Unit-4

Planning a Software Project

SOFTWARE PROJECT ESTIMATION

- Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece.
- Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists.
- Software project estimation is a form of problem solving, and in most cases, the problem to be solved is too complex to be considered in one piece.
- For this reason, you should decompose the problem, characterizing it as a set of smaller(and hopefully, more manageable) problems

EFFORT ESTIMATION

In softwaredevelopment, **effort estimation** is the process of predicting the most realistic amount of effort (expressed in terms of person-hours or money) required to develop or maintain software based on incomplete, uncertain and noisy input. Effort estimates may be used as input to project plans, iteration plans, budgets, investment analyses, pricing processes and bidding rounds.

BUILDING EFFORT ESTIMATION MODEL

- An estimation model can be viewed as a "function" that outputs the effortestimate, clearly this estimation function will need inputs about the project,from which it can produce the estimate.
- The basic idea of having a model orprocedure for estimation is that it reduces the problem of estimation to estimatingor determining the value of the "key parameters" that characterize the project, based on which the effort can be estimated.
- One common approach therefore for estimating effort is to make it a function of *project size,* and the equation of effort is considered as

EFFORT = a^SIZE

Where a and b are constants [5], and project size is generally in KLOCor function points.

A Bottom-Up Estimation Approach

- In this approach, the major programs (or units or modules) in the software being built are first determined. Each program unit is then classified as simple, medium, or complex based on certain criteria. For each classification unit, an average effort for coding (and unit testing) is decided. This standard coding effort can be based on past data.
- The procedure for estimation can be summarized as the following sequence of steps:

1. Identify modules in the system and classify them as simple, medium, or complex.
2. Determine the average coding effort for simple/medium/complex modules.
3. Get the total coding effort using the coding effort of different types of modules and the counts for them.
4. Using the effort distribution for similar projects, estimate the effort for other tasks and the total effort.
5. Refine the estimates based on project-specific factors. miler project, from some guidelines, or some combination of these.

COCOMOModel(EmpiricalProcessModel)

- COCOMO is one of the most widely used software estimation models in the world.
- This model is developed in 1981 by Barry Boehm to give estimation of number of man-months it will take to develop a software product.
- COCOMO predicts the efforts and schedule of software product based on size of software.
- COCOMO has three different models that reflect complexity

1. Basic Model
2. Intermediate Model
3. Detailed Model

- Similarly, there are three classes of software projects.

1. Organic mode in this mode, relatively simple, small software projects with a small team is handled. Such team should have good application experience to less rigid requirements.

2. Semi-detached projects in this class intermediate project in which team with mixed experience level are handled. Such project may have mix of rigid and less than rigid requirements.
3. Embedded projects in this class, project with tight hardware, software and operational constraints are handled.

- Each Model in detail

1. Basic Model

The basic COCOMO model estimate the software development effort using only Lines of code

Various equations in this model are

E= abKLOCbbD = cbEdbE = abKLOCbbD = cbEbd

Where, E is the effort applied in person-months,

D is the development time in chronological months and

KLOC is the estimated number of delivered lines of code for the project

2. Intermediate Model

This is extension of COCOMO model.

This estimation model makes use of set of "Cost Driver Attributes" to compute the cost of software.

I. Product attributes

a. required software reliability

b. size of application data base

c. complexity of the product

II. Hardware attributes

a. run-time performance constraints

b. memory constraints

c. volatility of the virtual machine environment

d. required turnaround time

III. Personnel attributes

a. analyst capability

b. software engineer capability

c. applications experience

d. virtual machine experience

e. programming language experience

IV. Project attributes

a. use of software tools

b. application of software engineering methods

c. required development schedule

Each of the 15 attributes is rated on a 6 point scale that ranges from "very low" to "extra high" (in importance or value).

The intermediate COCOMO model takes the form

$$E=aiKLOCbi\times EAFE=aiKLOCib\times EAF$$

Where, E is the effort applied in person-months and

KLOC is the estimated number of delivered lines of code for the project

3. Detailed COCOMO Model

The detailed model uses the same equation for estimation as the intermediate Model.

But detailed model can estimate the effort (E), duration (D), and person (P) of each of development phases, subsystem and models.

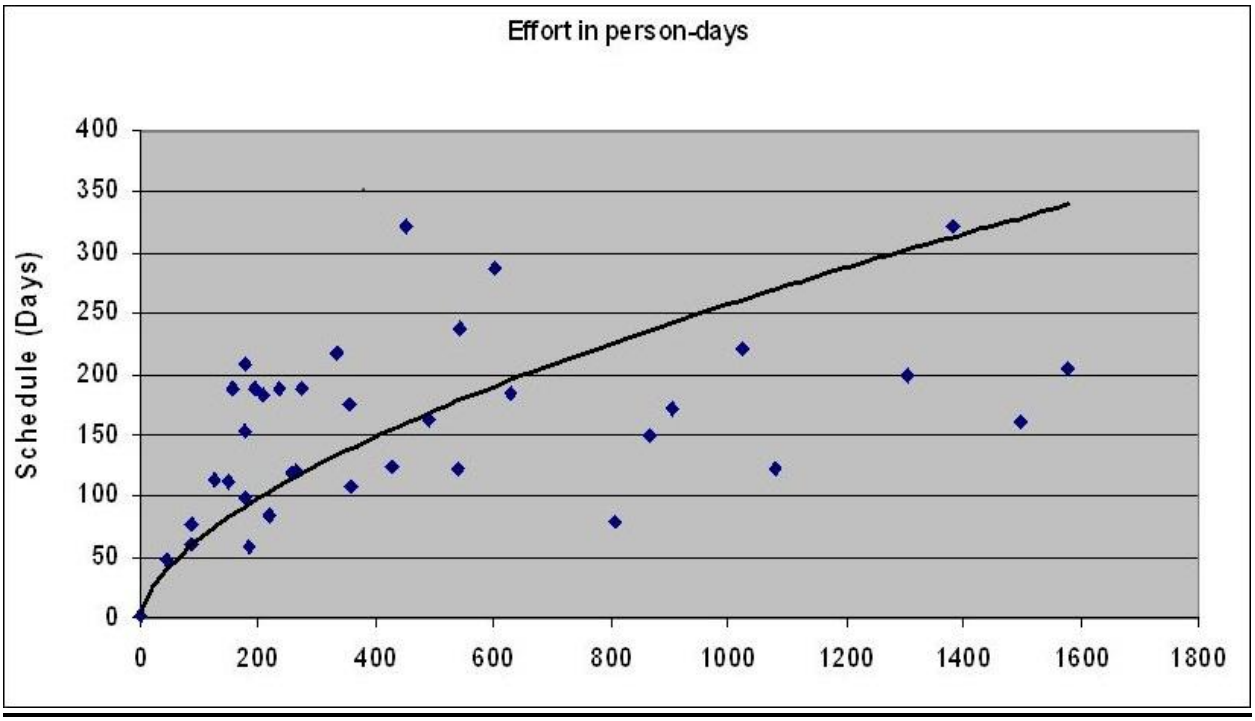
PROJECT SCHEDULE

- A project Schedule is at two levels - overall schedule and detailed schedule
- Overall schedule comprises of major milestones and final date
- Detailed schedule is the assignment of lowest level tasks to resources

OVERALL SCHEDULE

- Depends heavily on the effort estimate
- For an effort estimate, some flexibility exists depending on resources assigned
- Eg a 56 person-months project can be done in 8 months with 7 people, or 7 months with 8 people
- Stretching a schedule is easy; compressing is hard and expensive
- One method is to estimate schedule S (in months) as a function of effort in PMs
- Can determine the fn through analysis of past data; the function is non linear
- COCOMO: S = 2.5 E 3.8
- Often this schedule is checked and corrected for the specific project
- One checking method – square root check

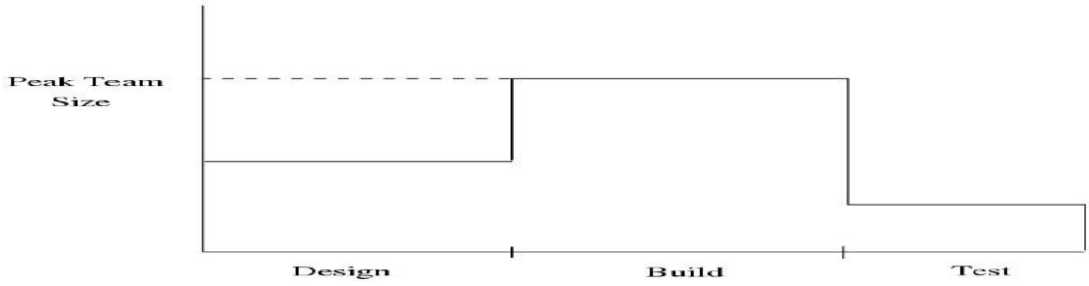
DETERMINING OVERALL SCHEDULE FROM PAST DATA



DETERMINING MILESTONES

- With effort and overall schedule decided, avg project resources are fixed
- Manpower ramp-up in a project decides the milestones
- Manpower ramp-up in a project follows a Rayleigh curve - like a normal curve
- In reality manpower build-up is a step function

MANPOWER RAMP-UP



MILESTONES

- With manpower ramp-up and effort distribution, milestones can be decided
- Effort distribution and schedule distribution in phases are different
- Generally, the build has larger effort but not correspondingly large schedule
- COCOMO specifies overall scheduling. Design – 19%, programming – 62%, integration – 18%

An Example Schedule

Task	Dur. (days)	Work (p-days)	Start Date	End Date
Project Init tasks	33	24	5/4	6/23
Training	95	49	5/8	9/29
Knowledge sharing	78	20	6/2	9/30
Elaboration iteration I	55	55	5/15	6/23
Construction iteration I	9	35	7/10	7/21

DETAILED SCHEDULING

- To reach a milestone, many tasks have to be performed
- Lowest level tasks - those that can be done by a person (in less than 2-3 days)
- Scheduling - decide the tasks, assign them while preserving high-level schedule
- Is an iterative task - if cannot "fit" all tasks, must revisit high level schedule
- Detailed schedule not done completely in the start - it evolves
- Can use Microsoft Project for keeping it
- Detailed Schedule is the most live document for managing the project
- Any activity to be done must get reflected in the detailed schedule

An example task in detail schedule

Module	Act Code	Task	Duration	Effort
History	PUT	Unit test # 17	1 day	7 hrs
St. date	**End date**	**%comp**	**Depend.**	**Resource**
7/18	7/18	0%	Nil	SB

- Each task has name, date, duration, resource etc assigned
- % done is for tracking (tools use it)
- The detailed schedule has to be consistent with milestones
- ➢ Tasks are sub-activities of milestone level activities, so effort should add up, total schedule should be preserved

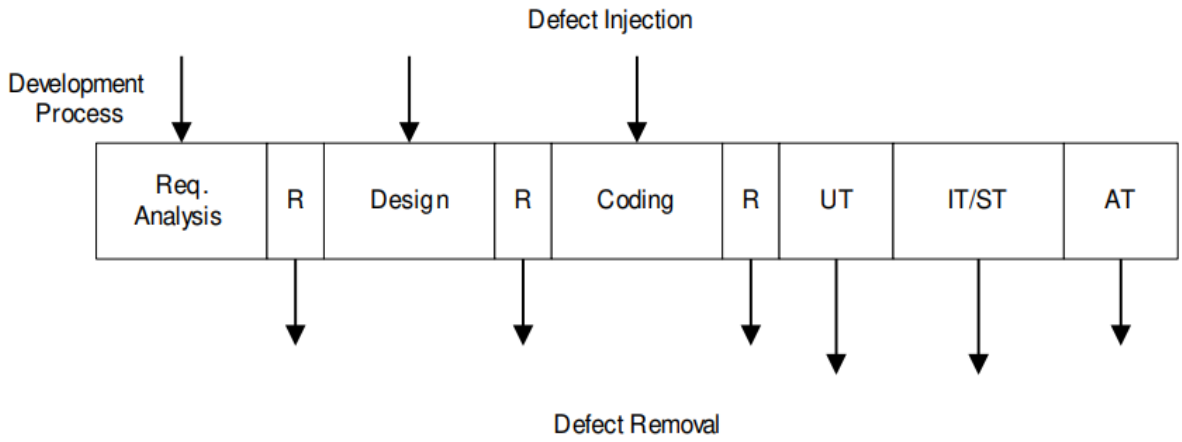
QUALITY PLANNING:

- Delivering high quality is a basic goal
- Quality can be defined in many ways
- Current industry standard - delivered defect density (e.g. #defects/KLOC)

- Defect - something that causes software to behave in an inconsistent manner
- Aim of a project - deliver software with low delivered defect density

DEFECT INJECTION AND REMOVAL

- Software development is labor intensive
- Defects are injected at any stage
- As quality goal is low delivered defect density, these defects have to be removed
- Done primarily by quality control (QC) activities of reviews and testing



APPROACHES TO QUALITY MANAGEMENT

- Ad hoc - some testing, some reviews done as and when needed
- Procedural - defined procedures are followed in a project
- Quantitative - defect data analysis done to manage the quality process

PROCEDURAL APPROACH

- A quality plan defines what QC tasks will be undertaken and when
- Main QC tasks - reviews and testing
- Guidelines and procedures for reviews and testing are provided
- During project execution, adherence to the plan and procedures ensured

QUANTITATIVE APPROACH

- Goes beyond asking “has the procedure been executed”

- Analyzes defect data to make judgments about quality
- Past data is very important
- Key parameters - defect injection and removal rates, defect removal efficiency (DRE)

QUALITY PLAN

- The quality plan drives the quality activities in the project
- Level of plan depends on models available
- Must define QC tasks that have to be performed in the project
- Can specify defect levels for each QC tasks (if models and data available)

SOFTWARE RISK MANAGEMENT:

Risk is an expectation of loss, a potential problem that may or may not occur in the future. It is generally caused due to lack of information, control or time.A possibility of suffering from loss in software development process is called a software risk. Loss can be anything, increase in production cost, development of poor quality software, not being able to complete the project on time. A software risk can be of two types : internal risks that are within the control of the project manager and external risks that are beyond the control of project manager. Risk management is carried out to:

Identify the risk

Reduce the impact of risk

Reduce the probability or likelihood of risk

Risk monitoring

A project manager has to deal with risks arising from three possible cases:

Known knowns are software risks that are actually facts known to the team as well as to the entire project. For example not having enough number of developers can delay the project delivery. Such risks are described and included in the Project Management Plan.

Known unknowns are risks that the project team is aware of but it is unknown that such risk exists in the project or not. For example if the communication with the client is not of good level then it is not possible to capture the requirement properly. This is a fact known to the project team however whether the client has communicated all the information properly or not is

unknown to the project.

Unknown Unknowns are those kind of risks about which the organization has no idea. Such risks are generally related to technology such as working with technologies or tools that you have no idea about because your client wants you to work that way suddenly exposes you to absolutely unknown risks.

Software risk management is all about risk quantification of risk. This includes:

Giving a precise description of risk event that can occur in the project

Defining risk probability that would explain what are the chances for that risk to occur

Defining How much loss a particular risk can cause

Defining the liability potential of risk

Risk Management comprises of following processes:

Software Risk Identification

Software Risk Analysis

Software Risk Planning

Software Risk Monitoring

These Processes are defined below.

SOFTWARE RISK IDENTIFICATION

In order to identify the risks that your project may be subjected to, it is important to first study the problems faced by previous projects. Study the project plan properly and check for all the possible areas that are vulnerable to some or the other type of risks. The best ways of analyzing a project plan is by converting it to a flowchart and examine all essential areas. Any decision taken related to technical, operational, political, legal, social, internal or external factors should be evaluated properly. In this phase of Risk management you have to define processes that are important for risk identification.

SOFTWARE RISK ANALYSIS

Software Risk analysis is a very important aspect of risk management. In this phase the risk is identified and then categorized. After the categorization of risk, the level, likelihood (percentage) and impact of the risk is analyzed. Likelihood is defined in percentage after examining what are the chances of risk to occur due to various technical conditions. These technical conditions can be:

Complexity of the technology

Technical knowledge possessed by the testing team

Conflicts within the team

Teams being distributed over a large geographical area

Usage of poor quality testing tools

With impact we mean the consequence of a risk in case it happens. It is important to know about the impact because it is necessary to know how a business can get affected:

What will be the loss to the customer

How would the business suffer

Loss of reputation or harm to society

Monetary losses

Legal actions against the company

Cancellation of business license

Level of risk is identified with the help of:

QUALITATIVE RISK ANALYSIS: Here you define risk as:

High

Low

Medium

QUANTITATIVE RISK ANALYSIS: can be used for software risk analysis but is considered inappropriate because risk level is defined in % which does not give a very clear picture.

SOFTWARE RISK PLANNING:

Software risk planning is all about:

Defining preventive measure that would lower down the likelihood or probability of various risks.

Define measures that would reduce the impact in case a risk happens.

Constant monitoring of processes to identify risks as early as possible

SOFTWARE RISK MONITORING

Software risk monitoring is integrated into project activities and regular checks are conducted on top risks. Software risk monitoring comprises of:

Tracking of risk plans for any major changes in actual plan, attribute, etc.

Preparation of status reports for project management.

Review risks and risks whose impact or likelihood has reached the lowest possible level should be closed.

Regularly search for new risks

PROJECT MONITORING PLAN:

There are two considerations for monitoring a project. They are

1. Measurements
2. Project Execution & Monitoring

MEASUREMENTS:

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

- **Software size estimation**

  Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and Function points vary according to the user or software requirement.

- **Effort estimation**

  The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.

- **Time estimation**

  Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on

day-to-day basis or in calendar months.

The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.

- **Cost estimation**

  This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider –

  - Size of software
  - Software quality
  - Hardware
  - Additional software or tools, licenses etc.
  - Skilled personnel with task-specific skills
  - Training and support

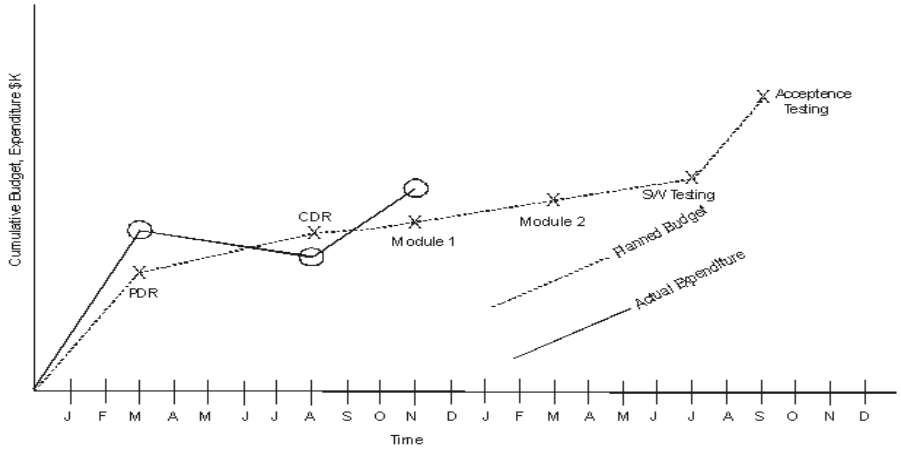
PROJECT EXECUTION & MONITORING:

In this phase, the tasks described in project plans are executed according to their schedules.Execution needs monitoring in order to check whether everything is going according to the plan. Monitoring is observing to check the probability of risk and taking measures to address the risk or report the status of various tasks.

These measures include -

- **Activity Monitoring -** All activities scheduled within some task can be monitored on day-to-day basis. When all activities in a task are completed, it is considered as complete.
- **Status Reports -** The reports contain status of activities and tasks completed within a given time frame, generally a week. Status can be marked as finished, pending or work-in-progress etc.
- **Milestones Checklist -** Every project is divided into multiple phases where major tasks are performed (milestones) based on the phases of SDLC. This milestone checklist is prepared once every few weeks and reports the status of milestones.

A graphical method of capturing the basic progress of a project as compared to its plans is the cost-schedule-milestone graph. The X-axis of this graph is time, where the months in the project schedule are marked. The y-axis represents the cost, in dollars or PMs. Twocurves are drawn. One curve is the planned cost and planned schedule, inwhich each important milestone of the project is marked. This curve canbe completed after the project plan is made. The second curve represents the actual cost and actual schedule, and the actual achievement of the milestones is marked. Thus, for each milestone the point representing the time.when the milestone is actually achieved and the actual cost of achieving itare marked. A cost-schedule-milestone graph for the example is shown in



In the above graph,a hypothetical project whose cost is estimated to be $100K is considered .The milestones in this project are PDR (preliminary design review), CDR (critical design review), Module 1 completion,Module 2 completion, integration testing, and acceptance testing.The planned budget is shown by a dotted fine. The actual expenditure is shown with a bold line. This chart shows that only two milestones have beenachieved, PDR and CDR, and though the project was within budget whenPDR was complete, it is now slightly over budget.

DESIGN

The design activity begins when the requirements document for the software tobe developed is available and the architecture has been designed. During design we further refine the architecture. The design of a system is essentially a blueprint or a plan for a solution for the system.

The design process for software systems often has two levels.

1. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what may be called the module design or the high-level design.
2. In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided.

A *design methodology* is a systematic approach to creating a design by applying of a set of techniques and guidelines.

DESIGN CONCEPTS

Design is correct, if it will satisfy all therequirements and is consistent with architecture.Of the correct designs, we want best design. We focus on modularity as the maincriteria (besides correctness).

Modularity

Modular system is a system in which modules can be builtseparately and changes in one have minimum impact on others. Modularity supports independence of models, enhances design clarity, easesimplementation, reduces cost of testing, debugging andmaintenance.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

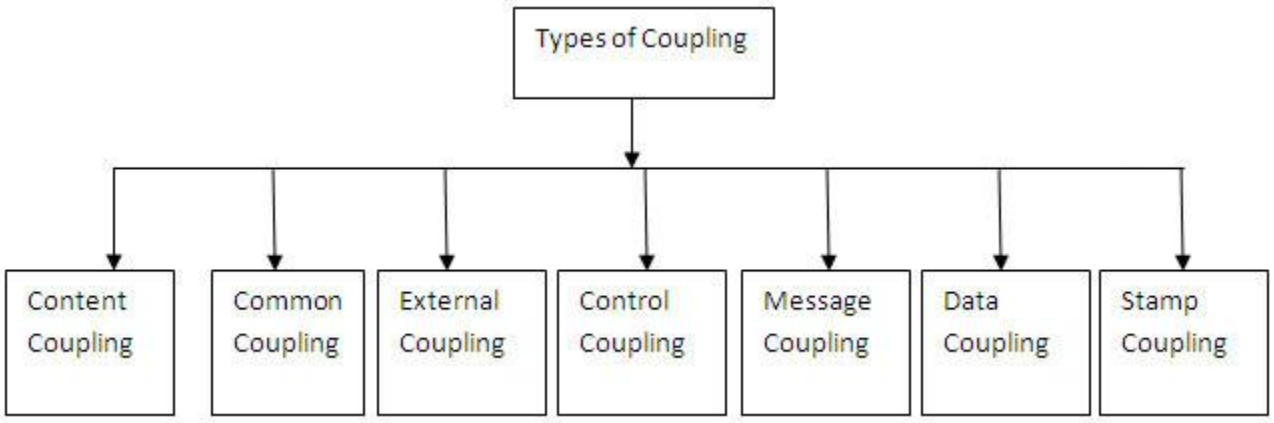
- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again

- Concurrent execution can be made possible
- Desired from security aspect

COUPLING AND COHESION

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Coupling:- Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.



There are five levels of coupling are:-

• ***Content coupling*** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

• ***Common coupling***- When multiple modules have read and write access to some global data, it is called common or global coupling.

• ***Control coupling***- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

• ***External Coupling***: This type of coupling occurs when an external imposed data format and communication protocol are shared by two modules. External Coupling is generally related to the communication to external devices.

• ***Message Coupling***: This type of coupling can be achieved by the state decentralization. It is the loosest type of coupling, in which the component communication is performed through message passing.

• ***Stamp coupling***- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

• ***Data coupling***- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Factors affecting coupling:

	Interface Complexity	Type of Connection	Type of Communication
Low	Simple obvious	To module by name	Data
High	Complicated obscure	To internal elements	Control Hybrid

The manifestation of coupling in OO systems is somewhat different as objects are semantically richer than functions. In OO systems, three different types of coupling exist between modules

1. Interaction coupling
2. Component coupling
3. Inheritance coupling

Interaction coupling occurs due to methods of a class invoking methods ofother classes. In many ways, this situation is similar to a function calling anotherfunction and hence this coupling is similar to coupling between functionalmodules discussed above. Like with functions, the worst form of coupling here isif methods directly access internal parts of other methods. Coupling is lowestif methods communicate directly through parameters.

Within this category,as discussed above, coupling is lower if only data is passed, but is higher ifcontrol information is passed since the invoked method impacts the executionsequence in the calling method.Also, coupling is higher if the amount of databeing passed is increased. Similarly, ifan object is passed to a method when only some of its component objects are used within the method, coupling increases unnecessarily.

The least couplingsituation therefore is when communication is with parameters only, with onlynecessary variables being passed, and these parameters only pass data.

Component coupling refers to the interaction between two classes where aclass has variables of the other class. Three clear situations exist as to how thiscan happen.

A class C can be component coupled with another class C1, if Chas an instance variable of type C1, or C has a method whose parameter is oftype C1, or if C has a method which has a local variable of type C1. Note thatwhen C is component coupled with C1, it has the potential of being componentcoupled with all subclasses of C1 as at runtime an object of any subclass may actually be used. It should be clear that whenever there is component coupling,there is likely to be interaction coupling.

Component coupling is considered tobe weakest (i.e. most desired) if in a class C, the variables of class C1 are eitherin the signatures of the methods of C, or are some attributes of C. If interactionis through local variables, then this interaction is not visible from outside, and therefore increases coupling.

Inheritance coupling is due to the inheritance relationship between classes.

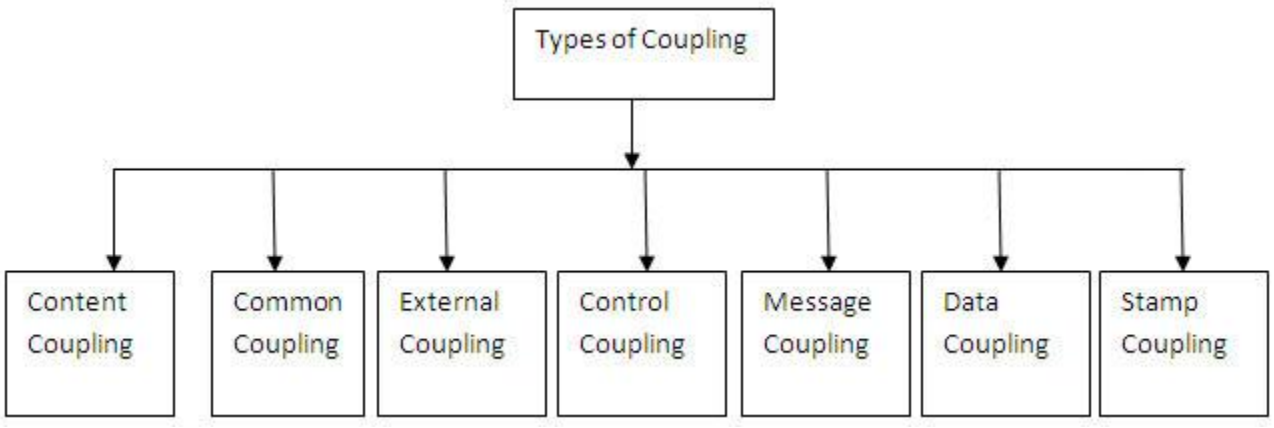
Two classes are considered inheritance coupled if one class is a direct or indirectsubclass of the other. If inheritance adds coupling, one can ask the questionwhy not do away with inheritance altogether. The reason is that inheritancemay reduce the overall coupling in the system. Let us consider two situations.If a class A is coupled with another class B, and if B is a hierarchy with B1 andDesignB2 as two subclasses, then if a method m() is factored out of B1 and B2 and put in the superclass B, the coupling drops as A is now only coupled with B,whereas earlier it

was coupled with both B1 and B2. Similarly, if B is a classhierarchy which supports specialization-generalization relationship, then if newsubclasses are added to B, no changes need to be made to a class A which callsmethods in B. That is, for changing B's hierarchy, A need not be disturbed. Without this hierarchy, changes in B would most likely result in changes in A Within inheritance coupling there are some situations that are worse thanothers. The worst form is when a subclass B1 modifies the signature of a methodin B (or deletes the method). This situation can easily lead to a runtime error,besides violating the true spirit of the is-a relationship. The least coupling scenario is when a subclass only adds instancevariables and methods but does not modify any inherited ones.

Cohesion:-

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear.

Types of cohesion:- 1. Coincidental cohesion 2.Logical cohesion 3.Temporal cohesion 4.Communicational cohesion 5. Sequential cohesion



Coincidental cohesion:-

This occurs when there is no relationship among elements of a module. They can occur if an

existing program modularized by chopping it into pieces and making differ piece of modules i.e. it performs a set of tasks that are related to each other very loosely. The modules contain a random collection of function.

Logical cohesion:-

A module having logical cohesion if there are some logical relationship between elements of a modules i.e. elements of a module performs the operation.

Temporal cohesion:-

It is same as logical cohesion except that the element must be executed in same time. Set of function responsible for initialization, startup, the shutdown of the same process. It is higher than logical cohesion since all elements are executed together. This avoids the problem of passing the flag.

Communicational cohesion:-

A module is said to have Communicational cohesion if all function of module refers to an update the same data structure.

Sequential cohesion:-

A module is said to have sequential cohesion if element module from different parts of the sequence. When the output from one element of the sequence is input to the next element of a sequence. A sequence bounded module may contain several functions or part of different functions.

Functional cohesion:-

It is the strongest cohesion in a functional bound module, all elements of the module are related to performing a single function. By function we not mean simply mathematical function but also these modules which have single goal function like computing square root and sort array are a clear example of functionality cohesion modules.

Cohesion in object-oriented systems has three aspects

1. Method cohesion
2. Class cohesion
3. Inheritance cohesion

Method cohesion is the same as cohesion in functional modules. It focuses onwhy the different code elements of a method are together within the method.

Class cohesion focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept orabstraction with all elements contributing toward supporting this concept.

Inheritance cohesion focuses on the reason why classes are together in ahierarchy

Open-closed Principle

Besides cohesion and coupling, open closed principlealso helps in achieving modularity.

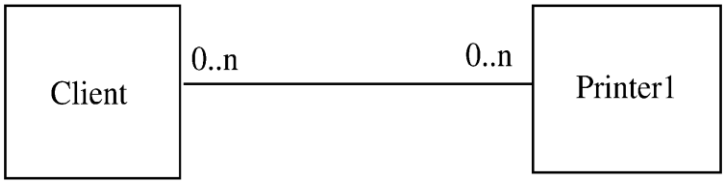
Principle: A module should be open for extension butclosed for modification.

Behavior can be extended to accommodate new requirements,but existing code is not modified i.e., allows addition of code, but not modification of existingcode.It minimizes risk of having existing functionality stop working dueto changes.

In ObjectOriented this principle is satisfied by usinginheritance and polymorphism.

Inheritance allows creating a new class toextend behavior without changing the original class.This can be used to support the open-closedprinciple.

Consider example of a client object whichinteracts with a printer object for printing



Client directly calls methods on Printer1.

If another printer is to be allowed.

A new class Printer2 will be created.

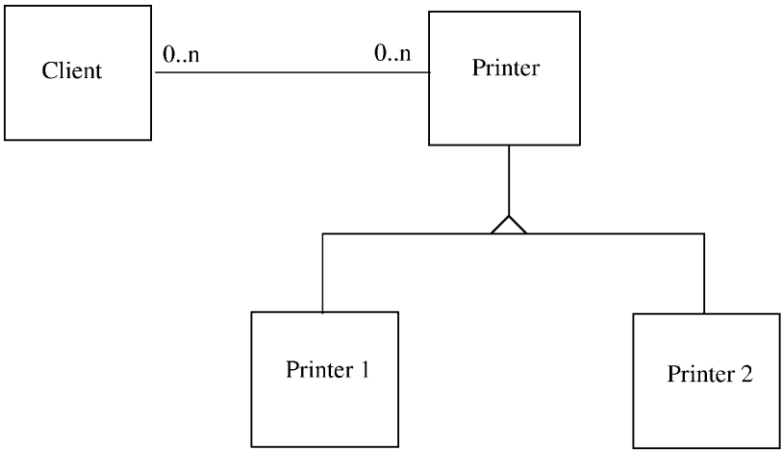
But the client will have to be changed if it wants touse Printer 2.

Alternative approach:

Have Printer1 a subclass of a general Printer.

For modification, add another subclass Printer 2.

Client does not need to be changed.



FUNCTION ORIENTED DESIGN

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

OBJECT ORIENTED DESIGN

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

DETAILED DESIGN

1. The process of refining and expanding the preliminary design phase of a system or component to the extent that the design is sufficiently complete to be implemented .

2. The result of the process in 1.

- To keep terminology consistent, we'll use the following definition:

1. The process of refining and expanding the software architecture of a system or component to the extent that the design is sufficiently complete to be implemented .

2. The result of the process in 1.

- During Detailed Design designers go deep into each component to define its internal structure and behavioral capabilities, and the resulting design leads to natural and efficient construction of software.

WHAT IS DETAILED DESIGN?

- "Architecture is design, but not all design is architecture. That is, many design decisions are left unbound by the architecture and are happily left to the discretion and good judgment of downstream designers and implementers. The architecture establishes constraints on downstream activities, and those activities must produce artifacts—finer-grained design and code—that are compliant with the architecture, but architecture does not define an implementation."
- Detailed design is closely related to architecture and construction; therefore successful designers (during detailed design) are required to have or acquire full understanding of the system's requirements and architecture. ü They must also be proficient in particular design strategies (e.g., objectoriented), programming languages, and methods and processes for software quality control. ü Just as architecture provides the bridge between requirements and design, detailed design provides the bridge between design and code.

There are many aspects to consider in the design of a piece of software. The importance of each

should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
- **Mod modification with slight or no modification.**
- **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security** - The software is able to withstand hostile acts and influences.
- **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.

KEY TASKS IN DETAILED DESIGN

- In practice, it can be argued that the detailed design phase is where most of the problemsolving activities occur. Consider the case in which a formal process is followed, so that the requirements is followed by architecture and detailed design. ü In many practical applications, the architectural design activity defers complex problem solving to detailed design, mainly through abstraction. ü In some cases, even specifying requirements is deferred to detailed design!
- For these reasons, detailed design serves as the gatekeeper for ensuring that the system’s specification and design are sufficiently complete before construction begins. ü This can be especially tough for large-scale systems built from scratch without experience with the development of similar systems.
- The major tasks identified for carrying out the detailed design activity include:
    1. Understanding the architecture and requirements
    2. Creating detailed designs
    3. Evaluating detailed designs

4. Documenting software design
5. Monitoring and controlling implementation.

SOFTWARE VERIFICATION AND VALIDATION

Verification and validation are not the same thing, although they are often confused. Boehm succinctly expressed the difference as

- Validation: Are we building the right product?
- Verification: Are we building the product right?

Building the right product implies creating a Requirements Specification that contains the needs and goals of the stakeholders of the software product. If such artifact is incomplete or wrong, the developers will not be able to build the product the stakeholders want. This is a form of "artifact or specification validation".

Building the product right implies the use of the Requirements Specification as input for the next phase of the development process, the design process, the output of which is the Design Specification. Then, it also implies the use of the Design Specification to feed the construction process. Every time the output of a process correctly implements its input specification, the software product is one step closer to final verification. If the output of a process is incorrect, the developers are not building the product the stakeholders want correctly. This kind of verification is called "artifact or specification verification".

Software verification

It would imply to verify if the specifications are met by running the software but this is not possible (e. g., how can anyone know if the architecture/design/etc. are correctly implemented by running the software?). Only by reviewing its associated artifacts, someone can conclude if the specifications are met.

Artifact or specification verification

The output of each software development process stage can also be subject to verification when checked against its input specification (see the definition by CMMI below).

Examples of artifact verification:

- Of the design specification against the requirement specification: Do the architectural design, detailed design and database logical model specifications correctly implement the functional and non-functional requirement specifications?
- Of the construction artifacts against the design specification: Do the source code, user interfaces and database physical model correctly implement the design specification?

VERIFICATION VS VALIDATION

According to the Capability Maturity Model

- Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.
- Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation during the software development process can be seen as a form of User Requirements Specification validation; and, that at the end of the development process is equivalent to Internal and/or External Software validation. Verification, from CMMI's point of view, is evidently of the artifact kind.

In other words, software verification ensures that the output of each phase of the software development process effectively carry out what its corresponding input artifact specifies (requirement -> design -> software product), while software validation ensures that the software product meets the needs of all the stakeholders (therefore, the requirement specification was correctly and accurately expressed in the first place). Software verification ensures that "you built it right" and confirms that the product, as provided, fulfills the plans of the developers. Software validation ensures that "you built the right thing" and confirms that the product, as provided, fulfills the intended use and goals of the stakeholders.

This article has used the strict or narrow definition of verification.

From testing perspective:

- Fault – wrong or missing function in the code.
- Failure – the manifestation of a fault during execution. The software was not effective. It does not do "what" it is supposed to do.

- Malfunction – according to its specification the system does not meet its specified functionality. The software was not efficient (it took too many resources such as CPU cycles, it used too much memory, performed too many I/O operations, etc.), it was not usable, it was not reliable, etc. It does not do something "how" it is supposed to do it.

SOFTWARE METRICS

A software metric is a measure of software characteristics which are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance,planning work items, measuring productivity, and many other uses.

Within the software development process, there are many metrics that are all related to each other. Software metrics are related to the four functions of management: Planning, Organization, Control, or Improvement**.**

BENEFITS OF SOFTWARE METRICS

The goal of tracking and analyzing software metrics is to determine the quality of the current product or process, improve that quality and predict the quality once the software development project is complete. On a more granular level, software development managers are trying to:

- Increase return on investment (ROI)
- Identify areas of improvement
- Manage workloads
- Reduce overtime
- Reduce costs

These goals can be achieved by providing information and clarity throughout the organization about complex software development projects. Metrics are an important component of quality assurance, management, debugging, performance, and estimating costs, and they're valuable for both developers and development team leaders:

- Managers can use software metrics to identify, prioritize, track and communicate any issues to foster better team productivity. This enables effective management and allows assessment and

prioritization of problems within software development projects. The sooner managers can detect software problems, the easier and less-expensive the troubleshooting process.

- Software development teams can use software metrics to communicate the status of software development projects, pinpoint and address issues, and monitor, improve on, and better manage their workflow.

Software metrics offer an assessment of the impact of decisions made during software development projects. This helps managers assess and prioritize objectives and performance goals.

Although many software metrics have been proposed over a period of time, ideal software metric is the one which is easy to understand, effective, and efficient. In order to develop ideal metrics, software metrics should be validated and characterized effectively. For this, it is important todevelop metrics using some specific guidelines, which are listed below.

- **Simple and computable:** Derivation of software metrics should be easy to learn and should involve average amount of time and effort.
- **Consistent and objective:** Unambiguous results should be delivered by software metrics.
- **Consistent in the use of units and dimensions:** Mathematical computation of the metrics should involve use of dimensions and units in a consistent manner.
- **Programming language independent:** Metrics should be developed on the basis of the analysis model, design model, or program's structure.
- **High quality:** Effective software metrics should lead to a high-quality software product.
- **Easy to calibrate:** Metrics should be easy to adapt according to project requirements.
- **Easy to obtain:** Metrics should be developed at a reasonable cost.
- **Validation:** Metrics should be validated before being used for making any decisions.
- **Robust:** Metrics should be relatively insensitive to small changes in process, project, or product.
- **Value:** Value of metrics should increase or decrease with the value of the software characteristics they represent. For this, the value of metrics should be within a meaningful range. For example, metrics can be in a range of 0 to 5.