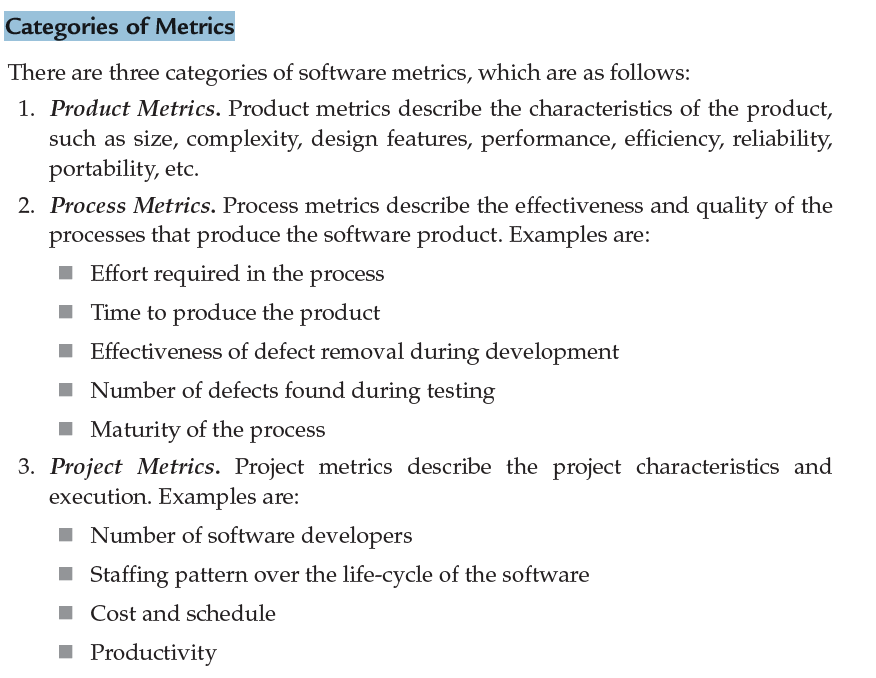
**Unit 2**

**SOFTWARE MEASUREMENT AND METRICS (Chpt – 2)**

**SOFTWARE METRICS :**

* Software metrics are quantifiable measures that could be used to measure different characteristics of a software system or the software-development process.
* Software metrics can be defined as “The continuous application of measurement-based techniques to the software-development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.”

****

**Measures, Metrics, and Indicators :**

* A *measure* provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.
* *Measurement* is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terminology* [IEE93b] defines *metric* as “a quantitative measure of the degree to which a system, component, or process possesses a given attribute.”
* A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test).
* An *indicator* is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the product to make things better.

**Measurement Principles :**

Roche [Roc94] suggests a measurement process that can be characterized by five activities:

• *Formulation.* The derivation of software measures and metrics appropriate for the representation of the software that is being considered.

• *Collection.* The mechanism used to accumulate data required to derive the formulated metrics.

• *Analysis.* The computation of metrics and the application of mathematical tools.

• *Interpretation.* The evaluation of metrics resulting in insight into the quality of the representation.

• *Feedback.* Recommendations derived from the interpretation of product metrics transmitted to the software team.

**Metrics for Object-Oriented Design :**

Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design:

* **Size.** Size is defined in terms of four views: population, volume, length, and functionality. *Population* is measured by taking a static count of OO entities such as classes or operations. *Volume* measures are identical to population measures but are collected dynamically—at a given instant of time. *Length* is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). *Functionality* metrics provide an indirect indication of the value delivered to the customer by an OO application.
* **Complexity.** Whitmire views complexity in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.
* **Coupling.** The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.
* **Sufficiency.** Whitmire defines *sufficiency* as “the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application.” Stated another way, we ask: “What properties does this abstraction (class) need to possess to be useful to me?” [Whi97]. In essence, a design component (e.g., a class) is *sufficient* if it fully reflects all properties of the application domain object that it is modeling—that is, that the abstraction (class) possesses the features required of it.
* **Completeness.** The only difference between completeness and sufficiency is “the feature set against which we compare the abstraction or design component” [Whi97]. Sufficiency compares the abstraction from the point of view of the current application. *Completeness* considers multiple points of view, asking the question: “What properties are required to fully represent the problem domain object?”
* **Cohesion.** The cohesiveness of a class is determined by examining the degree to which “the set of properties it possesses is part of the problem or design domain” [Whi97].
* **Primitiveness.** A characteristic that is similar to simplicity, primitiveness (applied to both operations and classes) is the degree to which an operation is atomic—that is, the operation cannot be constructed out of a sequence of other operations contained within a class. A class that exhibits a high degree of primitiveness encapsulates only primitive operations.
* **Similarity.** The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.
* **Volatility.** Volatility of an OO design component measures the likelihood that a change will occur.

**Operation-Oriented Metrics :**

* Churcher and Shepperd discuss this when they state: “Results of recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [Wil93], suggesting that connectivity structure of a system may be more important than the content of individual modules.”

Three simple metrics, proposed by Lorenz and Kidd [Lor94], are appropriate:

* **Average operation size (OSavg).** Size can be determined by counting the number of lines of code or the number of messages sent by the operation. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well allocated within a class.
* **Operation complexity (OC).** The complexity of an operation can be computed using any of the complexity metrics proposed for conventional software [Zus90]. Because operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.
* **Average number of parameters per operation (NPavg).** The larger the number of operation parameters, the more complex the collaboration between objects. In general, NPavg should be kept as low as possible.

**User Interface Design Metrics :**

Sears [Sea93] suggests that *layout appropriateness* (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the “cost” of the transition from one layout entity to the next all contribute to the appropriateness of the interface.

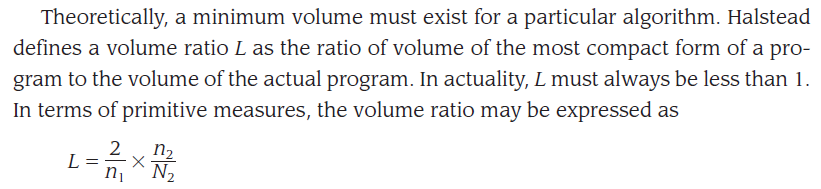
* + A study of Web page metrics [Ivo01] indicates that simple characteristics of the elements of the layout can also have a significant impact on the perceived quality of the GUI design. The number of words, links, graphics, colors, and fonts (among other characteristics) contained within a Web page affect the perceived complexity and quality of that page.
  + It is important to note that the selection of a GUI design can be guided with metrics such as LA, but the final arbiter should be user input based on GUI prototypes.
  + Nielsen and Levy [Nie94] report that “one has a reasonably large chance of success if one chooses between interface [designs] based solely on users’ opinions.
  + Users’ average task performance and their subjective satisfaction with a GUI are highly Correlated.”

**METRICS FOR SOURCE CODE :**

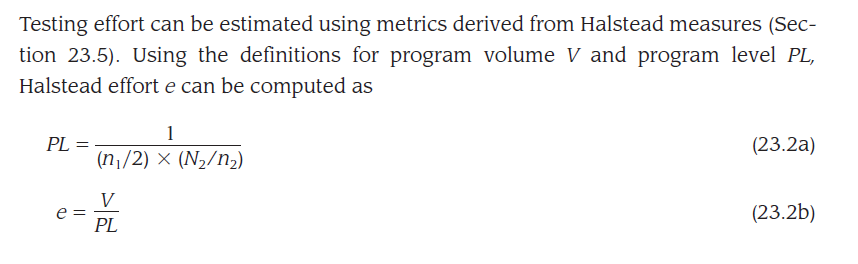
* Halstead’s theory of “software science” [Hal77] proposed the first analytical “laws” for computer software.
* Halstead assigned quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete.
* The measures are:
* *n*1 number of distinct operators that appear in a program
* *n*2 number of distinct operands that appear in a program
* *N*1 total number of operator occurrences
* *N*2 total number of operand occurrences
* Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software
* Halstead shows that length *N* can be estimated
* *N =* *n*1 log2 *n*1 + *n*2 log2 *n*2

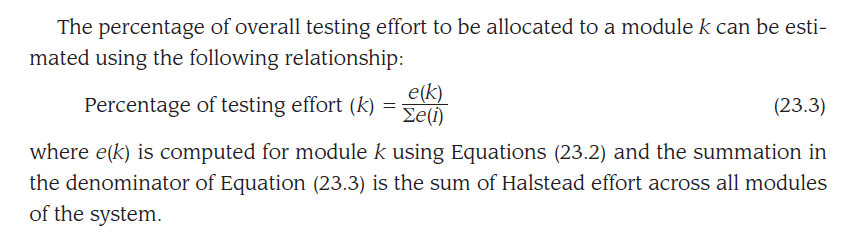
and program volume may be defined

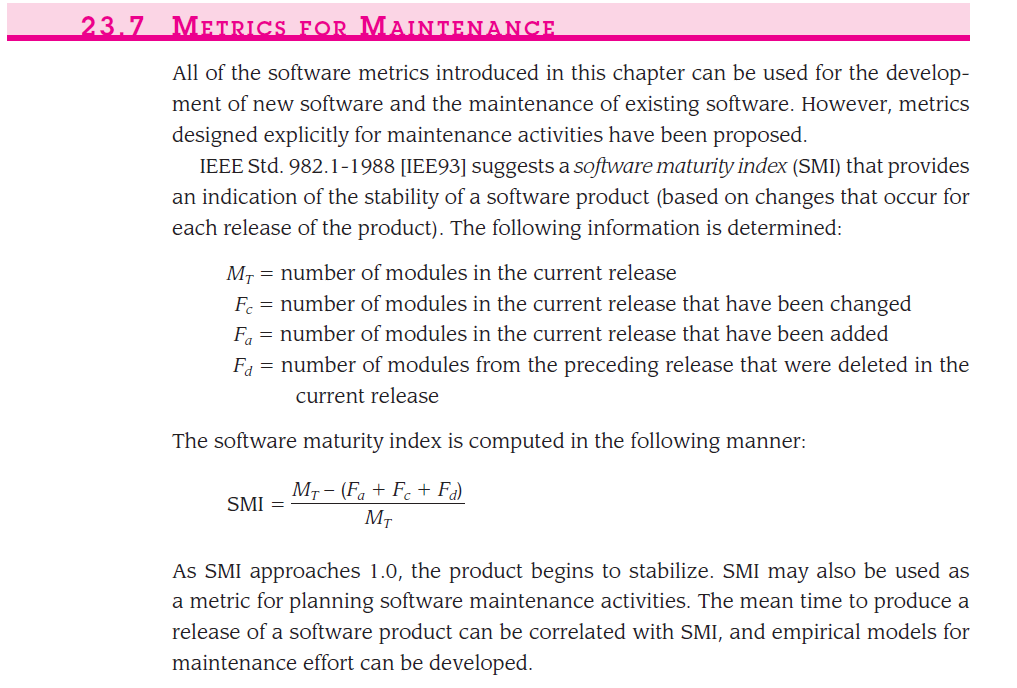
* *V =* *N* log2 (*n*1 *n*2)
* It should be noted that *V* will vary with programming language and represents the volume of information (in bits) required to specify a program.
* Halstead defines a volume ratio *L* as the ratio of volume of the most compact form of a program to the volume of the actual program.



**Halstead Metrics Applied to Testing :**

****

****

****

**METRICS FOR SOFTWARE QUALITY :**

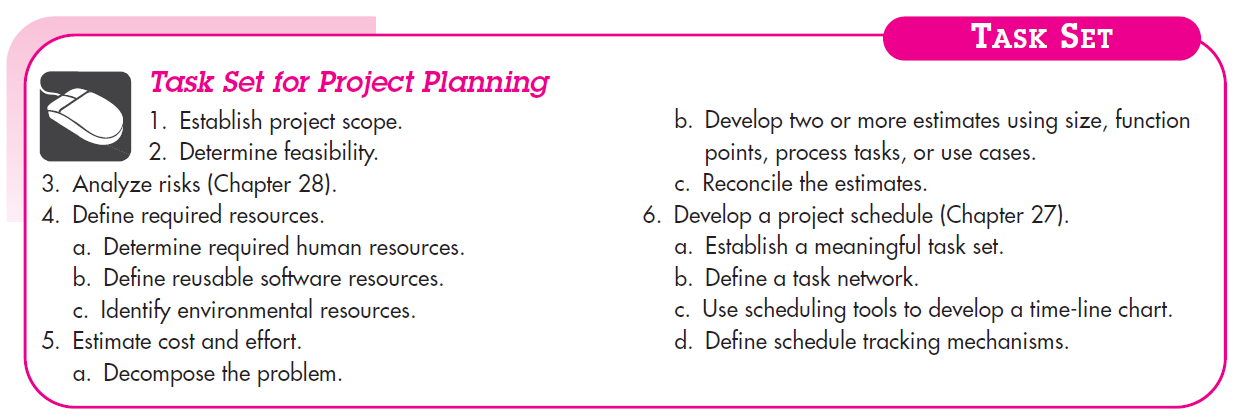
* Measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [Gil88] suggests definitions and measures for each.
* **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.
* **Maintainability.** Software maintenance and support accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, indirect measures are used. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
* **Integrity.** Software integrity has become increasingly important in the age of cyber terrorists and hackers. This attribute measures a system’s ability to withstand attacks (both accidental and intentional) to its security.
* Attacks can be made on all three components of software: programs, data, and documentation. To measure integrity, two additional attributes must be defined: threat and security.
* *Threat* is the probability that an attack of a specific type will occur within a given time.
* *Security* is the probability that the attack of a specific type will be repelled.
* **Usability.** If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify ease of use.

**Software Project Management (Chpt -3)**

**THE PROJECT PLANNING PROCESS**

* The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
* The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.
* Software project planning encompasses five major activities:

1. Estimation
2. Scheduling
3. Risk analysis
4. Quality management planning
5. Change management planning

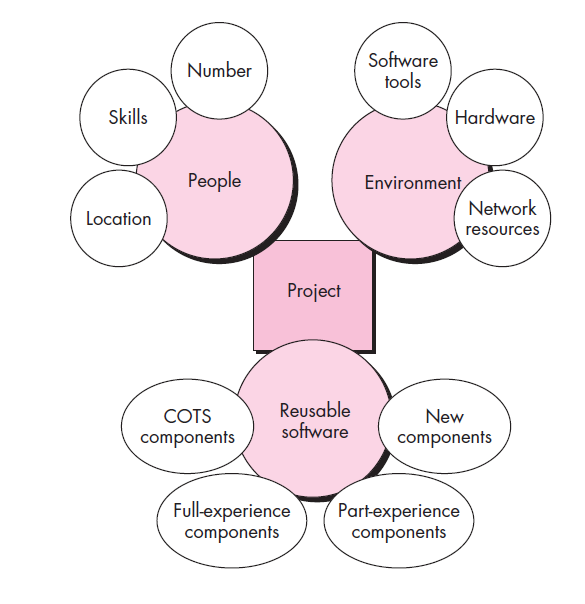


**SOFTWARE SCOPE AND FEASIBILITY**

* Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: “Can we build software to meet this scope? Is the project feasible?”
* After the scope is resolved, feasibility is addressed
* Software feasibility has four dimensions :
* Technology – Is the project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application's needs?
* Finance – Is it financially feasible? Can development be completed at a cost that the software organization, its client, or the market can afford?
* Time – Will the project's time-to-market beat the competition?
* Resources – Does the software organization have the resources needed to succeed in doing the project?

**RESOURCES :**

* The second planning task is estimation of the resources required to accomplish the software development effort.
* The three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools).
* Each resource is specified with four characteristics:
* description of the resource
* a statement of availability
* time when the resource will be required, and
* duration of time that the resource will be applied.

****

* Planners need to select the number and the kind of people skills needed to complete the project
* They need to specify the organizational position and job specialty for each person
* Small projects of a few person-months may only need one individual
* Large projects spanning many person-months or years require the location of the person to be specified also
* The number of people required can be determined only after an estimate of the development effort

**Reusable Software Resources :**

* Off-the-shelf components

– Components are from a third party or were developed for a previous project

– Ready to use; fully validated and documented; virtually no risk

* Full-experience components

– Components are similar to the software that needs to be built

– Software team has full experience in the application area of these components

– Modification of components will incur relatively low risk

• Partial-experience components

– Components are related somehow to the software that needs to be built but will require substantial modification

– Software team has only limited experience in the application area of these components

– Modifications that are required have a fair degree of risk

• New components

– Components must be built from scratch by the software team specifically for the needs of the current project

– Software team has no practical experience in the application area

– Software development of components has a high degree of risk

**Environment Resources**

* A software engineering environment (SEE) incorporates hardware, software, and network resources that provide platforms and tools to develop and test software work products
* Most software organizations have many projects that require access to the SEE provided by the organization
* Planners must identify the time window required for hardware and software and verify that these resources will be available

**Empirical Estimation Models – COCOMO II :**

* **COCOMO-II** is the revised version of the [original Cocomo (Constructive Cost Model)](https://www.geeksforgeeks.org/?p=193526) and is developed at University of Southern California.
* It is the model that allows one to estimate the cost, effort and schedule when planning a new software development activity.
* COCOMO II is actually a hierarchy of estimation models that address the following areas:
* ***Application composition model.***Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
* ***Early design stage model.*** Used once requirements have been stabilized and basic software architecture has been established.
* ***Post-architecture-stage model.***Used during the construction of the software.
* Difference between COCOMO 1 and COCOMO 2
* [**COCOMO 1 Model**](https://www.geeksforgeeks.org/software-engineering-cocomo-model/)**:**The Constructive Cost Model was first developed by Barry W. Boehm. The model is for estimating effort, cost, and schedule for software projects. It is also called as Basic COCOMO. This model is used to give an approximate estimate of the various parameters of the project. Example of projects based on this model is business system, payroll management system and inventory management systems.
* [**COCOMO 2 Model**](https://www.geeksforgeeks.org/software-engineering-cocomo-ii-model/)**:**The COCOMO-II is the revised version of the original Cocomo (Constructive Cost Model) and is developed at the University of Southern California. This model calculates the development time and effort taken as the total of the estimates of all the individual subsystems. In this model, whole software is divided into different modules. Example of projects based on this model is Spreadsheets and report generator.

**Estimation for Agile Development :**

Estimation for agile projects uses a decomposition approach that encompasses the following steps:

**1.** Each user scenario (the equivalent of a mini use case created at the very start of a project by end users or other stakeholders) is considered separately for estimation purposes.

**2.** The scenario is decomposed into the set of software engineering tasks that will be required to develop it.

**3a.** The effort required for each task is estimated separately. Note: Estimation can be based on historical data, an empirical model, or “experience.”

**3b.** Alternatively, the “volume” of the scenario can be estimated in LOC, FP, or some other volume oriented measure (e.g., use-case count).

**4a.** Estimates for each task are summed to create an estimate for the scenario.

**4b.** Alternatively, the volume estimate for the scenario is translated into effort using historical data.

**5.** The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

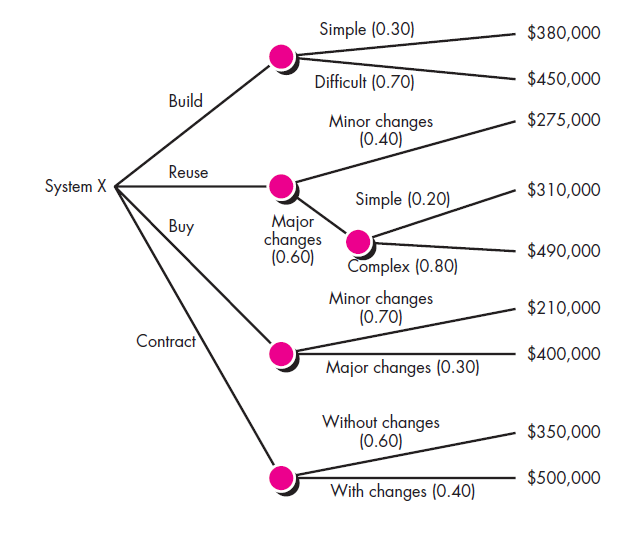
Because the project duration required for the development of a software increment is quite short (typically three to six weeks), this estimation approach serves two purposes:

1. to be certain that the number of scenarios to be included in the increment conforms to the available resources, and
2. to establish a basis for allocating effort as the increment is developed.

**THE MAKE/BUY DECISION :**

* A make-or-buy analysis is a general project management technique that is used to identify if a particular work can be accomplished by the project team or should just be purchased from external sources.
* The make-or-buy decision is the act of making a strategic choice between producing an item internally (in-house) or buying it externally (from an outside supplier). The buy side of the decision also is referred to as outsourcing.
* Make-or-buy decisions usually arise when a firm that has developed a product or part—or significantly modified a product or part—is having trouble with current suppliers, or has diminishing capacity or changing demand.
* Software engineering managers are faced with a make/ buy decision that can be further complicated by a number of acquisition options:
* (1) software may be purchased (or licensed) off-the-shelf,
* (2) “full-experience” or “partial-experience” software components may be acquired and then modified and integrated to meet specific needs, or
* (3) software may be custom built by an outside contractor to meet the purchaser’s specifications.
* The steps involved in Acquisition of Software are defined by the criticality of the Software to be Purchased and the end-cost.
* The make/buy decision can be made based on the following conditions :
* Will the software product be available sooner than internally developed software?
* Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
* Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support?

**Creating a Decision Tree :** A Decision Tree can be created by using a Statistical Technique to help the Management to make a Make or Buy Decision.



**Outsourcing:** Outsourcing concept is very simple. Software Engineering activities are contracted to a third party who does the work at Lower cost and hopefully High quality.

The decision to Outsource can be either Strategic or Tactical.

* At the Strategic level, Business Managers consider whether a significant portion of all software work can be contracted to other.
* At the Tactical level, Project Manager determine whether part or all of a Project can be best accomplished by subcontracting the Software work.

**Advantages of Outsourcing**

* 1. *Cost reduction*. Companies can secure competitive prices for contracted services, especially if the work can be outsourced offshore.
* 2. *Faster project completion*. Not only can work be done more cheaply, but it can also be done faster. Competitive pricing means more resources for the dollar. 3. *High level of expertise*. A high level of expertise and technology can be brought to bear on the project. A company no longer has to keep up with technological advances. Instead, it can focus on developing its core competencies and hire firms with the know-how to work on relevant segments of the project.
* 4. *Flexibility*. Organizations are no longer constrained by their own resources but can pursue a wide range of projects by combining their resources with talents of other companies.

**Disadvantages of Outsourcing**

* 1. *Coordination breakdowns*. Coordination of professionals from different organizations can be challenging, especially if the project work requires close collaboration and mutual adjustment. Breakdowns are exacerbated by physical separation with people working in different buildings, different cities, if not different countries.
* 2. *Loss of control*. There is potential loss of control over the project. The core team depends on other organizations that they have no direct authority over. While long-term survival of participating organizations depends on performance, a project may falter when one partner fails to deliver.
* 3. *Conflict*. Projects are more prone to interpersonal conflict since the different participants do not share the same values, priorities, and culture. Trust, which is essential to project success, can be difficult to forge when interactions are limited and people come from different organizations.
* 4. *Security issues*. Depending on the nature of the project, trade and business secrets may be revealed. This can be problematic if the contractor also works for your competitor. Confidentiality is another concern and companies have to be very careful when outsourcing processes like payroll, medical transcriptions, and insurance information.

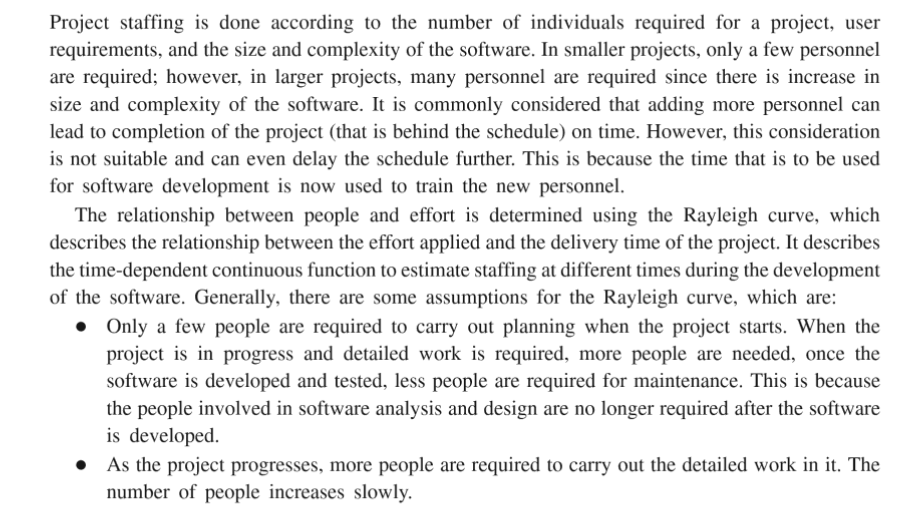
**Project Scheduling (Chpt – 3)**

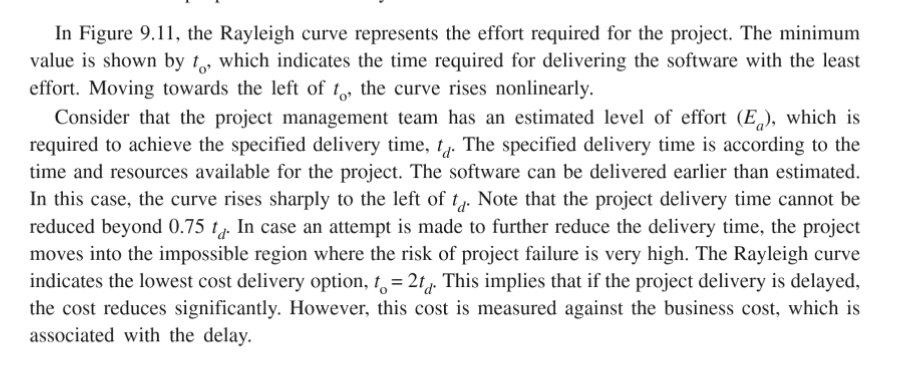
* Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity.
* It is a mechanism to communicate what tasks need to get done and which organizational resources will be allocated to complete those tasks in what timeframe.
* *Software project scheduling* is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.
* It is the tool that communicates what work needs to be performed, which resources of the organization will perform the work and the timeframes in which that work needs to be performed.
* The project schedule should reflect all of the work associated with delivering the project on time
* Scheduling in project management is the listing of activities, deliverables, and milestones within a project. A schedule also usually includes the planned start and finish date, duration, and resources assigned to each activity.  Effective project scheduling is a critical component of successful time management.
* In fact, when people discuss the processes for building a schedule, they are usually referring to the first six processes of time management:
* Plan schedule management.
* Define project activities.
* Sequence activities.
* Estimate resources.
* Estimate durations.
* Develop the project schedule.
* For scheduling a project, it is necessary to -
* Break down the project tasks into smaller, manageable form
* Find out various tasks and correlate them
* Estimate time frame required for each task
* Divide time into work-units
* Assign adequate number of work-units for each task
* Calculate total time required for the project from start to finish

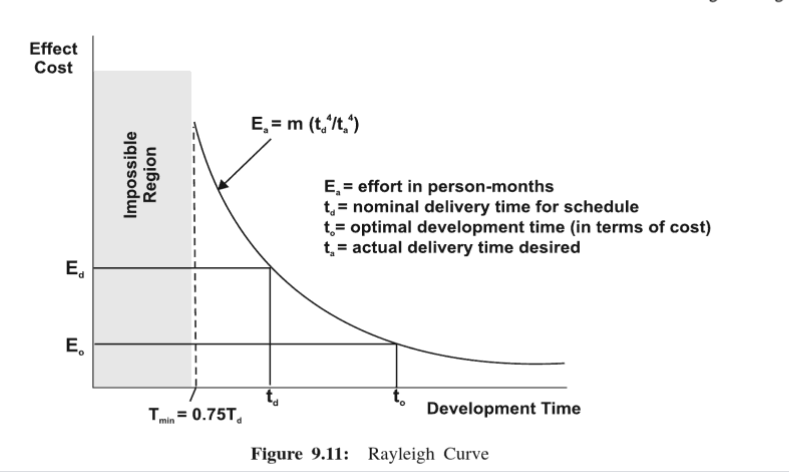
**Software Project Scheduling Principles :**

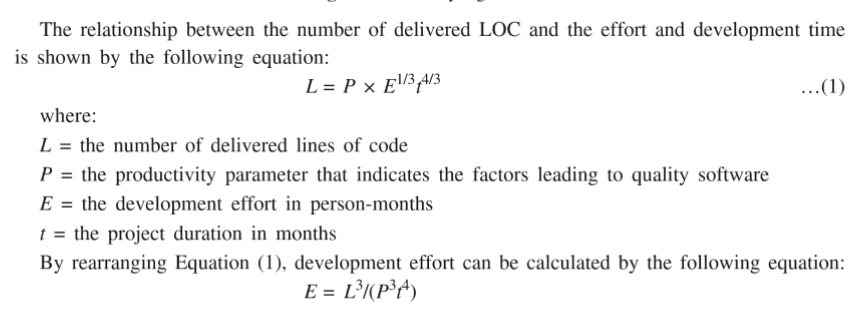
* Compartmentalization - the product and process must be decomposed into a manageable number of activities and tasks
* Interdependency - tasks that can be completed in parallel must be separated from those that must completed serially
* Time allocation - every task has start and completion dates that take the task interdependencies into account
* Effort validation - project manager must ensure that on any given day there are enough staff members assigned to completed the tasks within the time estimated in the project plan
* Defined Responsibilities - every scheduled task needs to be assigned to a specific team member
* Defined outcomes - every task in the schedule needs to have a defined outcome (usually a work product or deliverable)
* Defined milestones - a milestone is accomplished when one or more work products from an engineering task have passed quality review

**The Relationship Between People and Effort :**

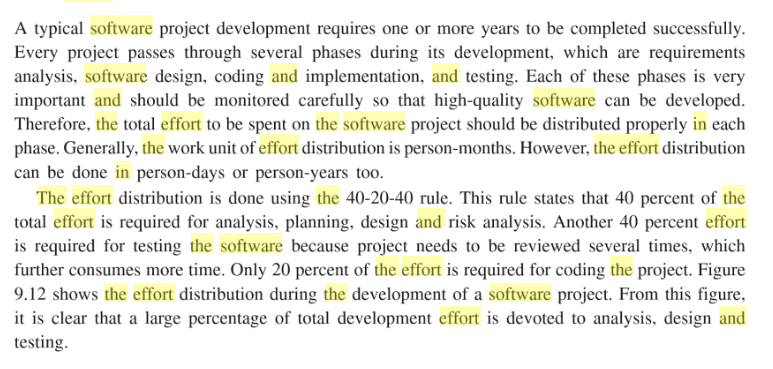


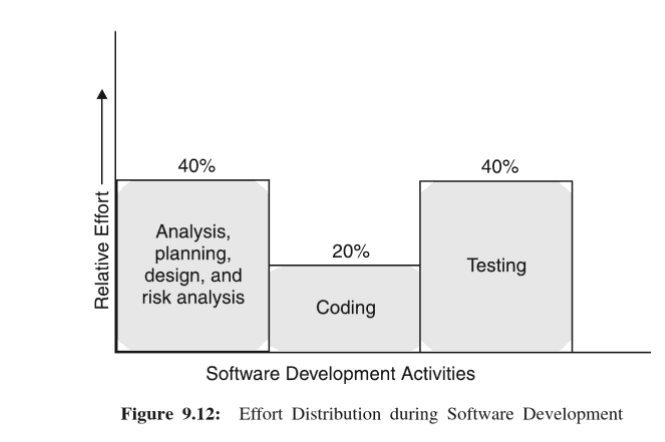
****

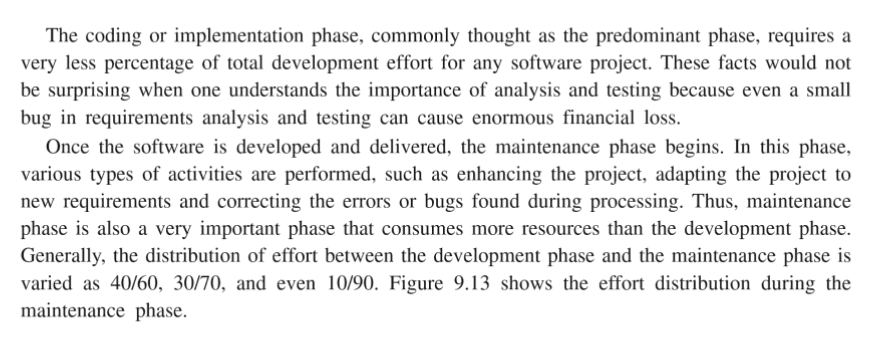
****

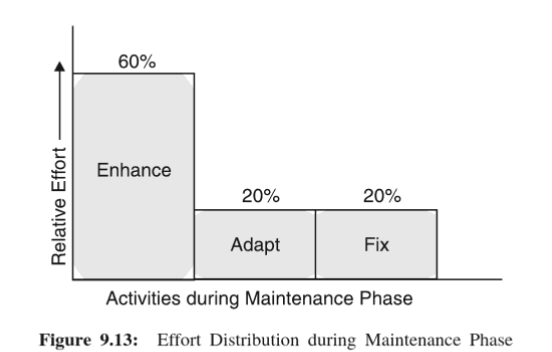
****

**Effort Distribution :**

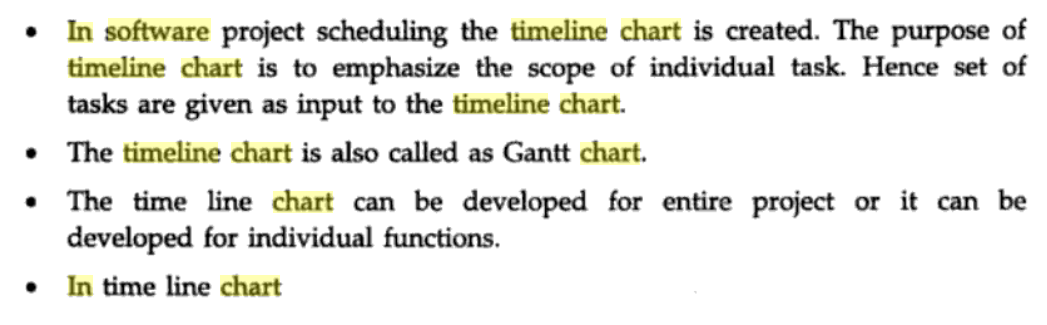


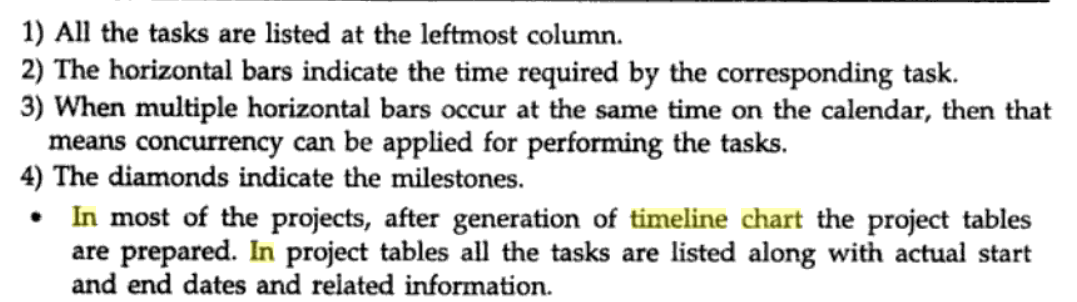


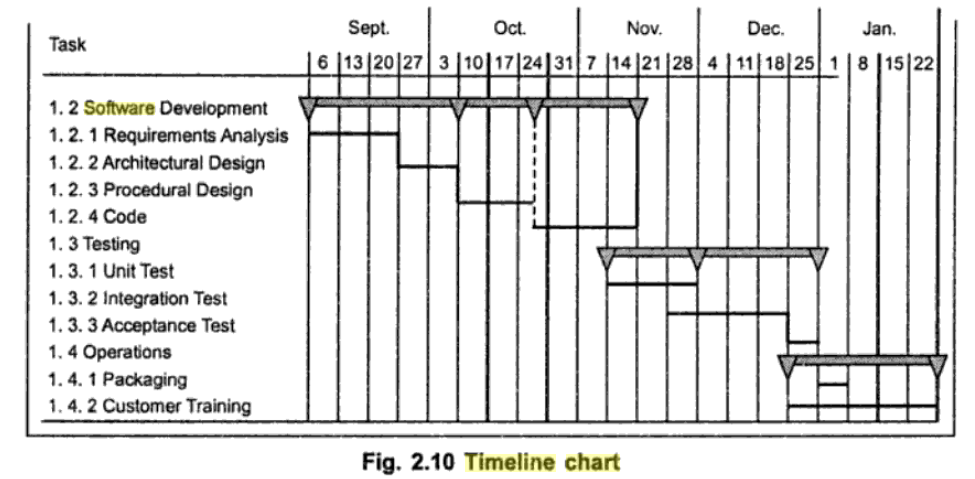




**TimeLine Chart :**

****

****

****