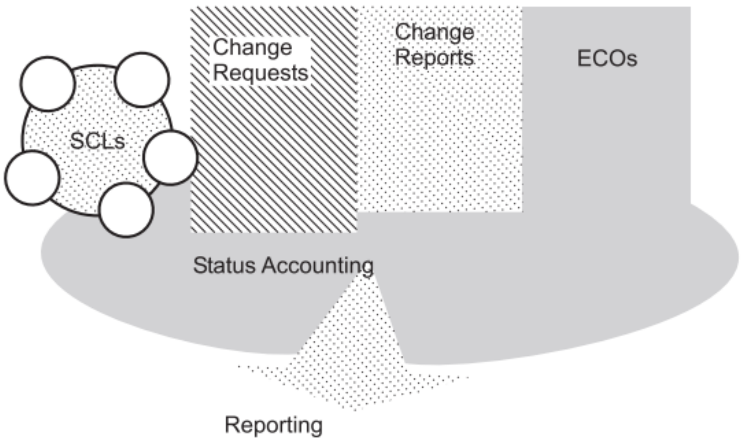
(*i*) *Status Accounting.* Once the changes in the baselines occur, some mechanisms must be used to record how the system evolves and what its current state is. This task is accomplished by status accounting. Its basic function is to record the activities related to the other SCM functions.

Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

(a) What happened? (b) Who did it?

(c) When did it happen? (d) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure above. 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. The key elements under status accounting are shown in Figure below.



Status Accounting

(*ii*) *Configuration Audit.* A software-configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review.

The audit asks and answers the following questions:

(a) Has the change specified in the ECO been made? Have any additional modifications been incorporated?

(b) Has a formal technical review been conducted to assess technical correctness?

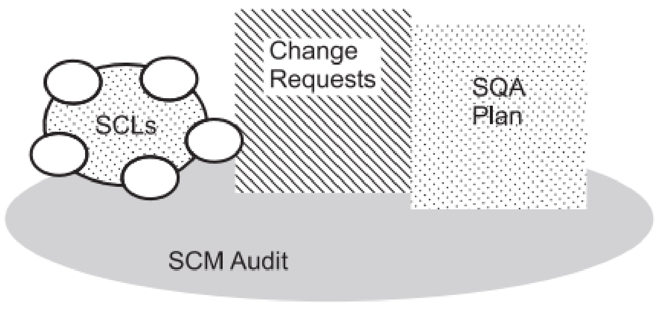
(c) Has the software process been followed and have software engineering standards been properly applied?

(d) 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?

(e) Have SCM procedures for noting the change, recording it, and reporting it been followed?

(f) Have all related SCIs been properly updated?

The SCM audit revolves around software-configuration items, change requests, and the software quality-assurance plan.



Configuration Accounting

CHANGE-CONTROL PROCESS

At any moment, the configuration-management team must know the state of any component or document in the system. Cashman and Holt (1980) suggest that we should always know the answers to the following questions:

Synchronization: When was the change made?

Identification: Who made the change?

Naming: What components of the system were changed?

Authentication: Was the change made correctly?

Authorization: Who authorized that the change be made?

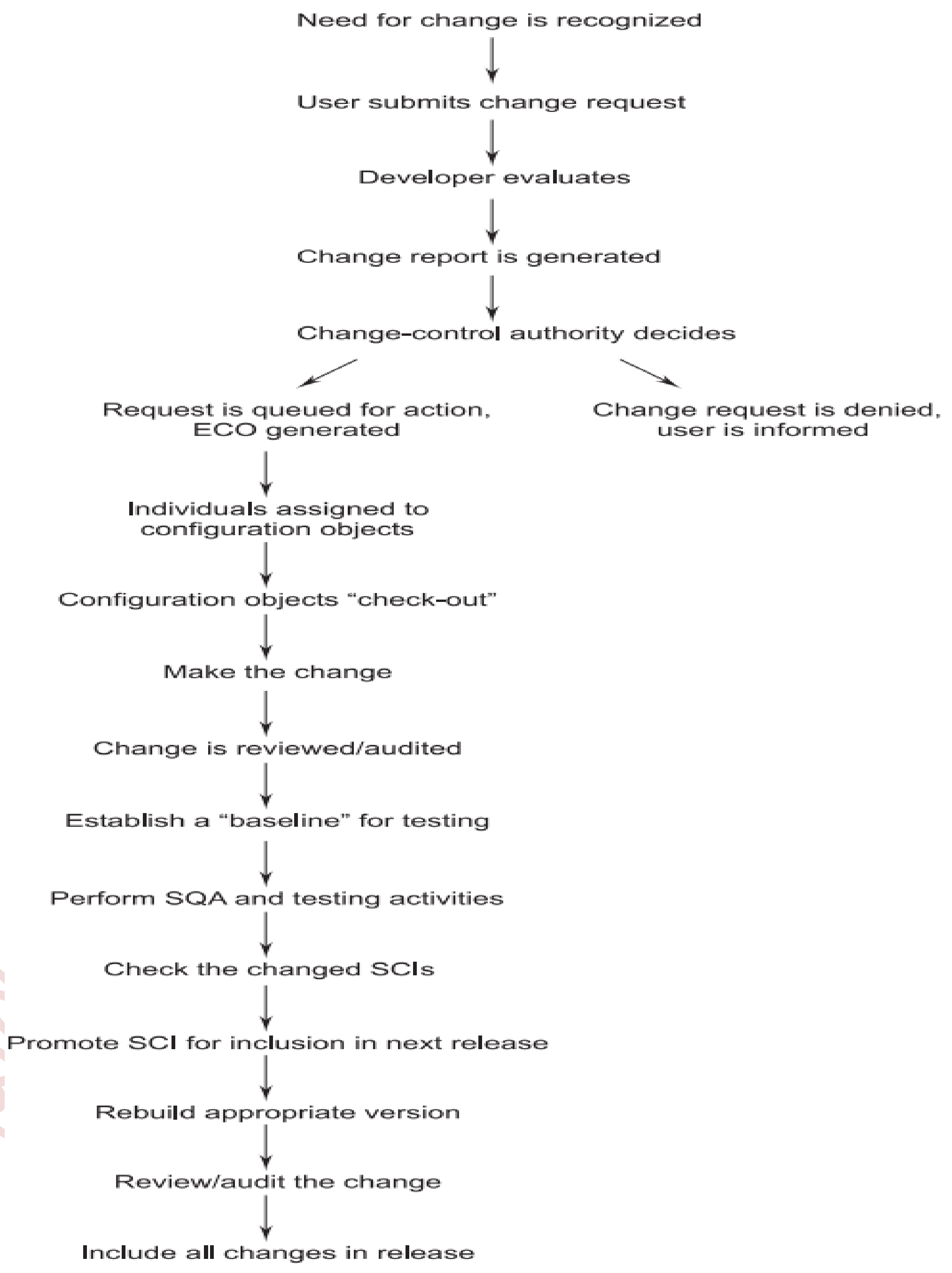
Routing: Who was notified of the change?

Cancellation: Who can cancel the request for a change?

Delegation: Who is responsible for the change?

Valuation: What is the priority of the change?

Change control combines human procedures and automated tools to provide a mechanism for the control of change. The change-control process is illustrated schematically in Figure.

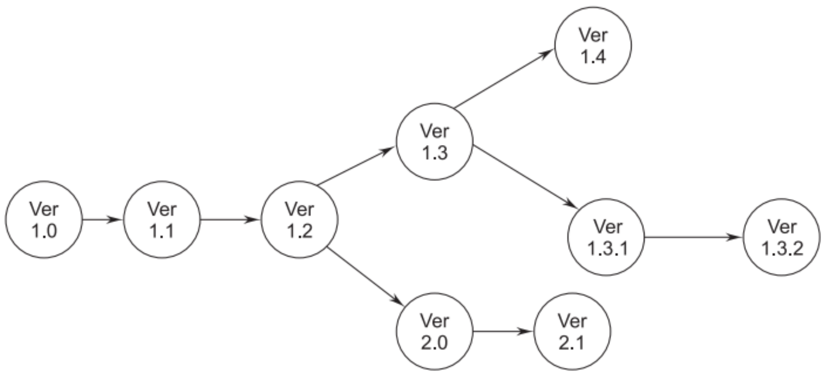

Need for change is recognized
User submits change request
Developer evaluates
Change report is generated
Change-control authority decides
Request is queued for action, ECO generated
Change request is denied, user is informed
Individuals assigned to configuration objects
Configuration objects "check-out"
Make the change
Change is reviewed/audited
Establish a "baseline" for testing
Perform SQA and testing activities
Check the changed SCIs
Promote SCI for inclusion in next release
Rebuild appropriate version
Review/audit the change
Include all changes in release

SOFTWARE-VERSION CONTROL

During the process of software evolution, many objects are produced; for example, files, electronic documents, paper documents, source code, executable code, and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept.

This report includes:

- The name of each source-code component, including the variations and revisions.
- The versions of the various compliers and linkers used.
- The name of the software staff who constructed the component.
- The date and the time at which it was constructed.



An Evolutionary Graph for a Different Versions of an Item

The above evolutionary graph depicts the evolution of a configuration item during the development life-cycle. The initial version of the item is given version number Ver 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as Ver 1.1 and Ver 1.2. After that, a major modification to Ver 1.2 is given a number Ver 2.0, and at the same time, a parallel version of the same item without the major modification is maintained and given a version number 1.3.

Commercial tools are available for version control, which perform one or more of the following tasks:

- Source-code control
- Revision control
- Concurrent-version control

SOFTWARE-CONFIGURATION MANAGEMENT

Software-configuration management deals with effectively tracking and controlling the configuration of a software product during its life-cycle. A new version of software is created when there is significant change in functionality, technology, or the hardware it runs on, etc.

On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc. The configuration-management team is responsible for assuring that each version or release is correct and stable before it is released for use, and that changes are made accurately and promptly.

Accuracy is critical, because we want to avoid generating new faults while correcting existing ones. Similarly, promptness is important, because fault detection and correction are proceeding at the same time that the test team searches for additional faults.

Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

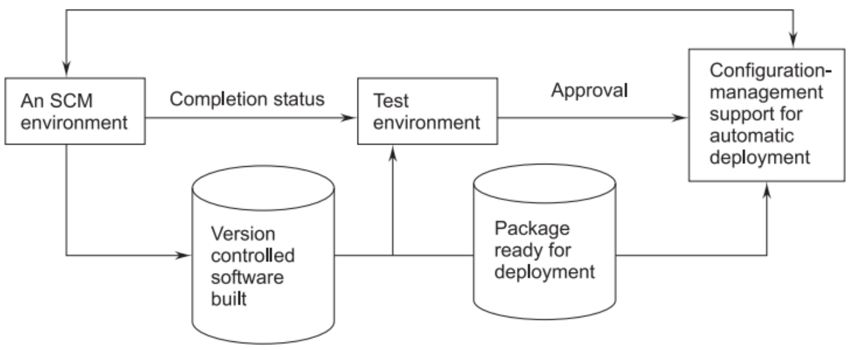
OR

Software-configuration management (SCM) is a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Configuration-management procedures define how to record and process proposed system changes, how to relate these to system components, and the methods used to identify different versions of the system. Configuration management tools are used to store versions of system components, build systems from these components, and track the release of system versions to customers.

The IEEE defines SCM as *the process of identifying and defining the items in the system, controlling the change of these items throughout their life-cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items.*

Figure shows a simple configuration-management procedure based on the rebuilding and deployment process.


An SCM environment
Completion status
Test environment
Approval
Configuration-management support for automatic deployment
Version controlled software built
Package ready for deployment

Versions and Releases

A configuration for a particular system is sometimes called a version. Thus, the initial delivery of a software package may consist of several versions, one for each platform or situation in which the software will be used.

For example, aircraft software may be built so that version 1 runs on Navy planes, version 2 runs on Air Force planes, and version 3 runs on commercial airliners. A new release of the software is an improved system intended to replace the old one.

Often, software systems are described as version *n*, release *m*, or as version *n.m*, where the number reflects the system's position as it grows and matures. Version *n* is sometimes intended to replace version *n*-1, and release *m* supersedes *m*-1.

Version and Release Management

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. New system versions should always be created by the CM team rather than the system developers, even when they are not intended for external release. This makes it easier to maintain consistency in the configuration database as only the CM team can change version information.

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. New system versions should always be created by the CM team rather than the system developers, even when they are not intended for external release. This makes it easier to maintain consistency in the configuration database as only the CM team can change version information.

A system release is a version that is distributed to customers. Each system release should either include new functionality or be intended for a different hardware platform. There are always many more versions of a system than releases as versions are created within an organization for internal development or testing that are never released to customers.

NEED FOR MAINTENANCE

Software maintenance is the activity associated with keeping an operational computer system continuously in tune with the requirements of users and data processing operations. The software maintenance process is expensive and risky and is very challenging.

There is a need for software maintenance due to the following reasons:

- Changes in user requirements with time
- Program/System problems
- Changing hardware/Software environment

- To improve system efficiency and throughout
- To modify the components
- To test the resulting product to verify the correctness of changes
- To eliminate any unwanted side effects resulting from modifications
- To augment or fine-tune the software
- To optimize the code to run faster
- To review standards and efficiency
- To make the code easier to understand and work with
- To eliminate any deviations from specifications

Maintenance To-Do List

- Correct errors
- Correct requirements and design flaws
- Improve the design
- Make enhancements
- Interface with other systems
- Convert for use with other hardware
- Migrate legacy systems
- Retire systems
- Major aspects
- Maintain control over the system's day-to-day functions
- Maintain control over system modification
- Perfect existing acceptable functions
- Prevent system performance from degrading to unacceptable levels

CATEGORIES OF MAINTENANCE

Maintenance may be **classified** into the four categories as follows:

- **Corrective -** reactive modifications to correct discovered problems.
- **Adaptive -** modifications to keep it usable in a changed or changing environment.
- **Perfective -** improve performance or maintainability.
- **Preventive -** modifications to detect and correct latent faults.

(*i*) **Corrective Maintenance.** Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems.

(*ii*) **Adaptive Maintenance.** Adaptive maintenance means changing the program functions. This is done to adapt to external environment changes. For example, the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares. The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be changed to adapt to the new system.

(*iii*) **Perfective Maintenance.** Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. For example, earlier data was sent from stores to headquarters on magnetic media but after stores were electronically linked via leased lines, the software was enhanced to send data via leased lines. As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through a maintenance management and software-modification audit. Software modification consists of program rewriting and system level upgradation.

(*iv*) **Preventive Maintenance.** Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

MAINTENANCE COSTS

In the 1970s, most of a software system's budget was spent on development. The ratio of development money to maintenance money was reversed in the 1980s, and various estimates place maintenance at 40 to 60% of the full life-cycle cost of a system (i.e., from development through maintenance to eventual retirement or replacement). However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of the software life-cycle to reduce maintenance costs. The defect repair ratio increases heavily from the analysis phase to the implementation phase as shown in Table. Therefore, more effort during development will certainly reduce the cost of maintenance.

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Defect Repair Ratio

Factors Affecting Effort

There are many other factors that contribute to the effort needed to maintain a system. These factors include the following:

- Application type
- System novelty
- Turnover and maintenance staff availability
- System life-span
- Dependence on a changing environment
- Hardware characteristics
- Design quality
- Code quality
- Documentation quality
- Testing quality

SOFTWARE-PROJECT ESTIMATION

Software-project estimation is the process of estimating various resources required for the completion of a project. Effective software-project estimation is an important activity in any software-development project.

Underestimating software projects and understaffing it often leads to low-quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company.

On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company.

Software-project estimation mainly encompasses the following steps:

1. **Estimating the Size of the Project.** There are many procedures available for estimating the size of a project, which are based on quantitative approaches, such as estimating lines of code or estimating the functionality requirements of the project called function points.

2. **Estimating Efforts Based on Person-months or Person-hours.** Person-month is an estimate of the personal resources required for the project.

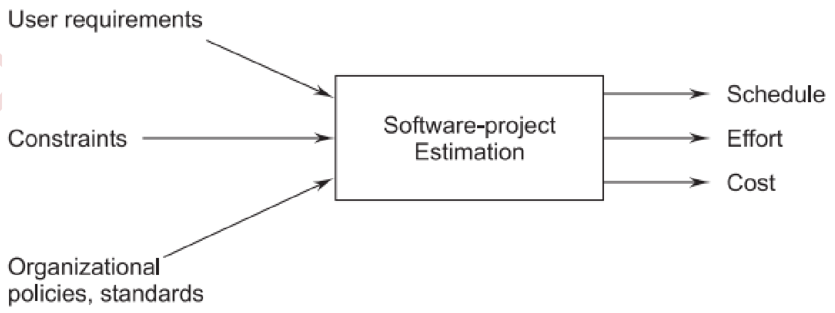
3. **Estimating Schedule in Calendar Days/Month/Year Based on Total Person-months Required and Manpower Allocated to the Project.** The duration in calendar month = Total person-months/Total manpower allocated.

4. **Estimating Total Cost of the Project Depending on the Above and Other Resources.** In a commercial and competitive environment, software-project estimation is crucial for managerial decision-making.

Table below gives the relationship between various management functions and software metrics/indicators. Project estimation and tracking help to plan and predict future projects and provide baseline support for project management and supports decision-making.

Activity	Tasks Involved
Planning	Cost estimation, planning for training of manpower, project scheduling, and budgeting the project.
Controlling	Size metrics and schedule metrics help the manager to keep control of the project during execution.
Monitoring/improving	Metrics are used to monitor progress of the project and wherever possible sufficient resources are allocated to improve it.

Software-Project Estimation


User requirements
Constraints
Organizational policies, standards
Software-project Estimation
Schedule
Effort
Cost

Software-project Estimation

Estimating Size

Estimating the size of the software to be developed is the very first step to make an effective estimation of the project. Customer requirements and system specifications form a baseline for estimating the size of a software. At a later stage of the project, system design documents can provide additional details for estimating the overall size of the software.

- The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.
- The other way of estimation is through product feature/functionality. The system is divided into several subsystems depending on functionality, and the size of each subsystem is calculated.

Estimating Effort

Once the size of software is estimated, the next step is to estimate the effort based on the size. The estimation of effort can be made from the organizational specifics of the software-development life-cycle. The development of any application software system is more than just the coding of the system. Depending on deliverable requirements, the estimation of effort for a project will vary.

Efforts are estimated in the number of person-months.

- The best way to estimate effort is based on the organization’s own historical data of development processes. Organizations follow a similar development life-cycle when developing various applications.
- If the project is of a different nature, which requires the organization to adopt a different strategy for development, then different models based on algorithmic approaches can be devised to estimate the required effort.

Estimating Schedule

The next step in the estimation process is estimating the project schedule from the effort estimated. The schedule for a project will generally depend on human resources involved in a process. Efforts in person-months are translated to calendar months.

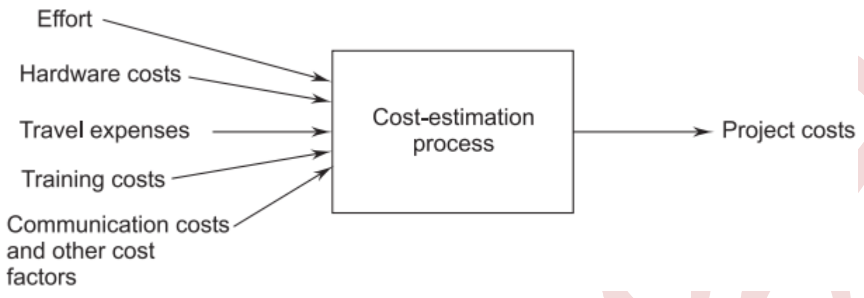
Schedule estimation in calendar months can be calculated using the following model [McConnell]:

$$\text{Schedule in calendar months} = 3.0*(\text{person-months})^{1/3}$$

The parameter 3.0 is variable, used depending on the situation that works best for the organization.

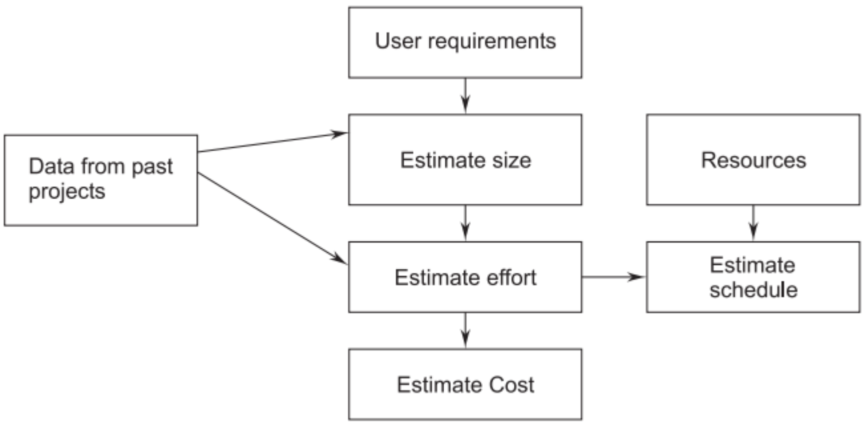
Estimating Cost

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters, such as hardware, travel expenses, telecommunication costs, training costs, etc. Figure depicts the cost-estimation process.



Cost-estimation Process

Figure below depicts the project-estimation process.



Project-estimation Process

Once the estimation is complete, we may be interested to know how accurate the estimates re. The answer to this is “we do not know until the project is complete.” There is always some uncertainty associated with all estimation techniques. The accuracy of project estimation will depend on the following:

- Accuracy of historical data used to project the estimation.
- Accuracy of input data to various estimates.
- Maturity of an organization’s software-development process.

The following are some of the reasons cost estimation can be difficult:

- Software-cost estimation requires a significant amount of effort. Sufficient time is usually not allocated for planning.

- Software-cost estimation is often done hurriedly, without an appreciation for the actual effort required and is far from realistic.
- Lack of experience for developing estimates, especially for large projects.
- An estimator uses the extrapolation technique to estimate, ignoring the non-linear aspects of the software-development process.

Reasons for Poor/Inaccurate Estimation

The following are some of the reasons for poor and inaccurate estimation:

- Requirements are imprecise. Also, requirements change frequently.
- The project is new and is different from past projects handled.
- Non-availability of enough information about past projects.
- Estimates are forced to be based on available resources.
- Cost and time tradeoffs.

If we elongate the project, we can reduce overall costs. Usually, customers and management do not like long project durations. There is always the shortest possible duration for a project, but it comes at a cost.

The following are some of the problems associated with estimates:

- Estimating size is often skipped and a schedule is estimated, which is of more relevance to management.
- Estimating size is perhaps the most difficult step, which has a bearing on all other estimates.
- Let us not forget that even good estimates are only projections and subject to various risks.
- Organizations often give less importance to collection and analysis of historical data of past development projects. Historical data is the best input to estimate a new project.
- Project managers often underestimate the schedule because management and customers often hesitate to accept a prudent realistic schedule.

Project-estimation Guidelines

Some guidelines for project estimation are as follows:

- Preserve and document data pertaining to past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimates. Associate people who will work on the project to reach a realistic and more accurate estimate.
- Use software-estimation tools.
- Re-estimate the project during the life-cycle of the development process.
- Analyze past mistakes in the estimation of projects.

CONSTRUCTIVE COST MODEL (COCOMO)

COCOMO stands for Constructive Cost Model. It was introduced by Barry Boehm in 1981. It is perhaps the best known and most thoroughly documented of all software-cost estimation models. It provides the following three levels of models:

- **Basic COCOMO:** A single-value model that computes software-development costs as a function of an estimate of LOC.
- **Intermediate COCOMO:** This model computes development costs and effort as a function of program size (LOC) and a set of cost drivers.
- **Complete COCOMO:** This model computes development effort and costs which incorporates all characteristics of intermediate levels with assessment of cost implications in each step of development (analysis, design, testing, etc.).

This model may be applied to three classes of software projects as given below:

- **Organic:** Small-size project. A simple software project where the development team has good knowledge of the application.
- **Semi-detached:** An intermediate-size project, and the project is based on rigid and semi-rigid requirements.
- **Embedded:** The project is developed under hardware, software, and operational constraints. Examples are embedded software and flight control software.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

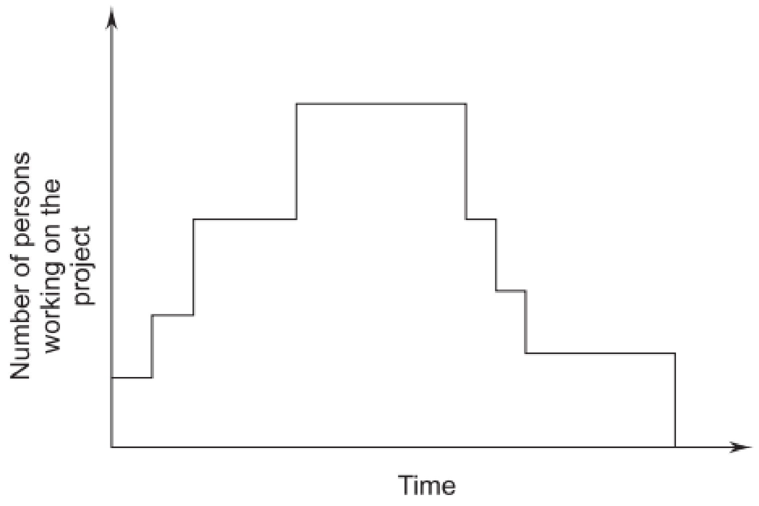
$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \qquad \text{PM}$$
$$T_{dev} = b_1 \times (\text{Effort})^{b_2} \qquad \text{Months}$$

where KLOC is the estimated size of the software product expressed in Kilo Lines and Code a_1, a_2, b_1, b_2 are constants of the software product.

T_{dev} is the estimated time to develop the software, expressed in months.

Effort is the total effort required to develop the software product, expressed in person-months (PM).

The person-month curve is shown in Figure below.

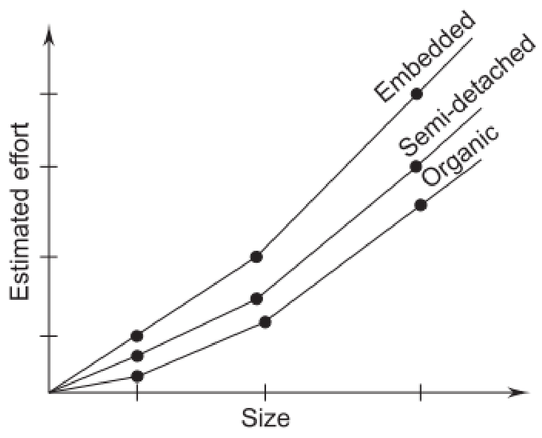


Person-month Curve

Estimation of Development Effort

$$\text{Organic:} \qquad \text{Effort} = 2.4\,(\text{KLOC})^{1.05}\ \text{PM}$$
$$\text{Semi-detached:} \qquad \text{Effort} = 3.0\,(\text{KLOC})^{1.12}\ \text{PM}$$
$$\text{Embedded:} \qquad \text{Effort} = 3.6\,(\text{KLOC})^{1.20}\ \text{PM}$$

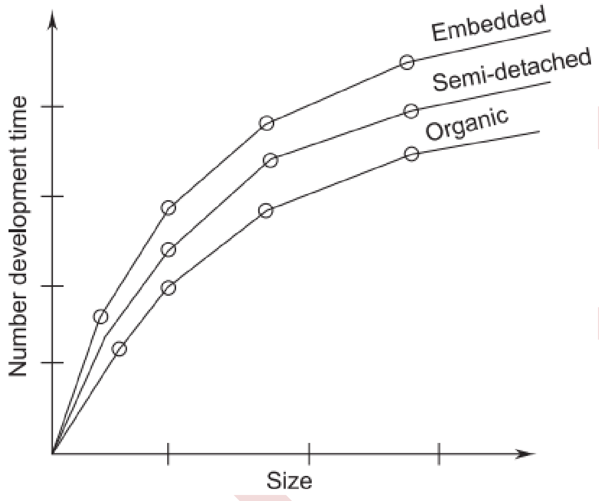
Figure shows a plot of the estimated effort versus the size for various product sizes. From this we can observe that the effort is almost linearly proportional to the size of the software product.



Estimation of Development Time

Organic:	$T_{dev} = 2.5\,(\text{Effort})^{0.38}$ Months
Semi-detached:	$T_{dev} = 2.5\,(\text{Effort})^{0.35}$ Months
Embedded:	$T_{dev} = 2.5(\text{Effort})^{0.32}$ Months

Figure below shows a plot of development time versus product size. From this, we can observe that the development time is a sub-linear function of the size of the product.



Intermediate COCOMO Model

The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained through the basic COCOMO expression by using a set of 15 cost drivers (multipliers) based on various attributes of software development.

Driver Type	Code	Cost Driver
Product attributes	RELY	Required software reliability
	DATA	Database size
	CPLX	Product complexity
Computer attributes	TIME	Execution time constraints
	STOR	Main storage constraints
	VIRT	Virtual machine volatility—degree to which the operating system changes
	TURN	Computer turnaround time
Personnel attributes	ACAP	Analyst capability
	AEXP	Application experience
	PCAP	Programmer capability
	VEXP	Virtual machine (i.e., operation system) experience
	LEXP	Programming-language experience
Project attributes	MODP	Use of modern programming practices
	TOOL	Use of software tools
	SCED	Required development schedule

COCOMO Intermediate Cost Drivers

Complete COCOMO Model

The shortcomings of both basic and intermediate COCOMO models are that they:

- Consider a software product a single homogenous entity. However, most large systems are made up of several smaller subsystems. Some subsystems may be considered organic, some embedded, etc. For some, the reliability requirements may be high, and so on.
- Consider the cost of each subsystem separately.
- Consider the costs of the subsystems but add them separately to obtain total cost.
- Reduce the margin of error in the final estimate.

A large amount of work has been done by Boehm to capture all significant aspects of software development. It offers a means for processing all the project characteristics to construct a software estimate. The complete model introduces two more capabilities:

1. **Phase-Sensitive Effort Multipliers.** Some phases (design, programming, and integration/test) are more affected than others by factors defined by the cost drivers. The complete model provides a set of phase-sensitive effort multipliers for each cost driver. This helps in determining the manpower allocation for each phase of the project.

2. **Three-Level Product Hierarchy.** Three product levels are defined. These are module, subsystem, and system levels. The ratings of the cost drivers are done at appropriate levels; that is, the level at which it is most susceptible to variation.

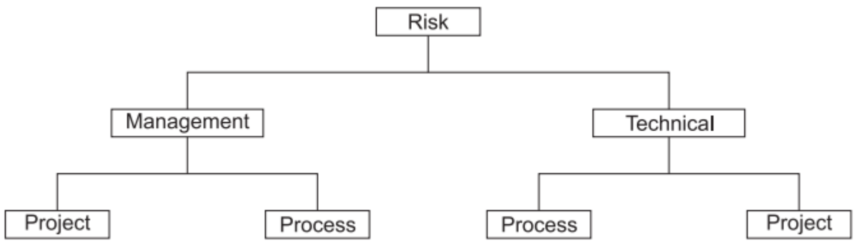
SOFTWARE-RISK ANALYSIS AND MANAGEMENT

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Risk is defined as an exposure to the chance of injury or loss. i.e., risk implies that there is a possibility that something negative may happen. In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule.

Risk Management

Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal. Risk management is a scientific process based on the application of game theory, decision theory, probability theory, and utility theory. The Software Engineering Institute (SEI) classifies the risk hierarchy as shown in Figure.



Risk Hierarchy

Risk scenarios may emerge out of management challenges and technical challenges in achieving specific goals and objectives. Risk management must be performed regularly throughout the achievement life-cycle.

Risks are dynamic, as they change over time. Risk management should not be regarded as an activity integral to the main process of achieving specific goals and objectives.

Risk and its management should not be treated as an activity outside the main process of achievement. Risk is managed best when risk management is implemented as a mainstream function in the software-development and goal achievement processes.

Management of Risks

Risk management plays an important role in ensuring that the software product is error-free. Firstly, risk management takes care that the risk is avoided, and if it is not avoidable, then the risk is detected, controlled, and finally recovered.

Risk-management Categories

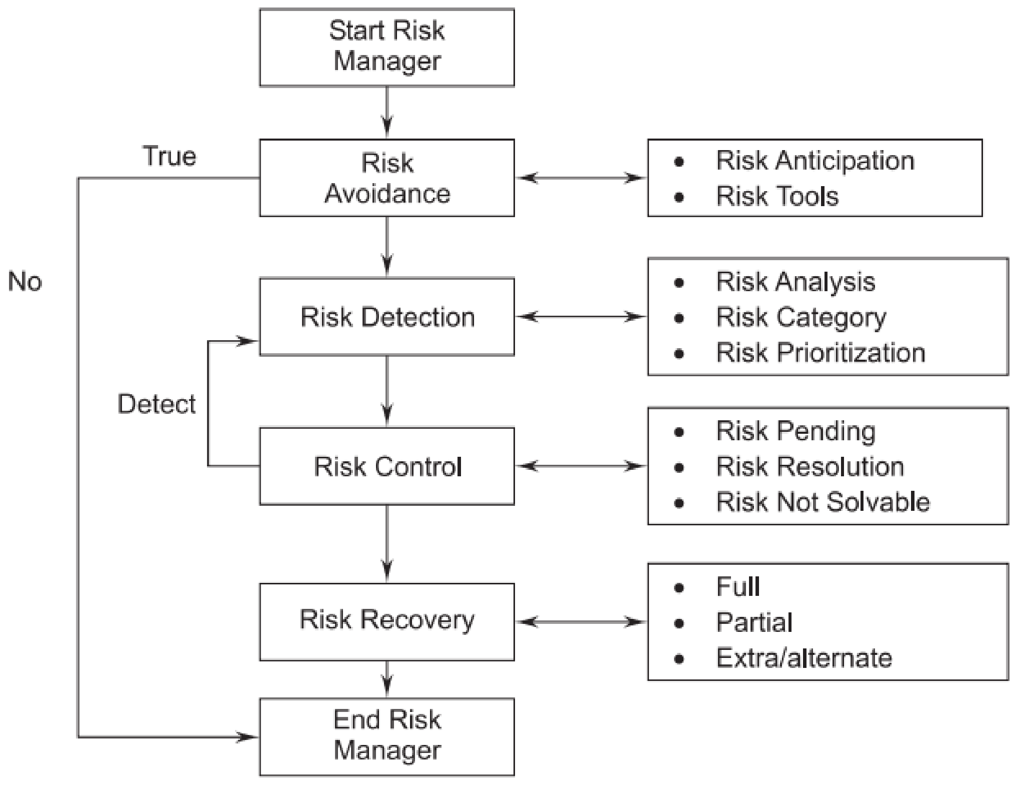
A priority is given to risk and the highest priority risk is handled first. Various factors of the risk include who are the involved team members, what hardware and software items are needed, where, when, and why. The risk manager does scheduling of risks.

Risk management can be further categorized as follows:

1. Risk Avoidance
   - Risk anticipation
   - Risk tools
2. Risk Detection
   - Risk analysis
   - Risk category
   - Risk prioritization

3. Risk Control
   - Risk pending
   - Risk resolution
   - Risk not solvable
4. Risk Recovery
   - Full
   - Partial
   - Extra/alternate feature

Figure depicts a risk-management tool.



Risk-management tool

From Figure, it is clear that the first phase is to avoid risk by anticipating and using tools from previous project histories. In the case where there is no risk, the risk manager stops. In the case of risk, detection is done using various risk analysis techniques and further prioritizing risks. In the next phase, risk is controlled by pending risks, resolving risks, and in the worst case (if the risk is not solved), lowering the priority. Lastly, risk recovery is done fully, partially, or an alternate solution is found.

1. Risk Avoidance

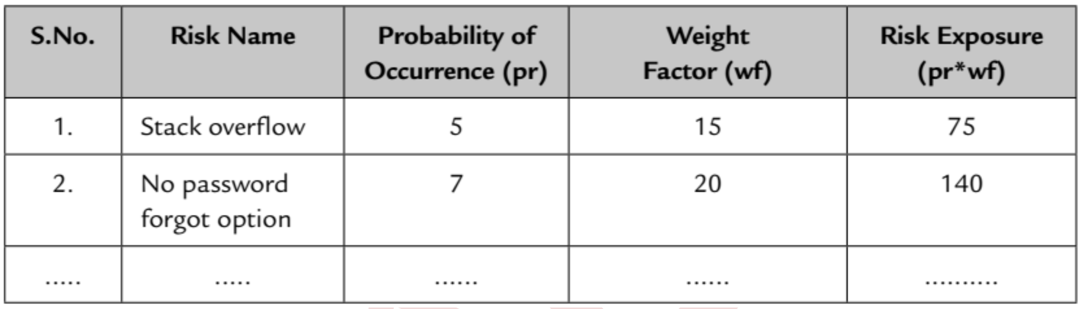
Risk Anticipation: Various risk anticipation rules are listed according to standards from previous projects, experience, and also as mentioned by the project manager.

Risk Tools: Risk tools are used to test whether the software is risk-free. The tools have a built-in database of available risk areas and can be updated depending upon the type of project.

2. Risk Detection

The risk-detection algorithm detects a risk and it can be categorically stated as the following:

Risk Analysis: In this phase, the risk is analyzed with various hardware and software parameters as probabilistic occurrence (pr), weight factor (wf) (hardware resources, lines of code, people), and risk exposure (pr * wf). Table below depicts a risk-analysis table.



S.No.	Risk Name	Probability of Occurrence (pr)	Weight Factor (wf)	Risk Exposure (pr*wf)
1.	Stack overflow	5	15	75
2.	No password forgot option	7	20	140
.....	.....	......	......	....

The maximum value of risk exposure indicates that the problem has to be solved as soon as possible and be given high priority. A risk-analysis table is maintained as shown in Table.

Risk Category: Risk identification can come from various factors, such as persons involved in the team, management issues, customer specification and feedback, environment, commercial, technology, etc. Once the proper category is identified, priority is given depending upon the urgency of the product.

Risk Prioritization: Depending upon the entries of the risk-analysis table, the maximum risk exposure is given high priority and has to be solved first.

3. Risk Control.

Once the prioritization is done, the next step is to control various risks as follows:

Risk Pending: According to the priority, low-priority risks are pushed to the end of the queue with a view of various resources (hardware, manpower, software) and if it takes more time their priority is made higher.

Risk Resolution: The risk manager decides how to solve the risk.

Risk Elimination: This action leads to serious errors in the software.

Risk Transfer: If the risk is transferred to some part of the module, then the risk-analysis table entries get modified. And again, the risk manager will control high-priority risks.

Disclosures: Announce the smaller risks to the customer or display message boxes as warnings so that the user take proper steps during data entry, etc.

Risk not Solvable: If a risk takes more time and more resources, then it is dealt with in its totality on the business side of the organization and thereby the customer is notified, and the team member proposes an alternate solution. There is a slight variation in the customer specifications after consultation.

4. Risk Recovery

Full: The risk-analysis table is scanned and if the risk is fully solved, then the corresponding entry is deleted from the table.

Partial: The risk-analysis table is scanned and due to partially solved risks, the entries in the table are updated and thereby priorities are also updated.

Extra/alternate features: Sometimes it is difficult to remove risks, and in that case, we can add a few extra features, that solve the problem. Therefore, some coding is done to resolve the risk. This is later documented or the customer is notified.

Sources of Risks

There are two major sources of risk, which are as follows:

1. ***Generic Risks.*** Generic risks are the risks common to all software projects. For example, requirement misunderstanding, allowing insufficient time for testing, losing key personnel, etc.

2. ***Project-Specific Risks.*** A vendor may be promising to deliver particular software by a particular date, but is unable to do it.

Types of Risks

There are three types of risks, which are discussed as follows:

1. ***Product Risks.***

2. ***Business Risks.***

3. ***Project Risks.***

1. ***Product Risks.*** Product risks are risks that affect the quality or performance of the software being developed. This originates from conditions, such as unstable requirement specifications, not being able to meet the design specifications affecting software performance, and uncertain test specifications. In view of the software product risks, there is a risk of losing the business and facing strained customer relations. For example, CASE tools under performance.

2. ***Business Risks.*** Business risks are risks that affect the organization developing or procuring the software. For example, technology changes and product competition.

The top five business risks are:

- Building an excellent product or system that no one really wants (market risks).
- Building a product that no longer fits into the overall business strategy for the company (strategic risk).
- Building a product that the sales force doesn't understand how to sell.
- Losing the support of senior management due to a change in focus or a change in people (management risk).
- Losing budgetary or personnel commitment (budget risks).

3 ***Project Risks.*** Project risks are risks that affect the project schedule or resources. These risks occur due to conditions and constraints about resources, relationships with vendors and contractors, unreliable vendors, and lack of organizational support. Funding is the significant project risk management has to face. It occurs due to initial budgeting constraints and unreliable customer payments. For example, staff turnover, management change, hardware uninvertibility.