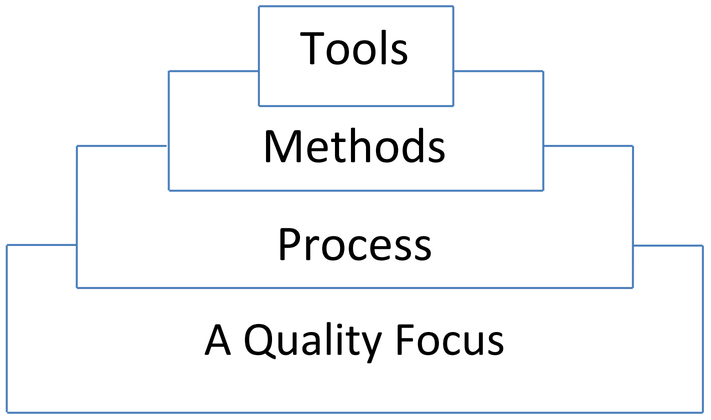
Software Engineering

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. [Fritz Bauer, IEEE Standard 610.12-1990, IEEE 1993]

Software engineering is the application of a systemic disciplined quantifiable approach to the development, operation and maintenance of software, i.e. the application of engineering to software.

Software Engineering is a layered technology.

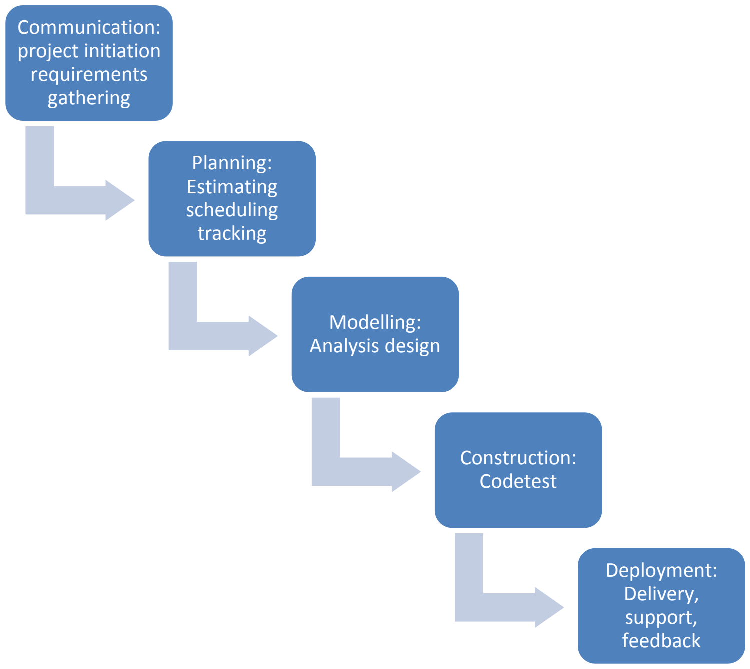


The foundation for software engineering is the process layer. Process defines a framework that must be established for effective delivery of software engineering technology.

Software engineering methods provides the technical “how to”-s for building software. Methods encompass a broad array of tasks that include communication, requirement analysis, design modeling, program construction, testing and support.

Software engineering tools provide automated or semi-automated support for the process and methods.

The Waterfall Model

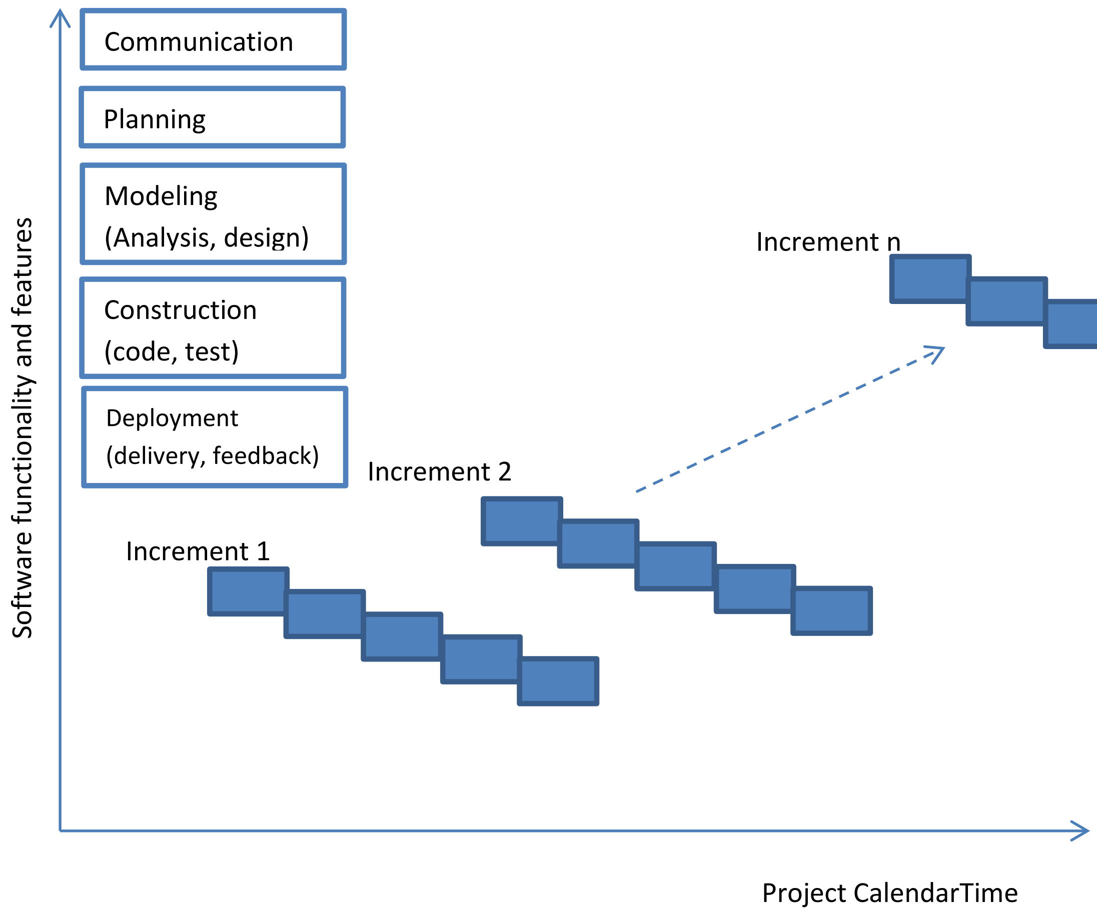

Communication: project initiation requirements gathering
Planning: Estimating scheduling tracking
Modelling: Analysis design
Construction: Codetest
Deployment: Delivery, support, feedback

The waterfall model suggest a systematic sequential approach to software development.

Drawbacks:

1. Real projects rarely follow the sequential flow that the model proposes although the linear model can accommodate iteration, it does so indirectly and as a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all the requirements explicitly. The waterfall model requires this and has difficultly accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience, a working version of the program will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

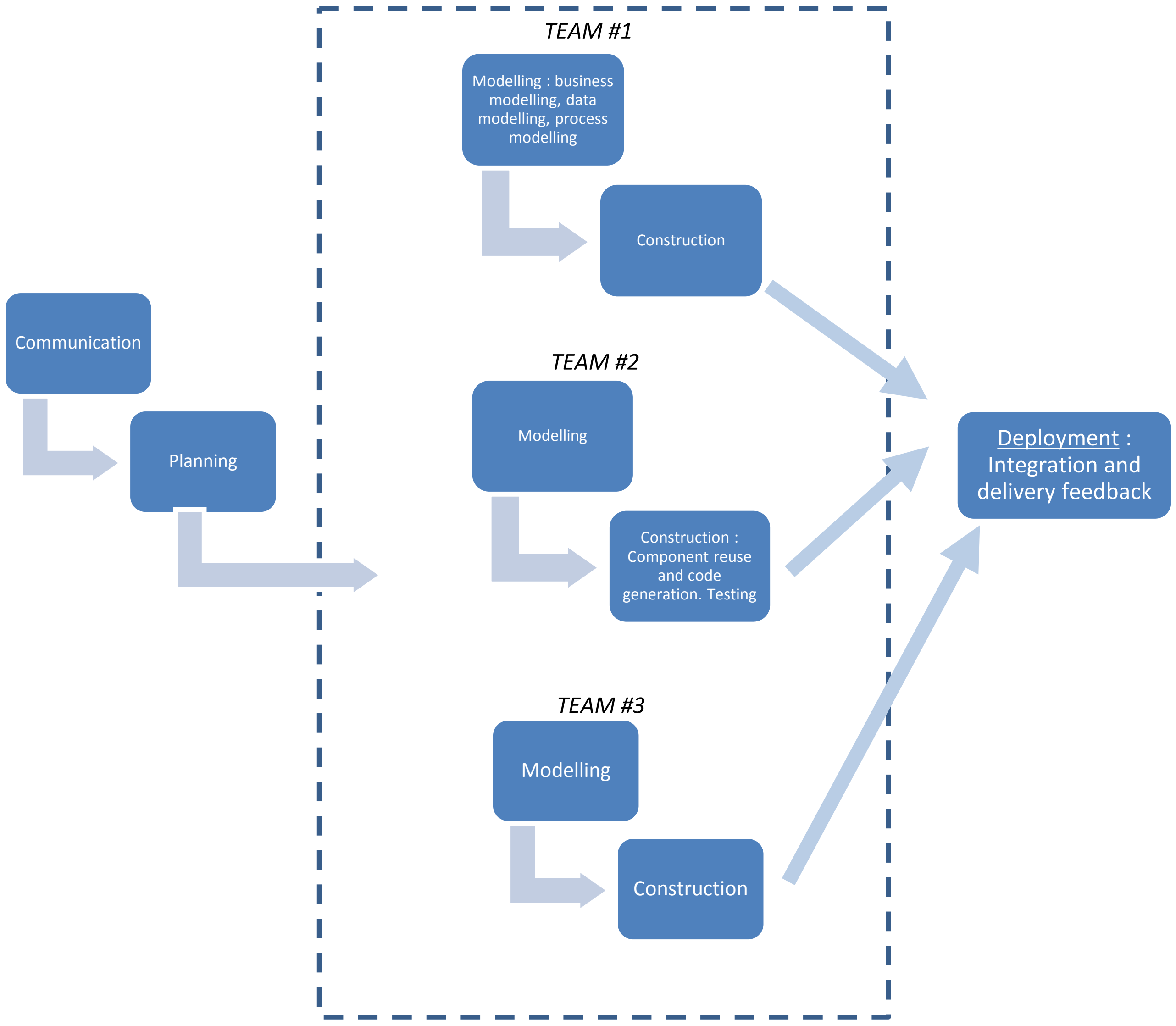
The Increment Model



The incremental model applies linear sequence in a staggered fashion as calendar time progresses. Each Linear sequence produces deliverable "increments" of the software. For e.g. word-processing software developed using the incremental paradigm might deliver basic file management, editing and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third, and advanced page layout capability in the fourth increment.

The Rapid Application Development (RAD) Model

RAD is an incremental software process model that emphasizes a short development cycle. It is a "high-speed" adaptation of the waterfall model, in which rapid development is achieved by using a component based construction approach.

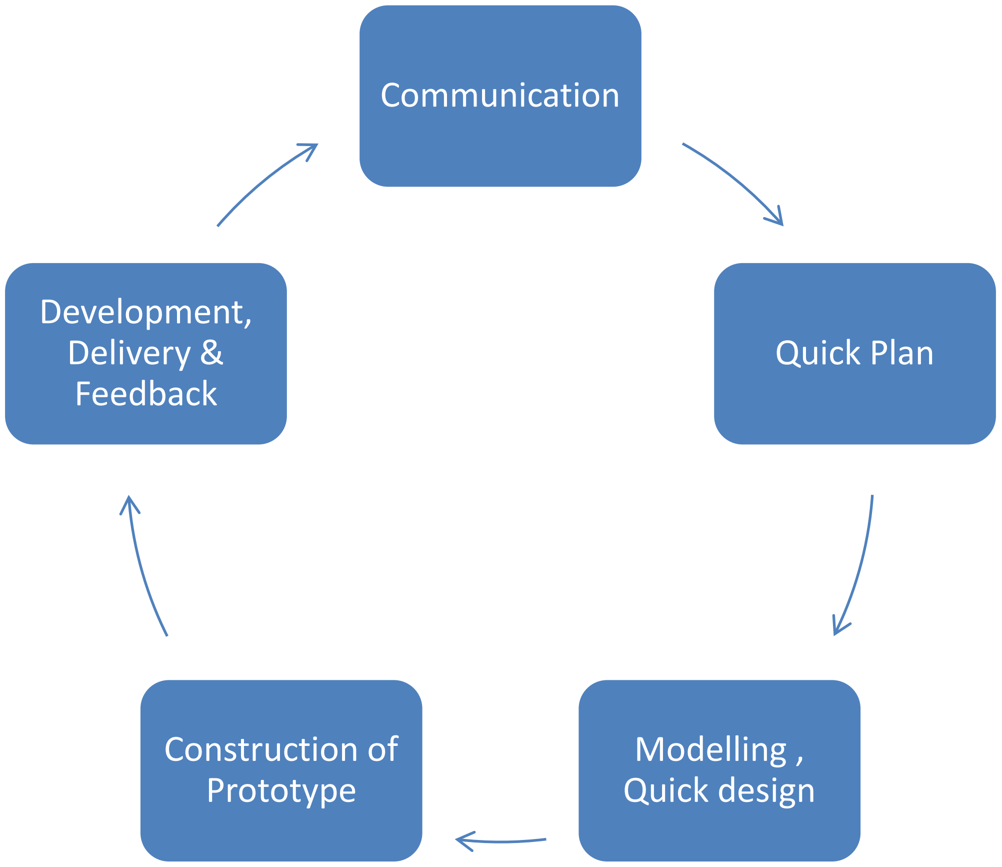


Communication
Planning
TEAM #1
Modelling : business modelling, data modelling, process modelling
Construction
TEAM #2
Modelling
Construction : Component reuse and code generation. Testing
TEAM #3
Modelling
Construction
Deployment : Integration and delivery feedback

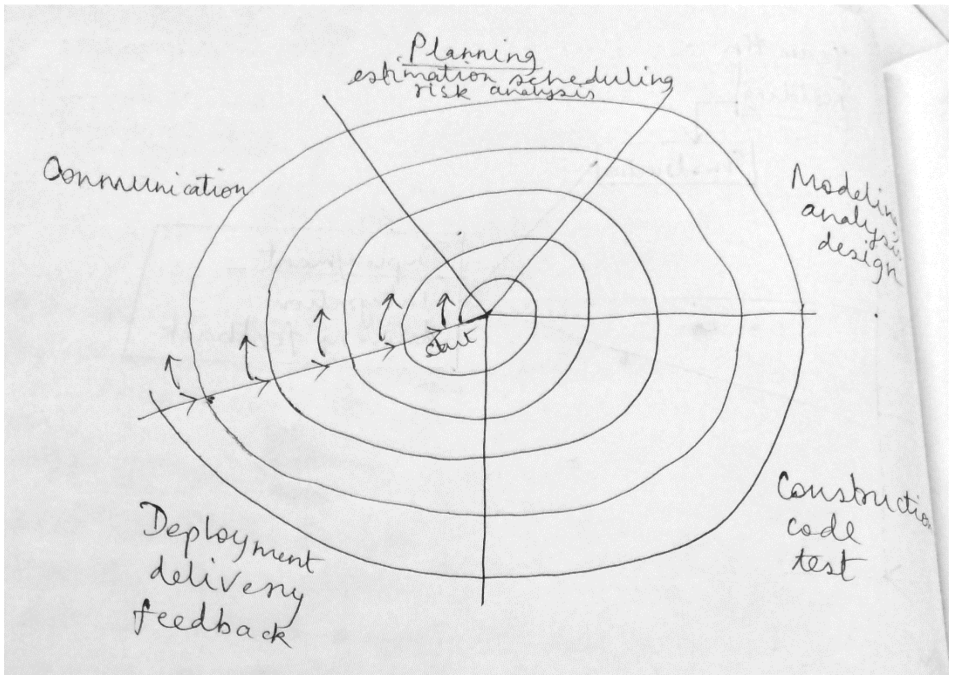
<u>THE PROTOTYPING MODEL</u>

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when the requirements are fuzzy.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time, enabling the developer to better understand what needs to be done.



THE SPIRAL MODEL



Each of the framework activities represent one segment of the spiral patch.

As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. Anchor point milestones – a combination of work products and conditions that are attained along the path of the spiral - are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes along the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan.

Cost and schedule are adjusted based on feedback derived from the customer after delivery.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.

The spiral model is a realistic approach to the development of large scale systems and software.

Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.

Software Project Planning

Software project planning encompasses five major activities.

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

Estimation

Estimation begins with a description of the scope of the product. The problem is then decomposed into a set of smaller problems, and each of these is estimated using historical data and experience as guide, problem complexity and risks are considered before a final estimate is made.

The product is a simple table delineating the tasks to be performed, the functions to be implemented and the cost, effort and time involved for each is generated.

Software scope

Software scope denotes the functions and features that are delivered to end users, the data that one inputs, the output, the "content" that is presented to users as a consequence of using the software, and the performance constraints, interfaces and reliability that bound the system.

Scope is defined using one of two technologies:

1. A narrative description of software scope is defined after communication with all stakeholders
2. A set of use-cases is developed by end-users.

The Stakeholders

1. Senior managers who define business issues.
2. Project (technical) managers who plan and motivate, organize and control the practitioners.
3. Practitioners who deliver the technical skills
4. Customers who specify the requirements
5. End-users, who interact with the software.

Software Feasibility

Once scope has been identified, it is reasonable to ask:

- Can we build software to meet this scope?
- Is the project feasible?

Software feasibility has four dimensions

1. 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?
2. Finance- Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford?
3. Time- Will the projects time-to-market beat the competition?
4. Resources- Does the organization have the resources needed to succeed?

Estimation of resources

Major categories of resources:

1. People
2. Reusable Software Components
3. Development environment(hardware and software tasks)

Each resource is specified with four characteristics:

1. Description of the resources
2. Statement of availability
3. Time when the resource will be required.
4. Duration of the time that the resource will be applied

Reusable software components

1. Off the shelf components
2. Full-experience components
3. Partial experience components
4. New components

Software sizing

1. Direct approach- size measured in lines of codes(LOC)
2. Indirect approach – size represented as function points (FP)

LOC and FP techniques have a number of characteristics in common.

Project planner begins with a statement of software scope. He then attempts to decompose software into problem functioning that can each be estimated individually.

<u>FP- based estimation</u>

The function point metric can be used effectively as a means of measuring the functionality delivered by a system:

Measurement of information domain

1. Number of external inputs(EIs)
2. Number of external outputs(EOs)
3. Number of external inquiries(EQs)
4. Number of internal logical files (ILF's)
5. Number of external interface files (EIF's)

Information Domain Value	Count	Weighing factors			
		Simple	Average	Complex	
EIs		3	4	6	
EOs		4	5	7	
EQs		3	4	6	
ILFs		7	10	15	
EIFs		5	7	10	
Count Total					

$$FP = count\ total * [0.65 + 0.01 * \sum F_i]$$

The F_i (i=1 to 14) are value adjustment factors

Value Adjustment factors

The F_i (i=1 to 14) are based on responses to the following questions:

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is Performance critical?
5. Will the system run in an existing heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require input transaction to be built over multiple screens of operations?
8. Are the ILF's updated online?
9. Are the inputs, output files, or inquiries complex?
10. Is the internal processing complex?
11. Is there the code designed to be reusable?
12. Are conversation and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and for ease of use.

Each of the questions is answered using a scale that ranges from 0(not important) to 5 (absolutely essential)

Example

Information Domain Value	Count				Weighing factors	FP Count
	Optimistic	Likely	Pessimistic	Estimate (rounded)		
EIs	20	24	30	24	4	97
EOs	12	15	22	16	5	78
EQs	16	22	28	22	4	88
ILFs	4	4	5	4	10	42
EIFs	2	2	3	2	7	15
Count Total						320

$$Estimate\ count = \frac{(Optimistic\ +\ 4 * Likely\ +\ Pessimistic)}{6}$$

Value adjustment factors:

S. No	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value Adjustment Factor *(F_i)*	4	2	0	4	3	4	5	3	5	5	4	3	5	5

$$\sum F_i = 52$$

$$FP_{estimated} = counttotal * [0.65 + 0.01 * \sum F_i]$$

= 320*[0.65+0.01*52]

=375

COnstructive COst MOdels (COCOMO)

COCOMO was originally described in the book:

Software Engineering Economies- by W.B. Boehm

(Prentice-Hall, upper saddle region, NJ, 198)

Boehm developed his cost estimation system by carefully studying the data related to the sixty-three completed software projects. The size of the projects he studied ranged from very small (5000 SLOC) to very large (Over 1,000,000 SLOC) [SLOC=Source lines of code]

COCOMO consists of three levels of model precision

- Basic
- Intermediate
- Detailed

Labor months (LM) and development time (DT) are estimated.

$$Cost(in\ \$) = Labour\ months * average\ cost\ per\ month$$

<u>Basic COCOMO Model</u>

The basic COCOMO distinguishes the software development modes

- Organic:
  - The development team consists of a small number of experienced developers who are familiar with the application are and the development environment
  - The project size ranges from small to medium
    LM(Estimated labor months) is
    LM=2.4*(KSLOC) $^{1.05}$
    Project Schedule (Estimated development time) is
    DT=2.5*(LM) $^{0.38}$
    [KSLOC= estimated size of the project measured in thousands of source lines of equivalent new code]
- Semi-Detached
- Embedded
  - Demands the most resources
  - Product has stringent requirements on reliability, performance, schedule, external(to the software) interfaces and timing as well as tightly controlled development environment, frequent reviews and rigid test requirements.
  - The problem being solved is usually unique in some way and frequently complex.
  - Often the development team cannot rely on its prior experience

- The project itself ranges in size from medium to very large.
  LM(Estimated labor months) is
  LM=3.6*(KSLOC)$^{1.20}$
  Project Schedule (Estimated development time) is
  DT=2.5*(LM)$^{0.32}$

Risk Management

1. <u>Reactive Strategy:</u>

   The majority of software teams rely solely on reactive risk strategies. At best a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then the team flies into action in an attempt to correct the problem rapidly. This is often called a fire fighting mode. When this fails, crisis management takes over and the project is in real jeopardy.

2. <u>Proactive strategy:</u>

   A proactive strategy begins long before technical work is initiated. Potential risks are identified, then probability and impact are assessed and they are ranked by importance. Then the software team establishes a plan for managing risks. The primary objective is to avoid risk but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

Software Risks

Risk involves two characteristics.

1) Uncertainty: risk may or may not happen
2) Loss: if the risk becomes a reality, unwanted consequences or losses will occur

<u>Categories of risks:</u>

1) Project risks 2) Technical risks 3) Business risks.

- If project risks become real, the project schedule sleeps and costs increase.
- If a technical risk becomes a reality, implementation may become difficult or impossible.
- Business risks threaten the viability of the software to be built.

<u>Risk Identification:</u>

Risk identification is a systematic attempt to specify threats to the project plan.
Generic risks are a potential threat to every software project.
Product specific risks can be identified only by those with a clear understanding of the technology, the people and the environment that is specific to the software to be built.

Generic risks:

1) Product size.
2) Business impact.
3) Customer characteristics
4) Process definition
5) Development environment
6) Technology to be built
7) Staff size and experience.

Assessing Overall Project Risk

A relatively short list of questions can be used to promote a preliminary indication of whether a project is "at risk"

1) Have top software and customer managers formally committed support the project?
2) Are end-users enthusiastically committed to the project and the system/product to be built?
3) Are requirements fully understood by the software engineering team and its customers?
4) Have the customers been involved fully in the definition of requirements?
5) Do end-users have relatively realistic expectations?
6) Is project scope stable?
7) Does the software engineering team have the right mix of skills?
8) Are project requirements stable?
9) Does the project team have experience with the technology to be implemented?
10) Is the number of people on the project team adequate to do the job?
11) Do all customer/user constituencies agree on the importance of the project and on the requirements of the system/product to be built?

Risk Projection:

Goal to prioritize risks:

Risks	Category	Probability	Impact
Size estimate be significantly low	PS	60%	2
Larger number of users than planned	PS	30%	3
Less reused than planned	PS	70%	2
End users resist system	BU	40%	3
Deliver deadline will be tightened	BU	50%	2
Funding will be lost	CU	40%	1

Legend:

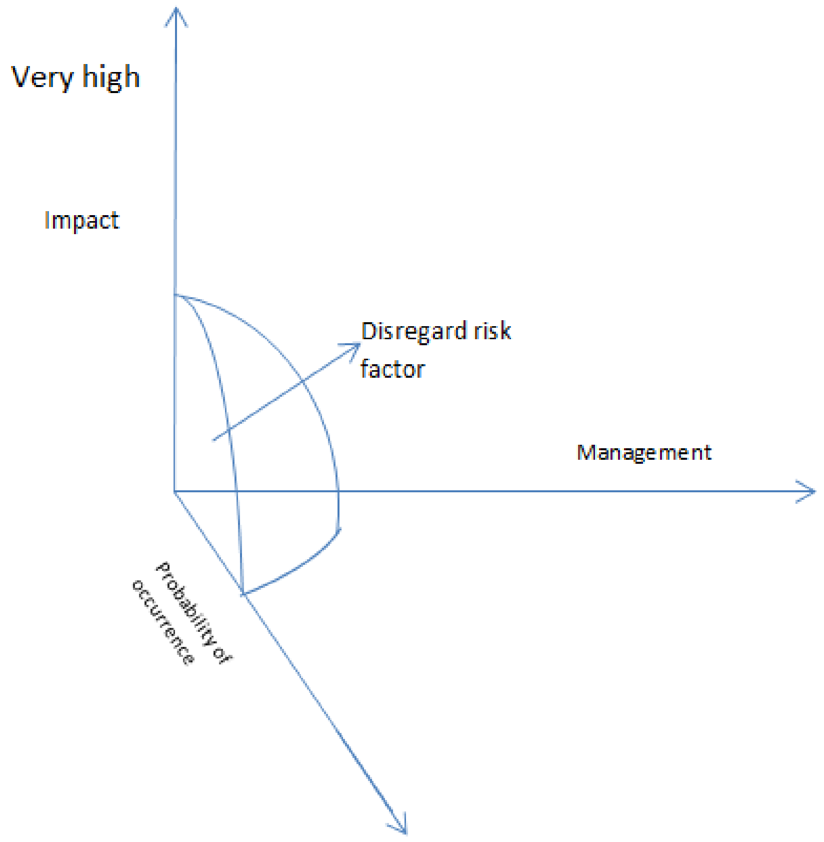
Impact	*Impact Values*
1	*Catastrophic*
2	*Critical*
3	*Marginal*
4	*Negligible*

	Category
PS	*Product Size*
BU	*Business Impact*
CU	*Customer Characteristics*
TE	*Technology to be built*

<u>Risk Project (or estimation) using risk table:</u>

Once the first four columns of the risk table have been completed the table is sorted by probability and impact. High probability high impact risks percolate to the top of the list and low probability risk drop to the bottom

Cutoff Line: The project manager studies the resultant sorted table and defines a cutoff line (drawn horizontally at some point in the table.) The cutoff line implies that only risks that will be above the line will be given further attentions. Risks that fall below the line are reevaluated to accomplish second order prioritization.



Processing risk Impact:

Risk exposure can be computed for each risk in the risk table:

$$RE = P * C$$

RE= Risk exposure
P=Probability of occurrence of risk
C=cost to the project if risk occurs

Example:

Risk Identification: Only 70% of the software components scheduled for reuse, will, in fact be integrated into the application. The remaining functionality will have to be custom developed

Risk probability = 80%

Risk impact:

- Number of reusable software components plan = 60
- Number of components to be developed from scratch= (100-70)% of 60=18
- Average size of component=100LOC
- SWE cost for 1 LOC=$14.00
- Overall cost to develop components (C) =18*100*14=$25200

Risk Exposure (RE) =P * C = 0.8*$25200=$20050

Project Scheduling

Select a process Model

Identify SWE tasks to be performed

Estimate amount of work and no of people

Consider the risks

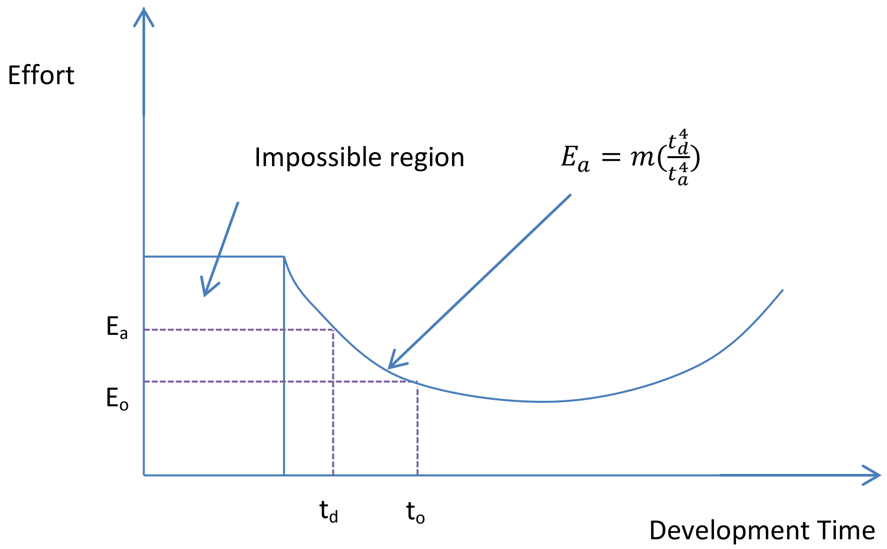
Create a network of SWE tasks to get the job done on time

Assign responsibility for each task

Why?

To build a complex system, many software engineering tasks occur in parallel and the result of work performed during the task may have a profound effect on work conducted in another task. These interdependencies are very difficult to understand without a schedule. It is also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

Relationship between effort and delivery time



Putnam Norden Rayleigh (PNR) Curve

E_a=Effort in person months
t_d=Nominal delivery time for schedule
t_o=Optimal development time (in terms of cost)
t_a=Actual delivery time desired.

Software Equation:

$$L = P * E^{.33} * t^{1.33}$$

$$E = L^3/(P^3 t^4)$$

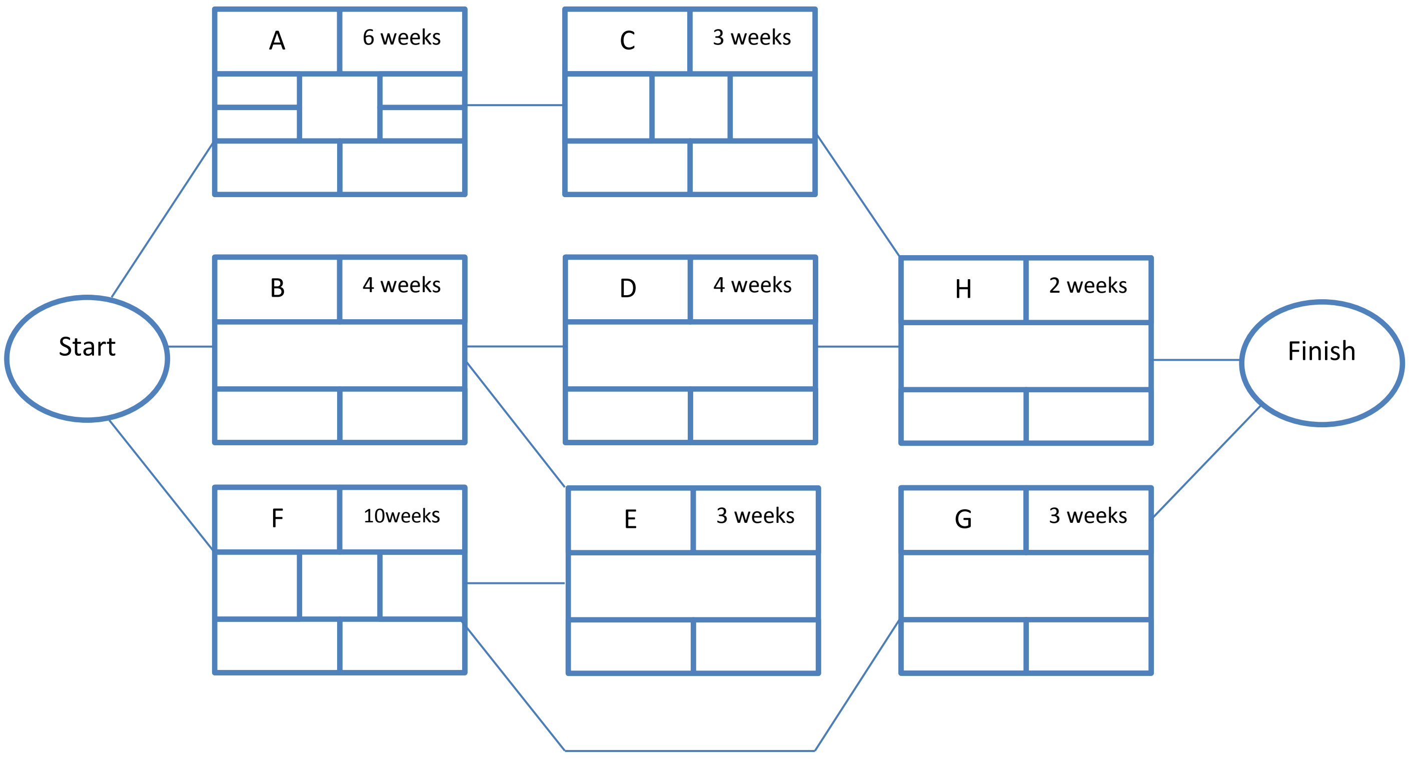
L= number of delivered lines of code(source statements)

P=Productivity parameter(typically 2000-12000)

t=Project duration in calendar months

E=Development effort in person month

Network Model:



Activity Label		Duration
Earliest Start	Activity Description	Earliest Finish
Latest Start		Latest Finish
Activity Span		Float

Critical path methods:

Two objectives:

1) Planning the project in such a way that it is completed as quickly as possible
2) Identifying those activities where a delay in their execution is likely to affect the overall end date of the project.

- The method requires that for each activity we have an estimate of its duration.
- The method is then analyzed by carrying out a forward pass, to calculate the earliest dates at which activities may commence and the project can be completed, and a backward pass to calculate the latest start, latest dates for activities and the critical path.

Backward pass:

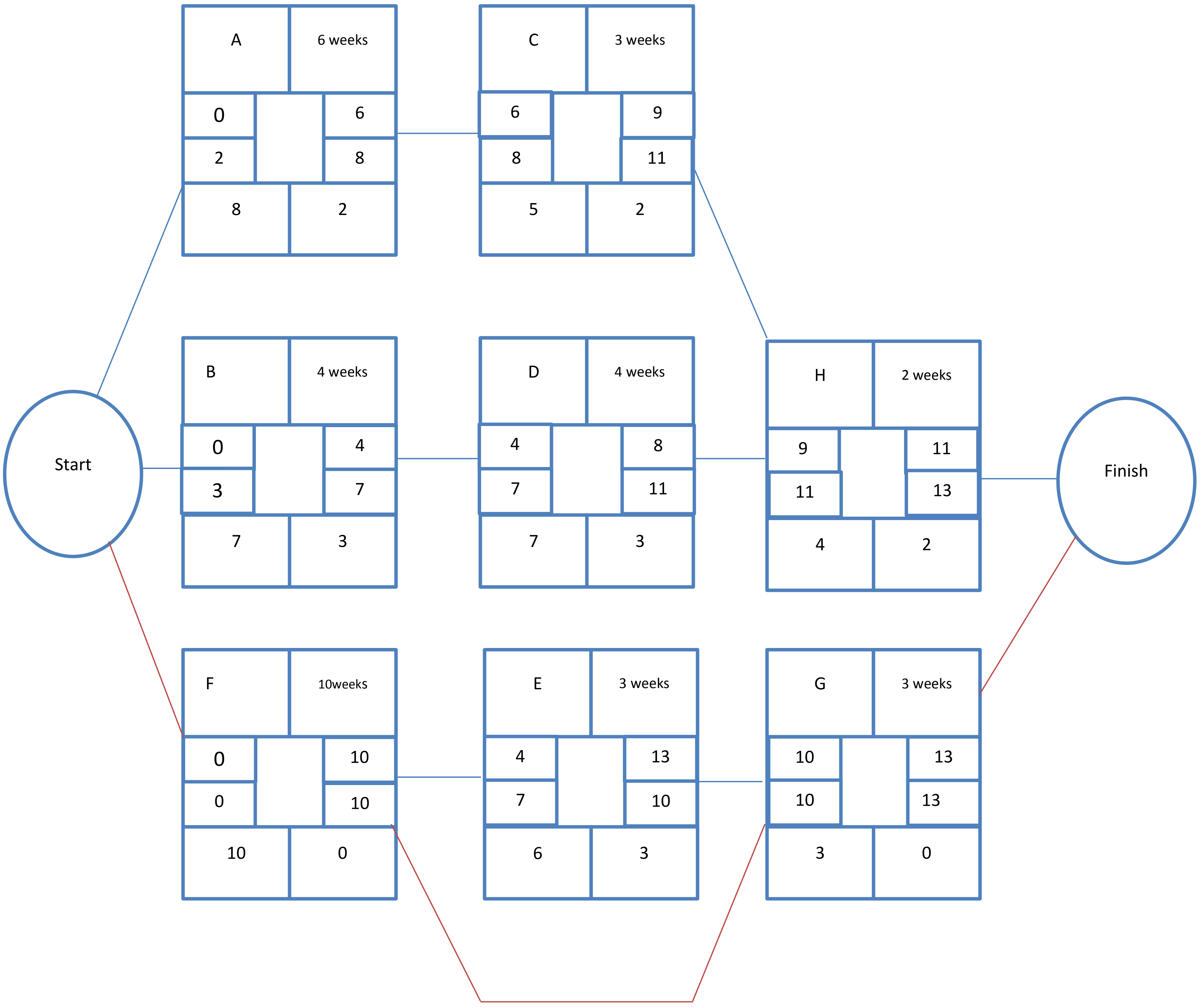
We assume that the latest finish date for the project is the same as the earliest finish date-that is we want to complete the project as early as possible.

The latest finish date for an activity is the latest start date for all activities that may commence immediately that activity is complete.

When more than one activity can commence, we take the earliest of the latest start dates for those activities.

$$Float = Latest\ Start\ Date - Earliest\ Start\ Date$$

$$= Latest\ Finish\ Date - Earliest\ Finish\ Date$$


Start
A
6 weeks
0
6
2
8
8
2
C
3 weeks
6
9
8
11
5
2
B
4 weeks
0
4
3
7
7
3
D
4 weeks
4
8
7
11
7
3
H
2 weeks
9
11
11
13
4
2
F
10weeks
0
10
0
10
10
0
E
3 weeks
4
13
7
10
6
3
G
3 weeks
10
13
10
13
3
0
Finish

Identifying the critical path:

There will be at least one path through the network that defines the duration of the project. This is known as the critical path.

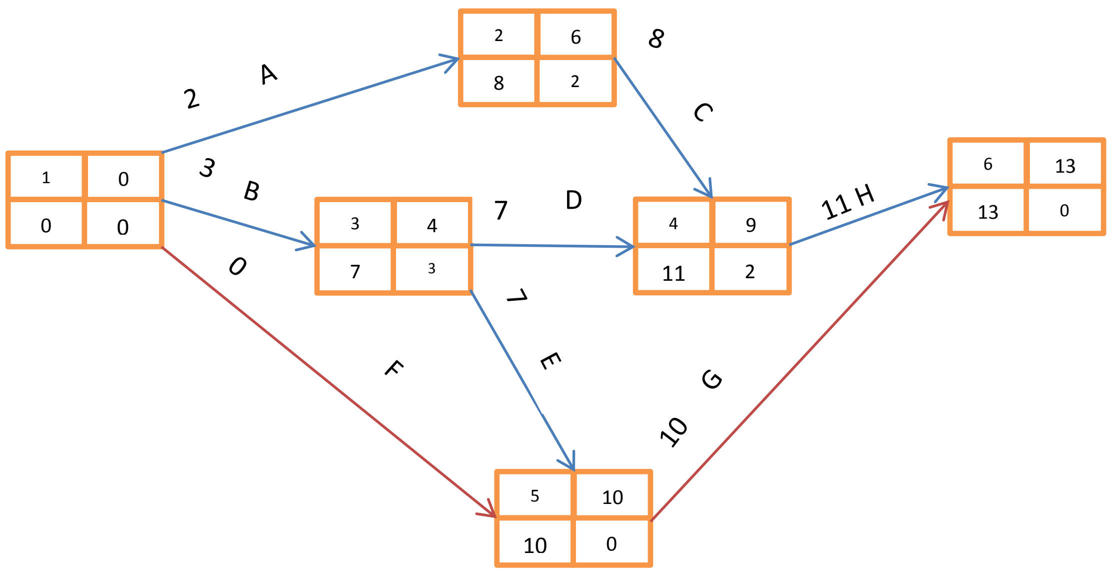
Any delay to any activity on this critical path will delay the completion of the project.

Float is a measure of how much the start or completion of the activity may be delayed without affecting the end date.

Any activity with a float of zero is critical in the sense that any delay in carrying out the activity will delay the completion date of the project as a whole.

Activity on arrow network.

Activity	Duration
A	6
B	4
C	3
D	4
E	3
F	10
G	3
H	4



Event Number	Earliest finish date
Latest finish Date	Slack = Latest – Earliest finish date

Activity	Duration	Earliest start	Latest start	Earliest Finish	Latest Finish
A	6	0	2	6	8
B	4	0	3	4	7
C	3	6	8	9	11
D	4	4	7	8	11
E	3	4	7	7	10
F	10	0	0	10	10
G	3	10	10	13	13
H	2	9	11	11	13

Backward Pass:

Assume:

Latest finish date = earliest finish date.

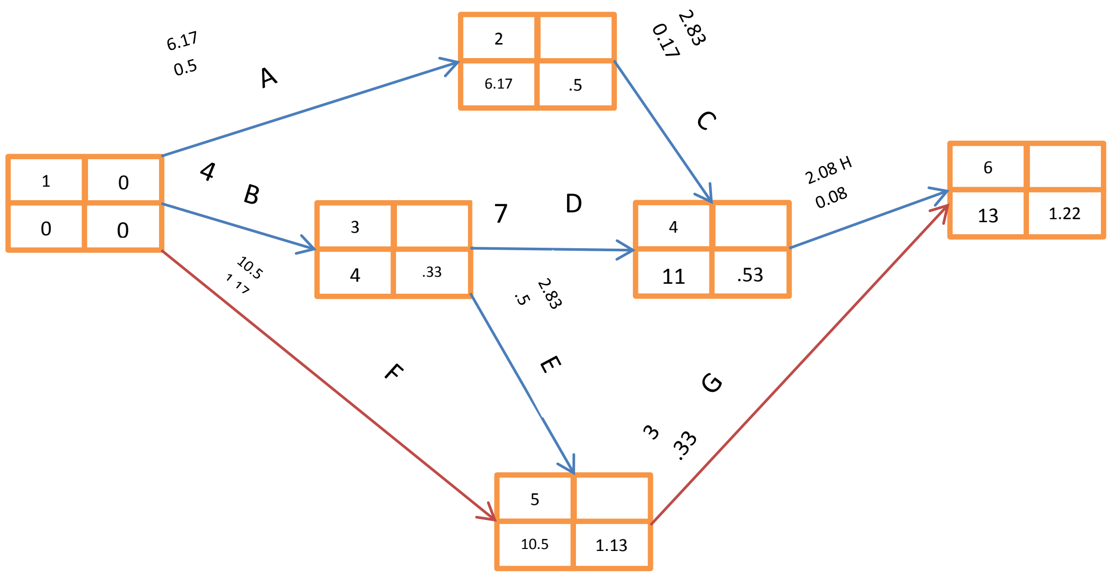
Slack= latest start-earliest start

Slack is a measure of how late an event may be without affecting the end date of the project.

<u>Uncertainty in task duration:</u>

PERT(program evaluation and review technique) provides methods for estimating the probability of affecting missing target dates.

Activity	Activity Duration		
	Optimist	Most likely	pessimists
A	5	6	8
B	3	4	5
C	2	3	3
D	3.5	4	5
E	1	3	4
F	8	20	15
G	3	4	4
H	2	2	2.5



Event Number	Target date
Expected Date	SD

t = (optimistic + 4*likely + pessimistic)/6

Standard deviation of an activity time is

S=(b-a)/6.

Where, b=Pessimistic activity duration

a=Optimistic activity Duration

It is a measure of the degree of uncertainty of an activity duration estimate.

<u>Method of calculating the probability of meeting of or missing target date.</u>

1) Calculate the standard deviation of each project event.

2) Calculate the Z value for each event that has a target date.
3) Convert z-values to probabilities

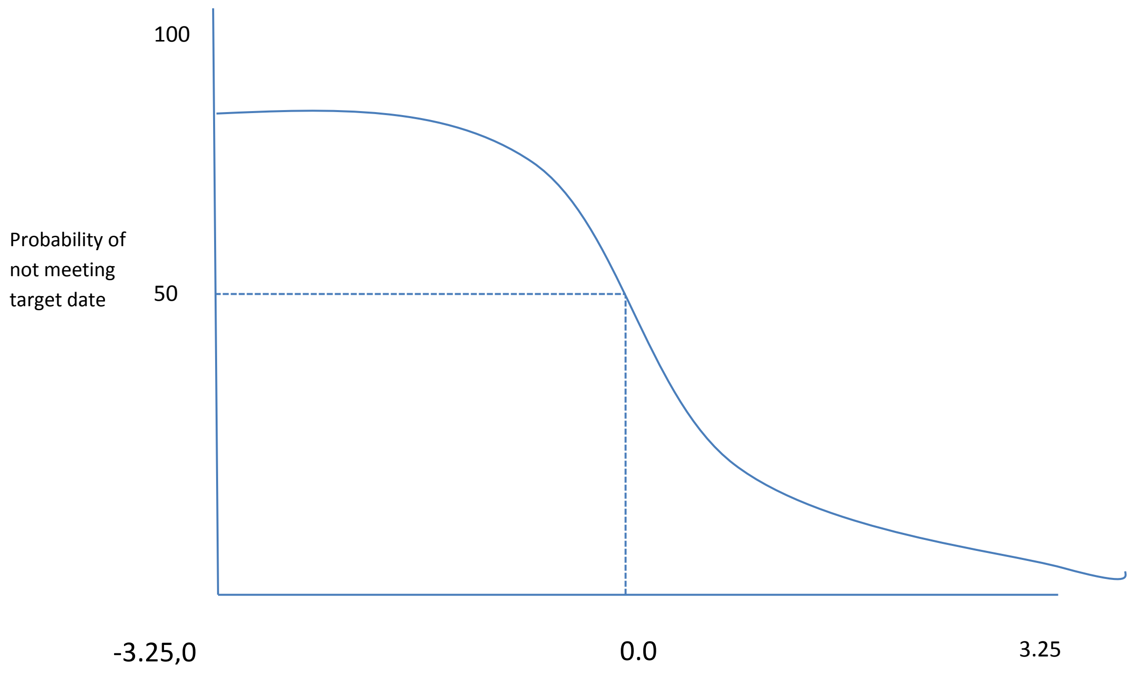
<u>Calculating Z-values:</u>

For a node having a target date T,

$$z = (T - t_e)/S$$

S=standard deviation for the event,
t_e= Expected date

<u>Converting Z-values to probabilities:</u>



EARNED VALUE ANALYSIS

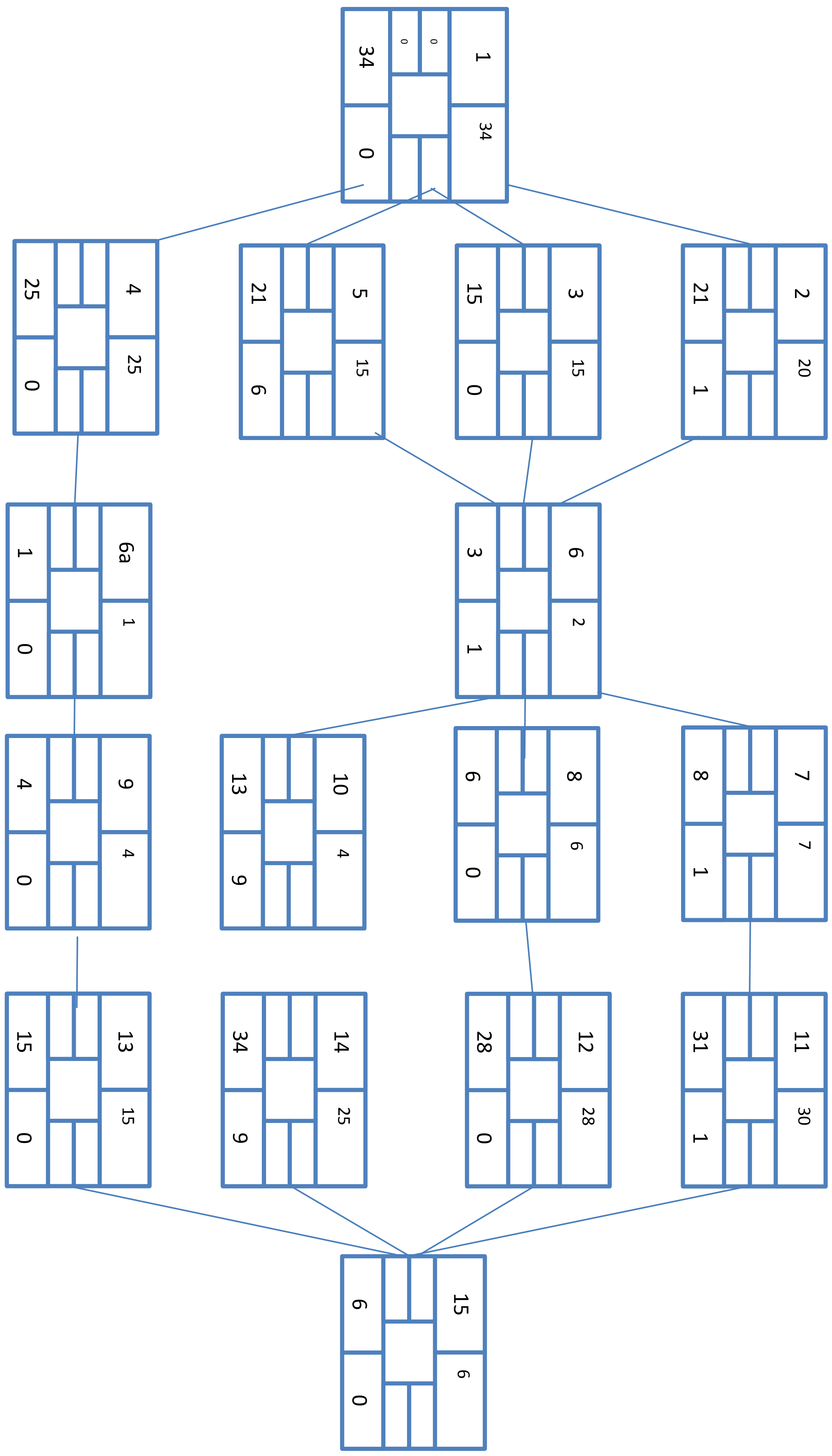
Objective: Monitoring program

Earned value analysis is based on assigning is based on assigning a “value” to each task based on the original expenditure forecast.

BCWS(budgeted cost of work scheduled)is the original budgeted cost for the item (also called “baseline budget”)

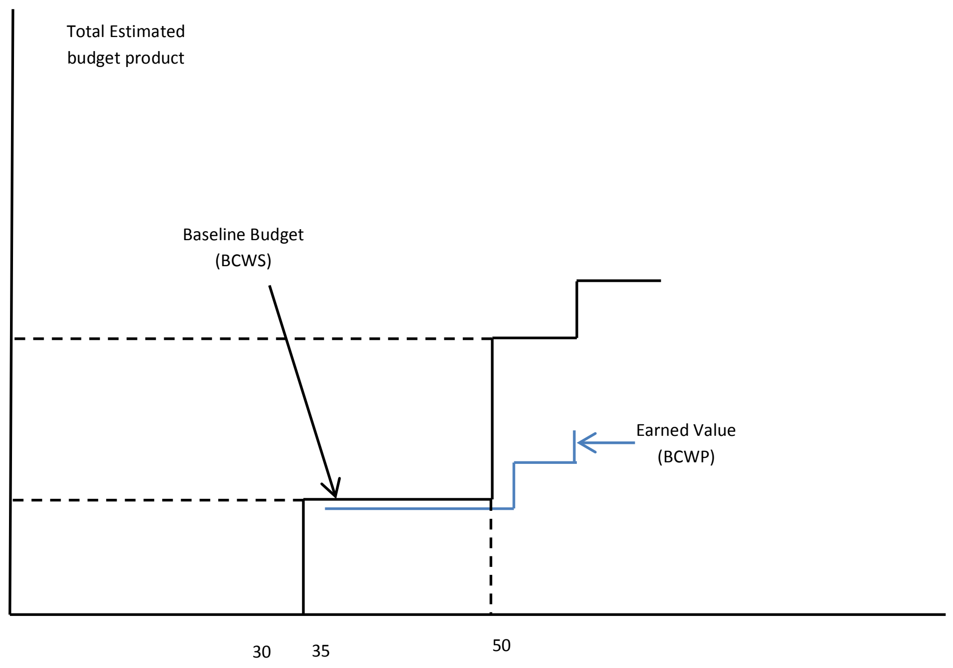
BCWP(budgeted cost of work performed) is the total value credited to a project at any point.

Creating the baseline budget:



1
34
34
0
0
0
2
20
21
1
3
15
15
0
5
15
21
6
4
25
25
0
6
2
3
1
6a
1
1
0
7
7
8
1
8
6
6
0
10
4
13
9
9
4
4
0
11
30
31
1
12
28
28
0
14
25
34
9
13
15
15
0
15
6
6
0

Task	Budgeted work days	Scheduled completion rate	Cumulative work days	% Cumulative earned value
1	34	34	34	14.35
3	15	49	64	32.00
5	15	49		
2	20	54	84	35.44
6	2	56	86	36.28
10	4	60	90	37.97
7	7	63	97	40.93
8	6	66	103	43.46
				43.88
4	25	74	129	54.43
9	4	79	153	56.12
14	25	85	158	66.67
11	30	93	188	79.32
12	28	94	231	97.47
13	15	94		
15	6	100	237	100



QUALITY MANAGEMENT:

Variation control is the heart of quality control

From one project to another, we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment and calendar time.

In general we would like to make sure our testing program covers a known percentage of the software from one release to another

We would like to ensure that the variance in the number of bugs is also minimized from one release to another

COST OF QUALITY

1. Preventive costs include quality planning, formal technical reviews, test equipment and training
2. Appraisal costs include activities to gain insight into product condition the "first time through" each process.
3. Failure costs are those that would disappear if no defects appeared before shipping a product to customers.
   3.1 Internal failure costs are incurred when we detect a defect in our product prior to shipping.
   3.2 External failure costs are associated with defects found after the product has been shipped to the customer.

SOFTWARE QUALITY

It is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

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

SOFTWARE QUALITY ASSURANCE (SQA)

Software quality assurance is a “planned and systematic pattern of actions” that are required to ensure high quality is software.

Many constituencies have software quality assurances responsibility---Software engineers, project managers, customers, sales people, and the individuals who serve within an SQA group.

The SQA group serves as the customer’s in house representative.

That is, the people who perform SQA must look at the software from the customer’s point of view. The SQA group attempts to answer the following questions and others:

1. Does the software adequately meet the quality factor?
2. Has software development been conducted according to pre-established standards?
3. Has technical disciplines properly performed their roles as part of the SQA activity?

McCall’s Quality Factors

James A. McCall grouped software qualities into three sets of quality factors:

- Product operation qualities;
- Product revision qualities;
- Product transition qualities;

Product Operation Quality Factors:

- Correctness: The extent to which a program satisfies its specifications and fulfills the user’s objectives.
- Reliability: The extent to which a program can be expected to perform its intended function with required precision.
- Efficiency: The amount of computer resources required by the software.
- Integrity: The extent to which access to software or data by unauthorized persons can be controlled.
- Usability: The effort required to learn, operate, prepare input and interpret output.

McCall's Quality Factor:

Product revision quality factor:

Maintainability: The effort required to locate and fix an error in an operational program.

Testability: The effort required to test a program to ensure it performs its intended function.

Flexibility: The effort required to modify an operational program.

Product transition quality factors

Portability: The effort required to transfer a program from one hardware configuration and/or software system environment to another.

Reusability: The extent to which a program can be used in other applications.

Interoperability: The effort required to couple one system to another.

Software Quality Criteria

McCall's software quality factors reflect the external view of software that users would have. These quality factors have to be translated into internal factors which the developers would be aware- software quality criteria.

Quality Factor	Software Quality Criteria
Correctness	Traceability, consistency, completeness
Reliability	Error tolerance, consistency, accuracy, simplicity
Efficiency	Execution efficiency, storage efficiency
Integrity	Access control, access audit
Usability	Operability, training, communicativeness, input/output volume, input/output rate
Maintainability	Consistency, simplicity, conciseness, modularity, self-descriptiveness
Testability	Simplicity, modularity, instrumentation, self-descriptiveness
Flexibility	Modularity, generality, expandability, self-descriptiveness
Portability	Modularity, self-descriptiveness, machine-independence, software system independence
Reusability	Generality, modularity, software system independence, machine independence, self-descriptiveness
Interoperability	Modularity, communications commonality, data commonality

ISO 9126 QUALITY FACTORS

ISO 9126 standard was published in 1991 to tackle the question of the definition of software quality. It identifies six software quality characteristics:-

- Functionality, which covers the functions that a software product provides to satisfy user needs;
- Reliability, which relates to the capability of the software to maintain its level of performance;
- Usability, which relates to the effort needed to use the software;
- Efficiency, which relates to the physical resources used when the software is executed;
- Maintainability, which relates to the effort needed to make changes to the software;
- Portability, which relates to the ability of the software to be transferred to a different environment.

Formal Technical Review(FTR)

A formal technical review is a software quality control activity performed by software engineers and others. The objectives of an FTR are:

1) To uncover errors in function, logic or implementation for any representation of the software;
2) To verify that the software under review meets its requirements;
3) To ensure that the software has been represented according to predefined standards;
4) To achieve software that is developed in a uniform manner; and
5) To make projects more manageable

Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

THE REVIEW MEETING

Every review meeting should abide by the following constraints:

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

An FTR focuses on a specific (and small) part of the overall software.

The focus of the FTR is on a work product (e.g., a portion of a requirements specification, a detailed component design, a source code listing for a component).

The individual who has developed the work product – the producer – informs the project leader that the work product is complete and that a review is required.

The project leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.

Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the product.

Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers, and the producer.

One of the reviewers takes on the role of the recorder; that is, the individual who records (in writing) all important issues raised during the review.

The FTR begins with an introduction of the agenda and a brief introduction by the producer.

The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation.

At the end of the review, all attendees of the FTR must decide whether to

1) Accept the product without further modification,
2) Reject the product due to severe errors, or
3) Accept the product provisionally.

FTR GUIDELINES

1) Review the product, not the producer.
2) Set an agenda and maintain it.
3) Limit debate and rebuttal.
4) Enunciate problem areas, but don't attempt to solve every problem noted.
5) Take written notes.
6) Limit the number of participants and insist upon advance preparation.
7) Develop a checklist for each product that is likely to be reviewed.
8) Allocate resources and schedule time for FTR's.
9) Conduct meaningful training for all reviewers.
10) Review your early reviews.

SOFTWARE RELIABILITY

A software system contains a fault if, for some input data, the behavior of the system is incorrect, that is, different from one included in software specification.

For each execution of the software system where its output is incorrect, we talk about software failure.

Thus the failure is a consequence of a certain fault being left in software.

<u>Classic definition</u>

Software reliability is a probability that the software system will function without failure under a given period of time.

STARTING POINT FOR SOFTWARE PROJECTS

	Requirements must be determined	Clients have produced requirement
New Development	A	B
Evolution of existing systems	C	D

Projects of type A or B – development team starts to develop new software from scratch ("green field development")

Projects of type C & D – the team evolves an existing system

A & C – development team has to determine the requirements for the software.

B & D – Development team is contracted to design and implement a very specific set of requirements. The customers' organization has normally done the requirements analysis, perhaps using in-house software engineers or consultants specializing in requirements analysis.

Projects where the requirements are pre-specified should be handled carefully. A software engineer should not accept a contract where he's required to implement requirements with no changes allowed.

REQUIREMENT

A requirement is a statement about what the proposed system will do that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.

Each requirement is a relatively short and concise piece of information, expressed as a fact. It can be written as a sentence or can be expressed using some kind of diagram. A collection of requirements is called a requirements document.

A statement about the system should not be declared to be an official requirement until all the stakeholders (users, customers, developers & their management) have reviewed it, have negotiated any needed changes and have agreed that it is valid.

A requirement says something about the tasks the system is supposed to accomplish. It does not describe the domain, nor how the system will be implemented.

A requirement should only be present if it helps solve the customer's problems.

Types of Requirements

1) Functional
2) Non Functional

Functional Requirements:

Functional requirements describe what the system should do, i.e. the services provided for the users and for other systems.

The functional requirements should include:

i) Everything that a user of the system would need to know regarding what the system does
ii) Everything that would concern any other system that has to interface to this system

Details that go any deeper into how the requirements are implemented should be left out.

Non-functional requirements:

Non-functional requirements are constraints that must be constrained, that must be adhered to during development. They limit what resources can be used and set bounds on aspects of the software's quality.

Functional Requirements: -- Categories

The fundamental requirements can be capitalized as follows

- What <u>inputs</u> the system should accept and under what conditions. This includes data and commands from the users and from other systems.
- What <u>outputs</u> the system should produce and under what condition
- What data the system should store that other systems might use
- What <u>computations</u> the system should perform
- The <u>timing and synchronization</u> of the above. Not all systems involve timing and synchronization – this category of functional requirements is of most importance in the hard real time systems.

An individual requirement often covers more than one of the above categories. For example, the requirements of a word processor might say, "When the user selects "word count", the system displays a dialog box listing the number of characters, words, sentences, lines, paragraphs, pages, and words per sentence of the current document". This requirement clearly describes input (selecting "word count"), output (what is displayed), and computation (counting all the information and computing words per sentence).

Non-functional Requirements:

Non-functional requirements can be divided into several groups:

GROUP-I

1.1 Response time: For systems that process a lot of data or use a network extremely, we should require that the system gives feedback to the user in a certain minimum time

1.2 Throughput: For number crunching programs that may take hours, or for servers that continually respond to client requests, it is a good idea to specify throughput in terms of computations or transactions per minute

1.3 Resource usage: For systems which use non-trivial amounts of such resources as memory and network bandwidth, we should specify the maximum amount of these resources that the system will consume

1.4 Reliability: Reliability is measured as the average amount of the time between failures or the probability of a failure in a given period.

1.5 Availability: Availability measures the amount of time that a server is running and available to respond to users

1.6 Recovery from failure: A non-functional requirement in this category specifies the maximum allowed impact of a failure. We should state that if the hardware or software crashes, or the power fails, then the system will be able to recover within a certain amount of time, and with a certain minimal loss of data.

1.7 Allowances for maintainability and enhancement: In order to ensure that the system can be adapted in the future, we should describe changes that are anticipated for subsequent releases.

1.8 Allowances for Reusability: It is desirable in many cases to specify that a certain percentage of the system, e.g. 40%, measured in terms of lines of code, must be designed generically so that it can be reused.

GROUP-II

2.1 Platform: It is quite important to make it clear what hardware and operating system the software must be able to work on. Normally such requirements specify the least powerful platforms.

2.2 Technology to be used: Such requirements are normally stated to ensure that all systems in an organization use the same technology – this reduces the need to train people in different technologies.

GROUP-III

3.1 Development process (Methodology) to be used: In order to ensure quality, some requirements documents specify that certain processes be followed, for example, particular approaches to testing.

3.2 Cost and delivery date: They are not often placed in the requirements document, but are found in the contract for the system or are left to a separate project plan document.

Software Design:

In the context of software, design is a problem solving process where objective is to find and describe a way to implement the systems functional requirements while respecting the constraints imposed by the non-functional requirements (including the budget and deadlines), and while adhering to general principles of good quality.

Component:

A component is any piece of software or hardware that has a clear role and can be isolated, allowing us to replace it with a different component with equivalent functionality. Components can be source code files, executable files, dynamic linked libraries and databases.

Module:

A module is a component that is defined at the programming language level. For example, methods, classes and packages are modules in java.

Design Engineering (Continued)

Framework: A framework is reusable software that implements a generic solution to a generalized problem. It provides common facilities applicable to different application programs.

Key principle: Applications that do different but related things that have similar designs – in particular, the patterns of interaction among the components tend to be very similar

The key thing that distinguishes a framework from other kinds of software subsystem is that a framework is intrinsically incomplete. This means that there certain classes or methods that are used by the framework, but which are missing.

The missing parts are often called slots. The application developer fills in these slots in an application specific way to adapt the framework to his or her needs.

Frameworks also usually have hooks. These are like slots, except that they represent functionality that it is optional for the developers to provide when they exploit the framework. Developers using frameworks not only fill slots and hooks, but they also use the services that the framework provides, i.e. methods that perform useful functions. The set of services taken together is often called the application program interface (API)

Example:

A framework for payroll management:

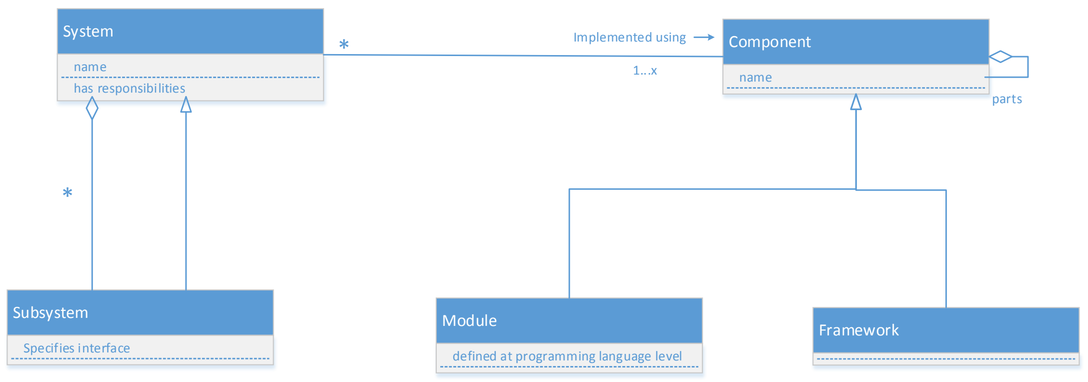
Most businesses have software that includes a payroll module. The rules and features needed in a payroll system will differ considerably, depending on the type of business, the local jurisdiction and other software the company uses. However, basic elements such as making regular payments, and computing taxes and other deductions will always exist. Although it is possible to purchase complete payroll applications, many businesses are of sufficient complexity that such applications do not implement all the needed features from scratch, several businesses could adapt a common framework to their individual needs.

System:

A system is a logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software, or both. A system can have a specification which is then implemented by collection of components. A system continues to exist, even if its components change over the course of time or are replaced by equivalent components.

Subsystem:

A subsystem is a system that is part of a larger system, and which has a definite interface. Java uses packages to implement particular low level subsystems.



DESIGN PRINCIPLES

1. Divide and Conquer

   A software system can be divided in many ways:

   - A distributed system is divided up into clients and services
   - A system is divided up into subsystems
   - A subsystem can be divided up into one or more packages
   - A package is divided up into classes
   - A class is divided up into methods

2. Increase cohesion, where possible

   Cohesion → Divide things, but keep things together that belong together.

S. NO	COHESION TYPE	WHICH THINGS ARE KEPT TOGETHER
1	Functional (Highest precedence)	Facilities that perform only one computation with no side-effects
2	Layer	Related services with strict hierarchy in which higher level services can access only lower level services
3	Communicational	Facilities for operating on the same data
4	Sequential	A set of procedures, which work in sequence to perform some computation
5	Procedural	A set of procedures which are called one after another
6	Temporal	Procedures used in the same general phase of execution, such as initialization or termination.
7	Utility (Lowest Precedence)	Related utilities when there is no way to group them using a stronger form of cohesion

i) <u>Functional cohesion</u>

This is achieved when a module only performs a single computation, and return a result, without having side effects.

A module lacks side-effects if performing the computation leaves the computation in the same state it was in before performing the computation. The result computed by the module is the only thing that should have an effect on subsequent computations.

The inputs to a functionally cohesive module typically include function parameters, but they can also include files of some other stream of data. Whenever exactly the same inputs are provided, the module will always compute the same result. The result is often a simple return value, but can also be a more complex data structure.

Modules that update a database or create a new file are not functionally cohesive since they have side-effects in the database or file-system respectively.

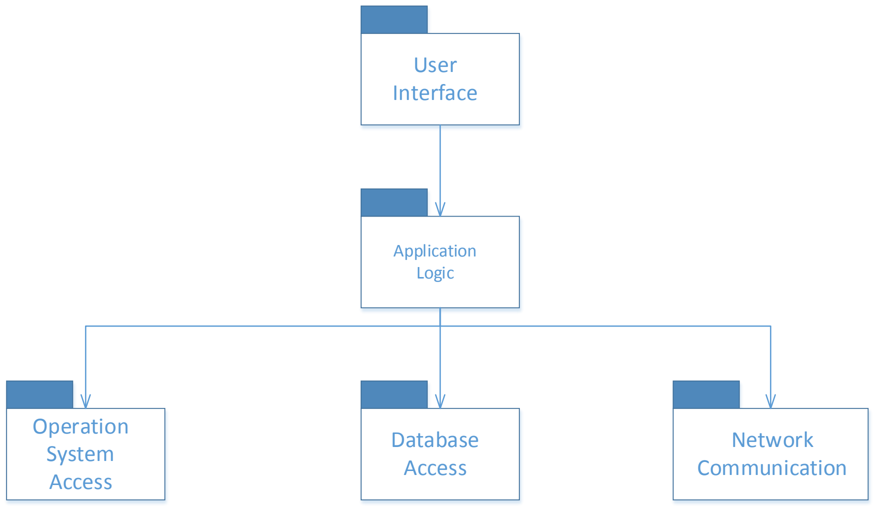
The following are examples of modules that can be designed to be functionally cohesive:

- A module that completes a mathematical function such as sine or cosine
- A module that takes a set of equations and solves for the unknowns

A functionally cohesive module can call the services of other modules, but the called modules must preserve the functional cohesion.

ii) <u>Layer Cohesion</u>

This is achieved when the facilities for providing a set of related services to the user or to higher level layers are kept together, and everything else is kept out



Any individual service in a layer can have functional cohesion. However this is NOT NECESSARY.

To have proper layer cohesion, the layers must form a hierarchy. Higher layers can access services of lower layers, but it is essential that the lower layers do not access higher layers.

The set of methods through which a layer provides services is commonly called an application programming interface (API)

iii) Communicational Cohesion:

This is achieved when modules that access or manipulate certain data are kept together (e.g. in the same class) – and everything else is kept out. One of the strong points of the object oriented paradigm is that it helps ensure communicational cohesion.

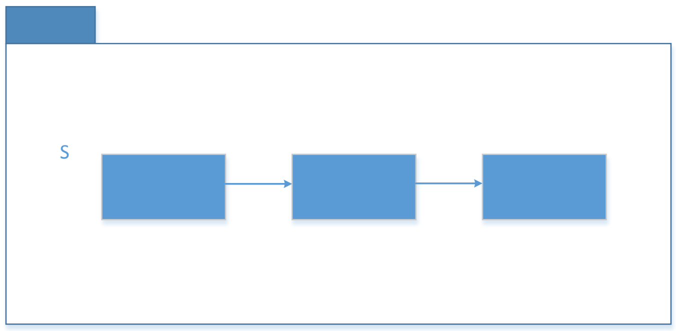
The term "communicational" arises from the fact that a set of procedures which use a piece of data communicate with each other. They do this via the side effects that some of them have on that data.

We should not sacrifice layer cohesion to achieve communicational cohesion for example even though objects may be stored in a database or in a remote host, a class must only load and save objects using the services in the API of the lower layer.

Example: A class called Employee would have good communicational cohesion if all the systems facilities for storing and manipulating employee data are contained in this class, and the class does not do anything other than manage employee data

iv) Sequential Cohesion

This is achieved when a series of procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out.



Module S has sequential cohesion

Our objective should be to achieve sequential cohesion, once we have already achieved functional, layer and communicational cohesion. Methods in two different classes may provide inputs to each other and be called in sequence; but they would each be kept in their own class, since communicational cohesion is more important than sequential cohesion.

v) <u>Procedural Cohesion</u>

This is achieved when we keep together several procedures that are used one after another, even though one does not necessarily provide input to the next

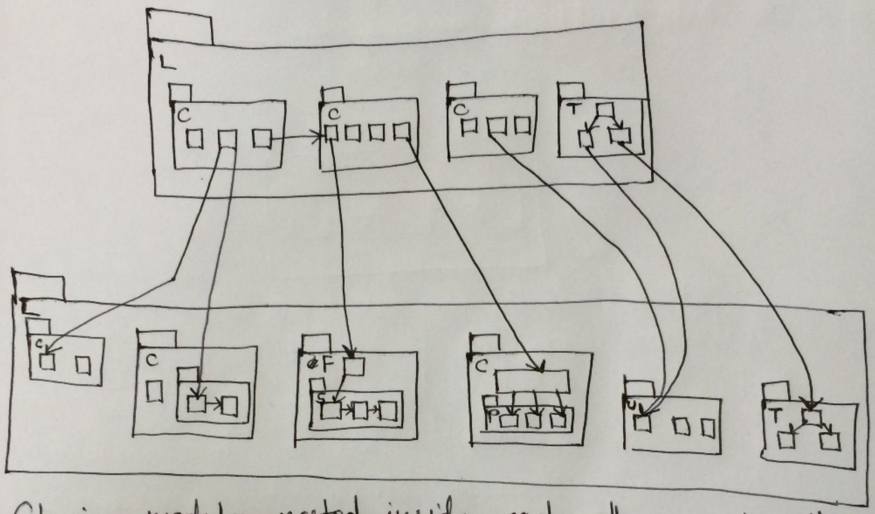
vi) <u>Temporal Cohesion</u>

This is achieved when operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out.

For example, a designer would achieve temporal cohesion by placing together the code used during system start-up or initialization, as long as this doesn't violate one of the other forms of cohesion in (i) …. (v). Similarly, all the code for system termination, or for certain occasionally used features, could be kept together to achieve temporal cohesion.

vii) <u>Utility Cohesion</u>

This is achieved when related utilities, which cannot be logically placed in other cohesive units are kept together. A utility is a procedure or class that has wide applicability to many different subsystems and are designed to be reusable.

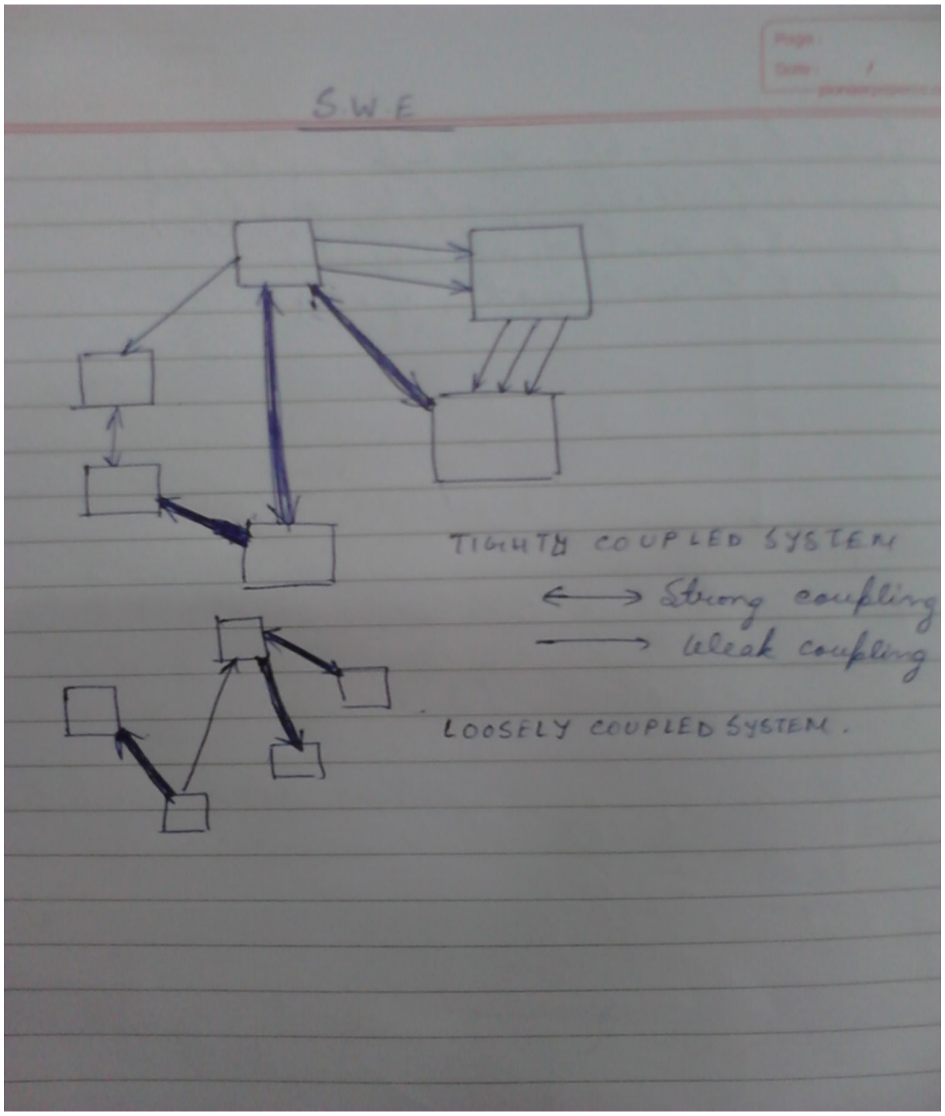


Cohesive modules, nested inside each other, using the services of other modules.

L → Layer
C → Communicational
T → Temporal
F → Functional
P → Procedural
S → Sequential.
U → Utility

Design Principle 3: Reduce COUPLING where possible

Coupling occurs when there are interdependencies between one module and another.



In general the more coupled a set of modules is, the harder it is to understand and hence change the system. Two rules for this are:

1. When interdependencies exist, changes in one place will require changes somewhere else. Requiring changes to be made in more than one place is problematic since it is time-consuming to find the different places that need changing, and it is likely that errors will be made.
2. A network of interdependencies makes it hard to see at a glance how some component works.

Design Principle 4: Keep the level of abstraction as high as possible

We should ensure that our designs allow us to hide or defer consideration of details, thus reducing complexity. The general term given to this property of designs is ABSTRACTION, and a good abstraction is said to provide INFORMATION HIDING. Abstractions are needed because the human brain can process only limited amount information at one time. Procedures provide procedural abstraction. When using a certain procedure, a programmer does not need to worry about details or how it performs its computation. Data abstractions group together the pieces of data that describe some entity. So that programmers can manipulate that data as a unit.

Classes are data abstractions that contain procedural abstractions. The fewer public methods in each class, the better the abstraction; and the abstraction is increased even more if all methods are private.

Design Principle 5: Increase Reusability where possible

Strategies for increasing reusability:

1. Generalize the design as much as possible

2. Follow design principles 2, 3, 4. Increasing cohesion increases reusability since the component has a well-defined purpose. Reducing coupling increases reusability since the component can stand alone. Increasing abstraction increases reusability since abstractions are naturally more general.

3. Design the system to contain hooks. For example, methods may be explicitly designed to be overridden in subclasses. One of the barriers to reuse occurs when a component does most of what someone else needs, but not quite everything. If a component has effective hooks, then other people can easily extend it to what they want.

4. Simplify the design as much as possible. The more complex the component, the less it is likely to be reusable in novel contexts. The most reusable components are those that do one simple thing but do it very well. Basic Unix commands such as 'grep', 'cat', 'head', 'tail', 'uniq', 'awk' and 'red' are considered classic examples of reusable components because they are very powerful yet relatively simple. Their simplicity comes from the fact that they are input and output the same data type: streams or characters. Their power comes from the fact that they can be strung together in large variety of combinations.

Design Principle 6: Reuse existing design and code where possible

Design with reuse is complimentary to design for reusability. Actively reusing designs or code allows us to take advantage of the investment others or we have made in reusable components.

Cloning should NOT be seen as a form of reuse. Cloning means copying code from one place to another. Cloning should be avoided since when there are two or more occurrences of the same or similar code in the system, any change made (e.g. to fix defects) will have to be made in all classes.

Unfortunately maintenances are often not aware of all the clones that exist. And hence only make the change in one place. The bug thus remains even though maintenance thinks it's fixed.

Design Principle 7: Design for flexibility

Designing for flexibility means actively anticipating changes that a design may have to undergo in the future and preparing for them. Such changes might include changes in implementation (e.g. to improve efficiency or to handle larger volumes of data) or changes on functional requirements.

Ways to build flexibility into a design:

1. Reducing coupling and increasing cohesion: this allows us to more readily replace part of a system. For example, if our current application saves data to a file, we might anticipate that in future we will want to use a commercial database package. Placing the data-saving part of the system in a subsystem that has layer cohesion will greatly will greatly facilitate such a change.

2. Creating abstractions: In particular we should try to create interfaces or superclasses with polymorphic operations. Doing this allows new extensions to be easily added.

3. Not hand-coding anything: constants should be banished from the code.

4. leaving all options open: For example, when a method encounters an exception, it is best that the method throws an exception rather than taking a definite action handle it. The caller then has the flexibility to decide what to do with the exception.

5. Using reusable code and making code reusable: design principles 5,6 tend to make designs more flexible.

Design Principle 8: Obsolescence

Anticipation of obsolescence is a special case of design for flexibility. Changes will inevitably occur in the technology a software system uses and the environment in which it runs.

Some rules to follow:

1. Avoid using early release of technology: the immediate problem is that early releases are likely to have more defects than later releases. However, even if it is possible to work around a defect, a secondary problem may arise: if the provider of the technology fixes the defect in a subsequent release of the technology, the original work-around may no longer work. Even when no defect exists, improvements to the technology, which are especially likely in the first few releases, can render designs that use the technology in need of change. For example, early adopters of java in the 1995-99 timeframe were later required to make many changes to their code because some of the methods and classes used were DEPRECATED. The java designers decided components to be deprecated. When they develop improved designs - they do not intend to support the deprecated components indefinitely. So, users are forced to update their software.
2. Avoid using software libraries that are specific to particular environments

3. Avoid using undocumented features or little used features of software libraries. The little used features are not only more likely to have defects but, more importantly, the manufacturers may feel that little harm may be done if they are removed or changed. On the other hand, if the technology provider makes changes to heavily used features there will be loud protests from many users, so changes are less likely to be made.
4. Avoid using reusable software or special hardware from smaller companies, or from those that are less likely to provide long term support. A small company is more likely to go out of business or not to have the resources to support older versions.
5. Use standard languages and technologies that are supported by multiple users. Doing this gives you some confidence that important technology will not be **.

Design Principle 9: Design for portability

Principle	*Primary objective*
Anticipate obsolescence	Survival of software
Design for portability	The ability to have the software run on as many platforms as possible (sometimes necessary for survival)

An important guideline to achieving portability is to avoid the use of facilities that are specific to one particular environment. Some programming languages, such as Java, make this easy because the language itself is designed to allow software to run on different platforms unchanged.