Unit– II

Software Project Management: Software project management complexities, Responsibilities of a software project manager, Metrics for project size estimation, Project estimation techniques, Empirical Estimation techniques, COCOMO, Halstead's software science, risk management.

Requirements Analysis and Specification: Requirements gathering and analysis, Software Requirements Specification (SRS), Formal system specification, Axiomatic specification, Algebraic specification, Executable specification and 4GL.

Software Project Management

What is management?

management is achieving goals in a way that makes the best use of all resources

This involves the following activities:

Planning – It is the basic function of management. "Planning is deciding in advance - what to do, when to do & how to do. It bridges the gap from where we are & where we want to be".

"deciding what is to be done"

Organizing – It is the process of bringing together physical, financial and human resources and developing productive relationship amongst them for achievement of organizational goals.

"making arrangements"

Staffing – It is the function of manning the organization structure and keeping it manned. Manpower, training"selecting the right people for the job"

Directing – Supervision, Motivation, Leadership, Communication. "giving instructions"

Controlling –The purpose of controlling is to ensure that everything occurs in conformities with the standards.

Monitoring – checking on progress

Innovating – coming up with solutions when problems emerge

Representing – liaising with clients, users, developers and other stakeholder

What is a project?

- A **project** is a temporary effort to create a unique product or service. Projects usually include constraints and risks regarding cost, schedule or performance outcome.
- A **project** is Planned set of interrelated tasks to be executed over a fixed period and within certain cost and other limitations.

What is Project Management?

- **Project Management** is the discipline of planning, organizing, motivating, and controlling resources to achieve specific goals
- **Project management** is a methodical approach to planning and guiding **project** processes from start to finish.

What is Software Project Management (SPM)?

- ✓ **Software Project Management (SPM)** is all about planning, organizing, and observing the software development process to make sure the project is completed successfully, on time, and within budget.
- ✓ It involves tasks like planning, defining the project scope, calculating how much time and resources are needed, scheduling tasks, allocating resources, and tracking progress.
- ✓ The ultimate goal is to deliver a high-quality software product that meets the needs and expectations of the users.

SOFTWARE PROJECT MANAGEMENT COMPLEXITIES

Invisibility:Invisibility of software makes it diffcult to assess the progress of a project and is a major cause for the complexity of managing a software project.

Changeability:Frequent changes to the requirements and the invisibility of software are possibly the two major factors making software project management a complex task

Complexity: Due to the inherent complexity of the functioning of a software product in terms of the basic parts making up the software, many types of risks are associated with its development.

Uniqueness: Every software project is usually associated with many unique features or situations. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the ones he/she might have encountered in the past.

Exactness of the solution: the parameters of a function call in a program are required to be in complete conformity with the function definition. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult.

Team-oriented and intellect-intensive work: Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work.

RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

Job Responsibilities for Managing Software Projects

- A software project manager takes the overall responsibility of steering a project to success.
- The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations.
- Most managers take responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients.
- We can broadly classify a project manager's varied responsibilities into the following two major categories:
  - Project planning, and
  - Project monitoring and control.

Project planning:

- Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.
- The initial project plans are revised from time to time as the project progresses and more project data become available.
- Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.

Project monitoring and control:

- Project monitoring and control activities are undertaken once the development activities start.
- A project manager usually needs to change the plan to cope up with specific situations at hand.

Skills Necessary for Managing Software Projects

Three skills that are most critical to successful project management are the following:

- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

Note:-

Software project management techniques such as cost estimation, risk management, and configuration management, etc.,

project managers need good communication skills and the ability to get work done.

Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience.

PROJECT PLANNING

Initial project planning is undertaken and completed before any development activity starts.

During project planning, the project manager performs the following activities.

Estimation: The following project attributes are estimated.

- Cost: How much is it going to cost to develop the software product?
- Duration: How long is it going to take to develop the product?
- Effort: How much effort would be necessary to develop the product?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

Scheduling: After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.

Staffing: Staff organisation and staffing plans are made.

Risk management: This includes risk identification, analysis, and abatement planning.

Miscellaneous plans: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Fundamental parameter based on estimations and project plans

Size is the most fundamental parameter, based on which all other estimations and project plans are made.

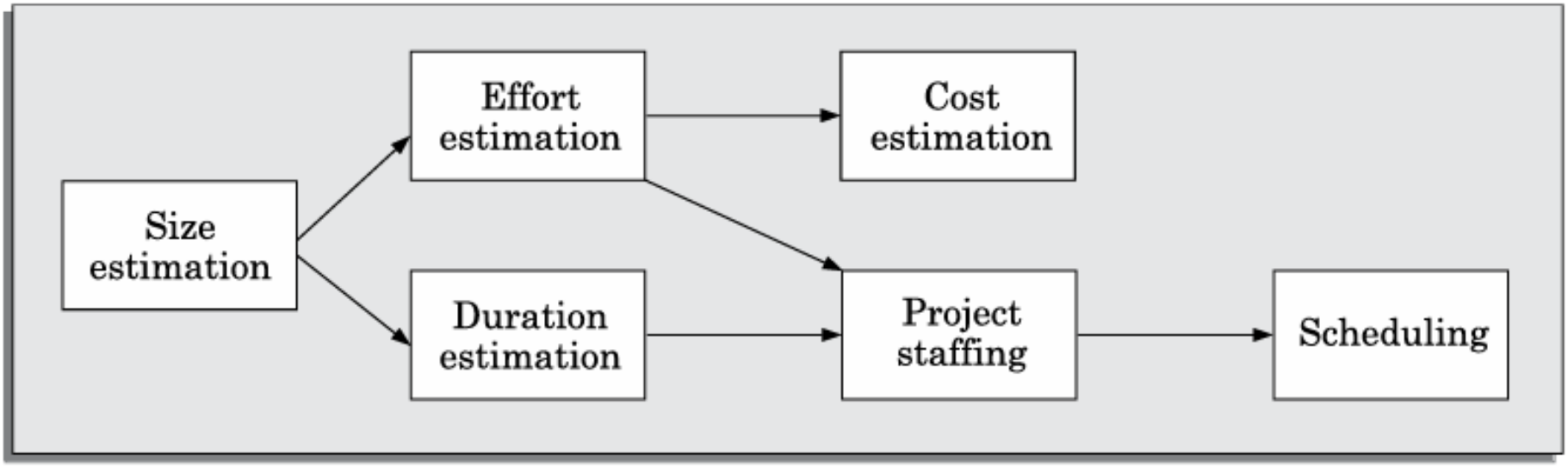


FIGURE 3.1 Precedence ordering among planning activities.

Sliding Window Planning

- Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as *sliding window planning*.
- At the start of a project, the project manager has incomplete knowledge about the nitty gritty of the project.
- His information base gradually improves as the project progresses through different development phases.
- The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear.
- The project parameters are re-estimated periodically as understanding grows and a periodically as project parameters change.
- By taking these developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence.

The SPMP Document of Project Planning

Once project planning is complete, project managers document their plans in a *software project management plan* (SPMP) document.

Organisation of the software project management plan (SPMP) document

1. **Introduction**
   - (a) Objectives
   - (b) Major Functions
   - (c) Performance Issues
   - (d) Management and Technical Constraints
2. **Project estimates**
   - (a) Historical Data Used
   - (b) Estimation Techniques Used
   - (c) Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   - (a) Work Breakdown Structure
   - (b) Task Network Representation
   - (c) Gantt Chart Representation
   - (d) PERT Chart Representation
4. **Project resources**
   - (a) People
   - (b) Hardware and Software
   - (c) Special Resources
5. **Staff organisation**
   - (a) Team Structure
   - (b) Management Reporting
6. **Risk management plan**
   - (a) Risk Analysis
   - (b) Risk Identification
   - (c) Risk Estimation
   - (d) Risk Abatement Procedures
7. **Project tracking and control plan**
   - (a) Metrics to be tracked
   - (b) Tracking plan
   - (c) Control plan
8. **Miscellaneous plans**
   - (a) Process Tailoring
   - (b) Quality Assurance Plan
   - (c) Configuration Management Plan
   - (d) Validation and Verification
   - (e) System Testing Plan
   - (f) Delivery, Installation, and Maintenance Plan

METRICS FOR PROJECT SIZE ESTIMATION

- The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.
- The size of a project is obviously not the number of bytes that the source code occupies, neither is it the size of the executable code.
- Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP).

Lines of Code (LOC)

- LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular.
- This metric measures the size of a project by counting the number of source instructions in the developed program.
- Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.
- Determining the LOC count at the end of a project is very simple.
- However, accurate estimation of LOC count at the beginning of a project is a very difficult task.
- One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work.
- The project manager divides the problem into modules, and each module into sub-modules andso on, until the LOC of the leaf-level modules are small enough to be predicted.

LOC metric has several shortcomings when used to measure problem size.

LOC is a measure of coding activity alone.

- ✓ A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, design, code, test, etc.) and not just the coding effort.
- ✓ LOC, however, focuses on the coding activity alone—it merely computes the number of source lines in the final program.

LOC count depends on the choice of specific instructions:

LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers.

By coding style, we mean the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used.

Different programmers may lay out their code in very different ways.

For example, one programmer might write several source instructions on a single line, whereas another might split a single instruction across several lines.

LOC measure correlates poorly with the quality and efficiency of the code:

- ❖ Larger code size does not necessarily imply better quality of code or higher efficiency.
- ❖ Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms.

LOC metric penalises use of higher-level programming languages and code reuse:

- ✓ A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower.
- ✓ This would show up as smaller program size, and in turn, would indicate lower effort! Thus, if managers use the LOC count to measure the effort put in by different developers (that is, their productivity), they would be discouraging code reuse by developers.
- ✓ Modern programming methods such as object-oriented programming and reuse of components makes the relationships between LOC and other project attributes even less precise.

LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities:

- Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic.
- To realise why this is so, imagine the effort that would be required to develop a program having multiple nested loops and decision constructs and compare that with another program having only sequential control f low.

It is very difficult to accurately estimate LOC of the final program from problem specification:

- ✓ The biggest shortcoming of the LOC metric is that the LOC count is very difficult to estimate during project planning stage and can only be accurately computed after the software development is complete.
- ✓ Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

Function Point (FP) Metric

- Function point metric was proposed by Albrecht and Gaffney in 1983.
- This metric overcomes many of the shortcomings of the LOC metric.
- Since its inception, function point metric has steadily gained popularity.
- Function point metric has several advantages over LOC metric.
- One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself.
- Using the LOC metric, on the other hand, the size can accurately be determined only after the code has been fully written.
- The size of a software product is directly dependent on the number of different high-level functions or features it supports.
- The function point metric by counting the number of input and output data items and the number of files accessed by the function.
- The implicit assumption made is that the more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function.

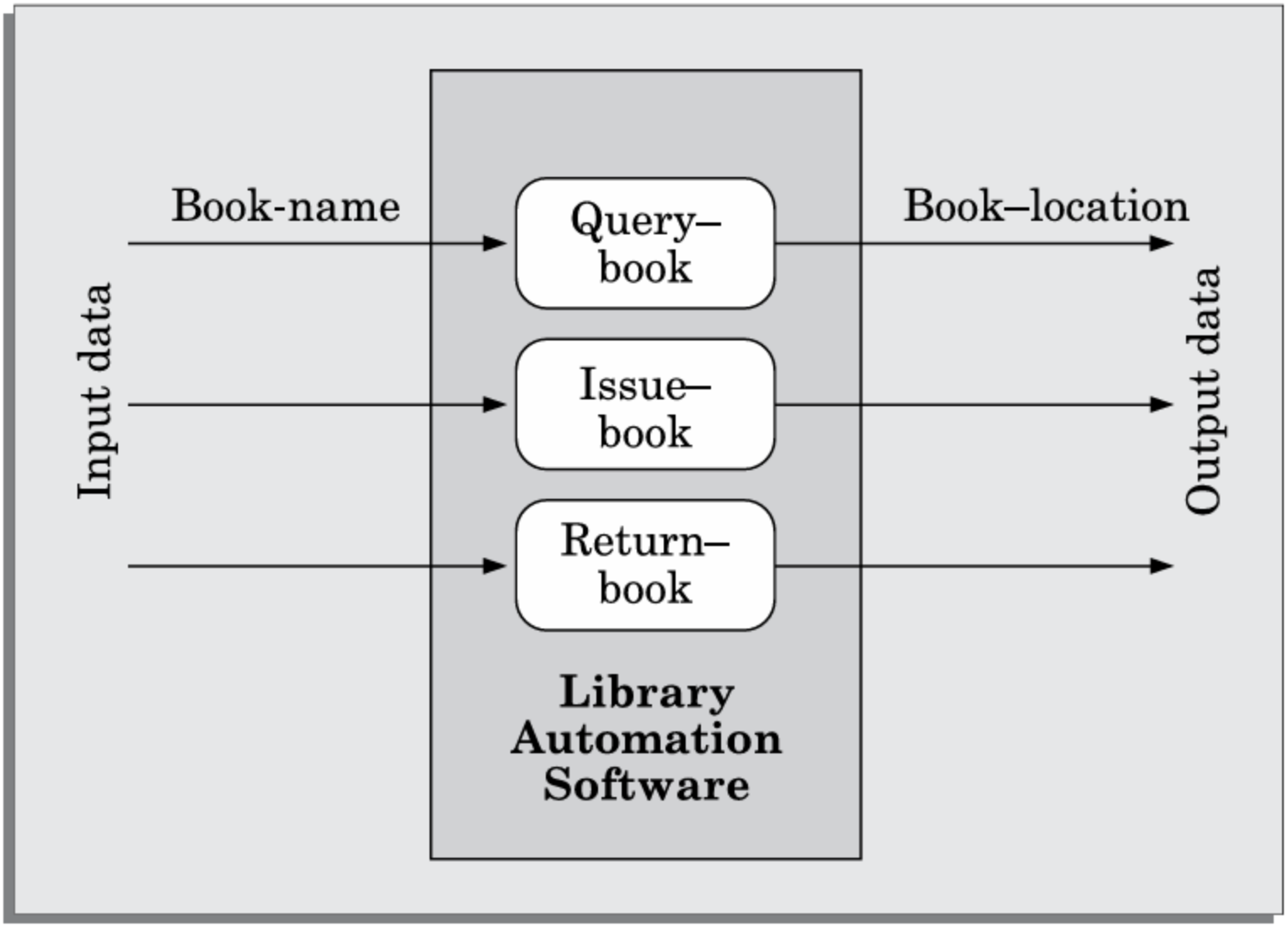


FIGURE 3.2 System function as a mapping of input data to output data.

- Albrecht postulated that in addition to the number of basic functions that a software performs, its size also depends on the number of files and the number of interfaces associated with it.
- Here, interfaces refer to the different mechanisms for data transfer with external systems including the interfaces with the user, interfaces with external computers, etc.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

Function Point (FP) Metric

- The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification.
- It is computed using the following three steps:

Step 1:Compute the Unadjusted Function Point (UFP) using a heuristic expression.

Step 2:Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.

Step 3:Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression.

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

The meanings of the different parameters.

1. Number of inputs: Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries.
2. Number of outputs: The outputs considered include reports printed, screen outputs, error messages produced, etc.
3. Number of inquiries: An inquiry is a user command (without any data input) and only requires some actions to be performed by the system.

Examples: print account balance, print all student grades, display rank holders' names, etc.

4. Number of files: The files referred to here are logical files. A logical file represents a group of logically related data
5. Number of interfaces: Here the interfaces denote the different mechanisms that are used to exchange information with other systems.

Examples: data files on tapes, disks, communication links with other systems, etc.

Step 2: Refine parameters

- UFP computed at the end of step 1 is a gross indicator of the problem size.
- This UFP needs to be refined by taking into account various peculiarities of the project.
- The complexity of each parameter is graded into three broad categories—simple, average, or complex.
- The weights for the different parameters are determined based on the numerical values shown in Table 3.1.
- Based on these weights of the parameters, the parameter values in the UFP are refined.
- For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

TABLE 3.1 Refinement of Function Point Entities

Type	*Simple*	*Average*	*Complex*
Input (I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

- In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2.
  - Examples: high transaction rates, response time requirements, scope for reuse, etc.
- Albrecht identified 14 parameters that can influence the development effort. The list of these parameters have been shown in Table 3.2.
- Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence).
- The resulting numbers are summed, yielding the total *Degree of Influence (DI)*.
- A *Technical Complexity Factor (TCF)* for the project is computed and the TCF is multiplied with UFP to yield FP.
- The TCF expresses the overall impact of the corresponding project parameters on the development effort.
  - TCF is computed as $(0.65 + 0.01 \times DI)$.
- As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49.
- Finally, FP is given as the product of UFP and TCF. That is, $FP = UFP \times TCF$.

Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations
Extent of change and focus on ease of use

PROJECT ESTIMATION TECHNIQUES

- The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost.
- A large number of estimation techniques have been proposed by researchers.
- These can broadly be classified into three main categories:

1. Empirical estimation techniques
2. Heuristic techniques
3. Analytical estimation techniques

Empirical Estimation Techniques

- Empirical estimation techniques are essentially based on making an educated guess of the project parameters.
- We shall discuss two such formalisations of the basic empirical estimation techniques:

1. Expert Judgement
2. Delphi Cost Estimation

Heuristic Techniques

- Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions.
- Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression.
- Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

$$\text{Estimated Parameter} = c_1 \times e^{d_1}$$

$$\text{Estimated Resource} = c_1 \times p^{d_1} + c_2 \times p^{d_2} + \cdots$$

- *Estimated Parameter is the dependent parameter (to be estimated).*
- *The dependent parameter to be estimated could be effort, project duration, staff size, etc.,*
- *c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data).*

Analytical Estimation Techniques

- ✓ Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project.
- ✓ Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis.
- ✓ example of an analytical technique is Halstead's software science.
- ✓ Halstead's software science is especially useful for estimating software maintenance efforts.
- ✓ In fact, it outperforms both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned

Expert Judgement

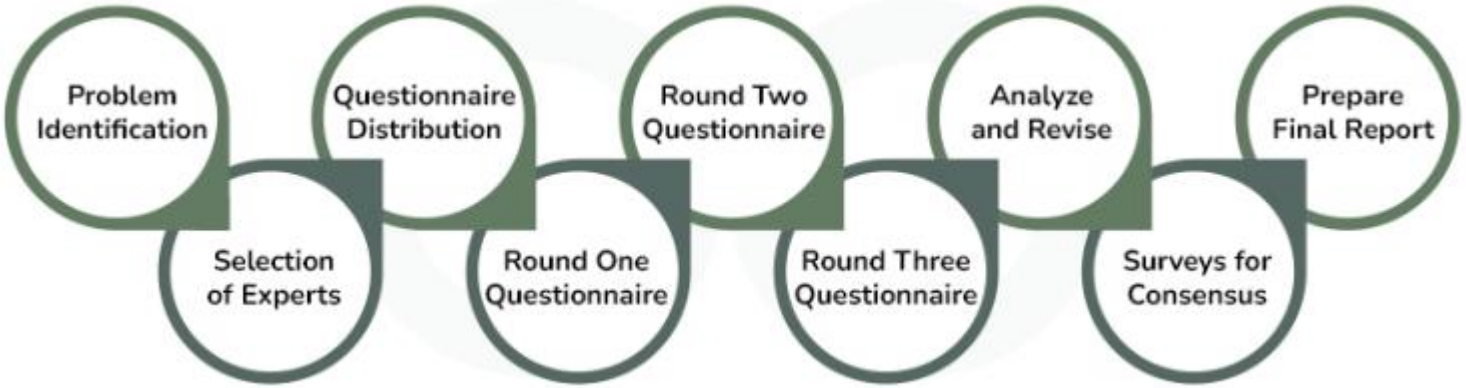
- Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly.
- The expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate.
- This technique suffers from several shortcomings.
  - The outcome of the expert judgement technique is subject to human errors and individual bias.
  - It is possible that an expert may overlook some factors inadvertently.
  - An expert making an estimate may not have relevant experience and knowledge of all aspects of a project.

Example:

- He may be conversant with the database and user interface parts, but may not be very knowledgeable about the computer communication part.
- Due to these factors, the size estimation arrived at by the judgement of a single expert may be far from being accurate.

Delphi Cost Estimation

- Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach.
- Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator.
- In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate.
- Estimators complete their individual estimates anonymously and submit them to the co-ordinator.



- In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations.
- The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators.
- The prepared summary information is distributed to the estimators.
- Based on this summary, the estimators re-estimate.
- This process is iterated for several rounds.
- However, no discussions among the estimators is allowed during the entire estimation process.
- The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior.
- After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate.
- The Delphi estimation, though consumes more time and effort, overcomes an important shortcoming of the expert judgement technique in that the results cannot unjustly be influenced by overly assertive and senior members.

COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

- ✓ COnstructive COst estimation MOdel (COCOMO) was proposed by Boehm [1981]. COCOMO prescribes a three stage process for project estimation.
- ✓ In the first stage, an initial estimate is arrived and Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate.
- ✓ COCOMO uses both single and multivariable estimation models at different stages of estimation.
- ✓ The three stages of COCOMO estimation technique are:
  - ✓ Basic COCOMO
  - ✓ Intermediate COCOMO, and
  - ✓ Complete COCOMO.

Three basic classes of software development projects

Organic:
Semidetached:
Embedded:

Aspects	Organic	Semidetached	Embedded
Project Size	2 to 50 KLOC	50-300 KLOC	300 and above KLOC
Complexity	Low	Medium	High
Team Experience	Highly experienced	Some experienced as well as inexperienced staff	Mixed experience, includes experts
Environment	Flexible, fewer constraints	Somewhat flexible, moderate constraints	Highly rigorous, strict requirements
Effort Equation	$E = 2.4(400)^{1.05}$	$E = 3.0(400)^{1.12}$	$E = 3.6(400)^{1.20}$
Example	Simple payroll system	New system interfacing with existing systems	Flight control software

Basic COCOMO Model

- The basic COCOMO model is a single variable heuristic model that gives an approximate estimate of the project parameters.
- The basic COCOMO estimation model is given by expressions of the following

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ months}$$

where,

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- a_1, a_2, b_1, b_2 are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in months.
- Effort is the total effort required to develop the software product, expressed in person-months (PMs).

Software Projects	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Intermediate COCOMO

- ✓ The basic COCOMO model assumes that effort and development time are functions of the product size alone.
- ✓ The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down.
    - ✓ Example:

      If modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1.
- ✓ Boehm requires the project manager to rate 15 different parameters for a particular project on a scale of one to three.
- ✓ For each such grading of a project parameter, he has suggested appropriate cost drivers (or multipliers) to refine the initial estimates.

In general, the cost drivers identified by Boehm can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

Development environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

$E = a*(KLOC)^b * EAF\ PM$

$Tdev = c*(E)^d$

Where,

- *E is effort applied in Person-Months*
- *KLOC is the estimated size of the software product indicate in Kilo Lines of Code*
- *EAF is the Effort Adjustment Factor (EAF) is a multiplier used to refine the effort estimate obtained from the basic COCOMO model.*
- *Tdev is the development time in months*
- *a, b, c are constants determined by the category of software project given in below table.*

Software Projects	a	b	c	d
Organic	3.2	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Complete COCOMO

- Detailed COCOMO goes beyond Basic and Intermediate COCOMO by diving deeper into project-specific factors.
- It considers a wider range of parameters, like team experience, development practices, and software complexity.
- By analyzing these factors in more detail, Detailed COCOMO provides a highly accurate estimation of effort, time, and cost for software projects.
- It's like zooming in on a project's unique characteristics to get a clearer picture of what it will take to complete it successfully.

HALSTEAD'S SOFTWARE SCIENCE—AN ANALYTICAL TECHNIQUE

- Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products.
- Halstead used a few primitive program parameters to develop the expressions for overall program length, potential minimum volume, actual volume, language level, effort, and development time.

Measure	Symbol	Formula
Program length	N	$N = N_1 + N_2$
Program vocabulary	N	$n = n_1 + n_2$
Volume	V	$V = N * (\log_2 n)$
Difficulty	D	$D = (n_1 / 2) * (N_2 / 2)$
Effort	E	$E = D * V$

Program length: The length of a program is total usage of operators and operands in the program.

Program vocabulary: The Program vocabulary is the number of unique operators and operands used in the program.

Program Volume: The Program Volume can be defined as minimum number of bits needed to encode the program.

Guideline for calculating operands and operators:

1. All the variables and constants are considered as operands.
2. Local variables with same name, if occurring in different functions are counted as unique operand.
3. Function calls are considered as operators.
4. The looping statements, do … while, while, for, are operators. The statements if, if … else, are operators. The switch … case statements are considered as operators.
5. The reserve words, returns, default, continue, break, sizeof are all operators.
6. The brackets, commas, semicolons, are operators.
7. The unary and binary operators are considered as operators. The & is considered as operator.
8. In arrays, array name and index are considered as operands and [ ] is considered as operator.
9. All hash directives can be ignored.
10. Comments are not considered.
11. In Goto statement, goto is considered as operator and label as operand

Example: Obtain Halstead's length and volume measure for following C function.

```
Void swap (int a[ ], int i)
{
int temp;
Temp = a[i];
a[i] = a[i+1];
a[i+1] = temp;
}
```

Operands	Occurrences	Operators	Occurrences
swap	1	( )	1
a	5	{ }	1
i	5	void	1
temp	3	int	3
1	2	[ ]	5
		,	1
		;	4
		=	3
		+	2
$n_1 = 5$	$N_1 = 16$	$n_2 = 9$	$N_2 = 21$

$N = N_1 + N_2$

$= 16 + 21 = 37$

$N = 37$

$n = n_1 + n_2$

$= 5 + 9 = 14$

$n = 14$

Estimated length = $n_1 \log n_1 + n_2 \log n_2$

$= 5 \log 5 + 9 \log 9$

$= 5 * 2.32 + 9 * 2.19 = 31.37$

Estimated length = 31.37

Volume = N * log n

$= 37 * \log (14)$

$37 * 2.63 = 97.64$

Volume (V) = 97.64

REQUIREMENTS ANALYSIS AND SPECIFICATION

What are the main activities carried out during requirements analysis and specification phase?

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis
- Requirements specification

REQUIREMENTS GATHERING AND ANALYSIS

- The complete set of requirements are almost never available in the form of a single document from the customer.
- The requirements have to be systematically gathered by the analyst from several sources in bits and pieces.
- We can conceptually divide the requirements gathering and analysis activity into two separate tasks:
  - Requirements gathering
  - Requirements analysis

Requirements Gathering

- Requirements gathering activity is also popularly known as requirements elicitation.
- The primary objective of the requirements gathering task is to collect the requirements from the stakeholders.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly is concerned with the software.

In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1. Studying existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the analyst. Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, and the broad category of features required.
2. Interview: Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software.
3. Task analysis: The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a ***task***.

4. Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations.
5. Form analysis: Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.

Requirements Analysis

- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.
- The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

1. What is the problem?
2. Why is it important to solve the problem?
3. What exactly are the data input to the system and what exactly are the data output by the system?
4. What are the possible procedures that need to be followed to solve the problem?
5. What are the likely complexities that might arise while solving the problem?

During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly:

It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

Inconsistency:

Two requirements are said to be inconsistent, if one of the requirements contradicts the other.

Incompleteness:

- An incomplete set of requirements is one in which some requirements have been overlooked.
- The lack of these features would be felt by the customer much later, possibly while using the software.
- Incompleteness is caused by the inability of the customer to visualise the system that is to be developed and to anticipate all the features that would be required.
- An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise the requirements in the form of an SRS document.

Users of SRS Document

Usually, many different people need the SRS document for very different purposes.

Users, customers, and marketing personnel:

These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs.

Software developers:

The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.

Test engineers:

The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working.

User documentation writers:

The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.

Project managers:

The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.

Maintenance engineers:

The SRS document helps the maintenance engineers to understand the functionalities supported by the system.

Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work.

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.

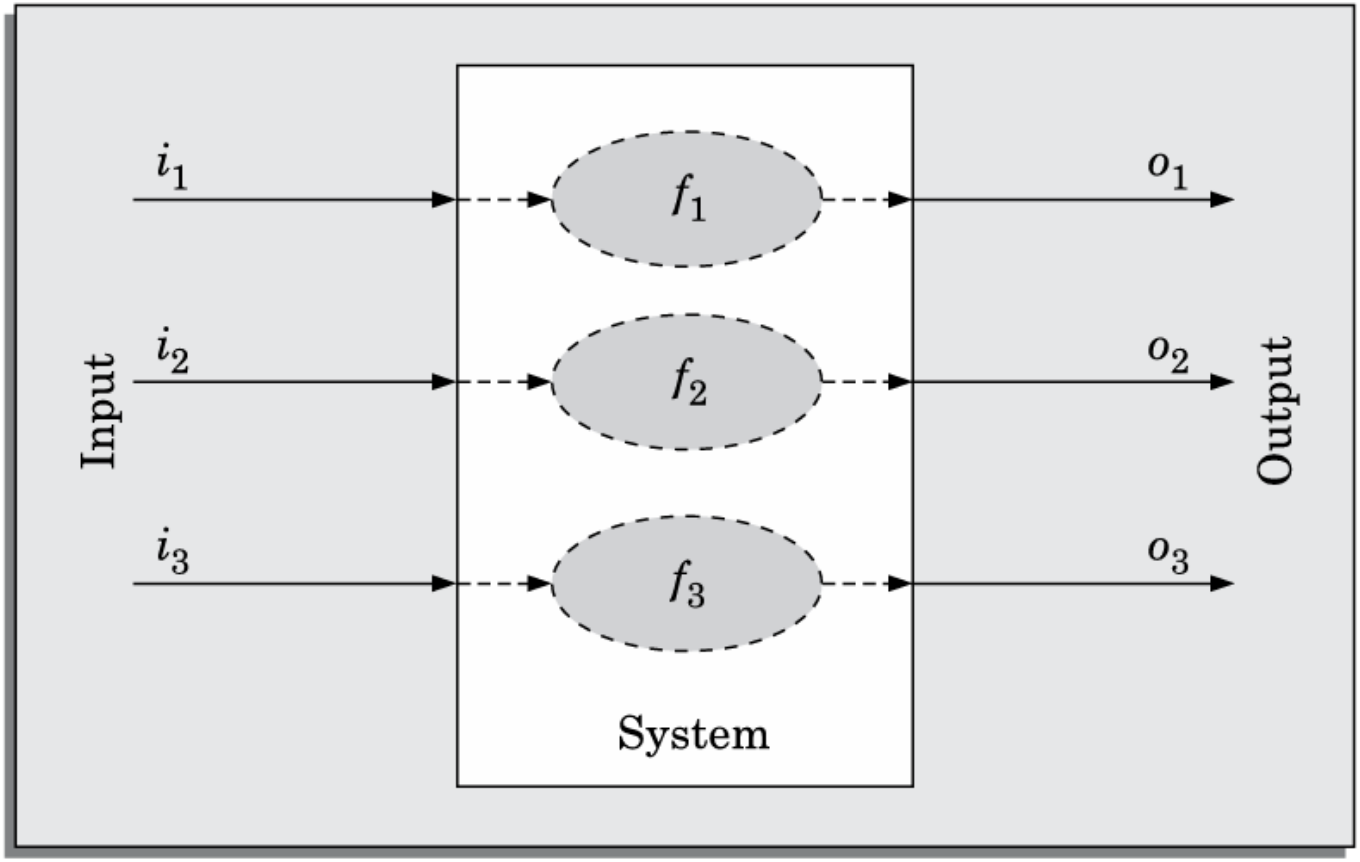
Provides a baseline for validation and verification: The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements.

Characteristics of a Good SRS Document

- The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects.
- IEEE Recommended Practice for Software Requirements Specifications [IEEE, 1998] describes the content and qualities of a good software requirements specification (SRS).
- Some of the identified desirable qualities of an SRS document are the following:
- Concise: The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document

Implementation-independent: The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. The SRS document should describe the system (to be developed) as a black box, and should specify only the externally visible behaviour of the system. For this reason, the SRS document is also called the *black-box specification* of the software being developed.



4.1 The black-box view of a system as performing a set of functions.

Traceable: It should be possible to trace a specific requirement to the design elements that implement it and vice versa. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyse the impact of changing a requirement on the design elements and the code.

Modifiable: Customers frequently change the requirements during the software development due to a variety of reasons.

Identification of response to undesired events: The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable.

Attributes of Bad SRS Documents

Over-specification: It occurs when the analyst tries to address the "how to" aspects in the SRS document. Over-specification restricts the freedom of the designers in arriving at a good design solution.

Forward references: One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.

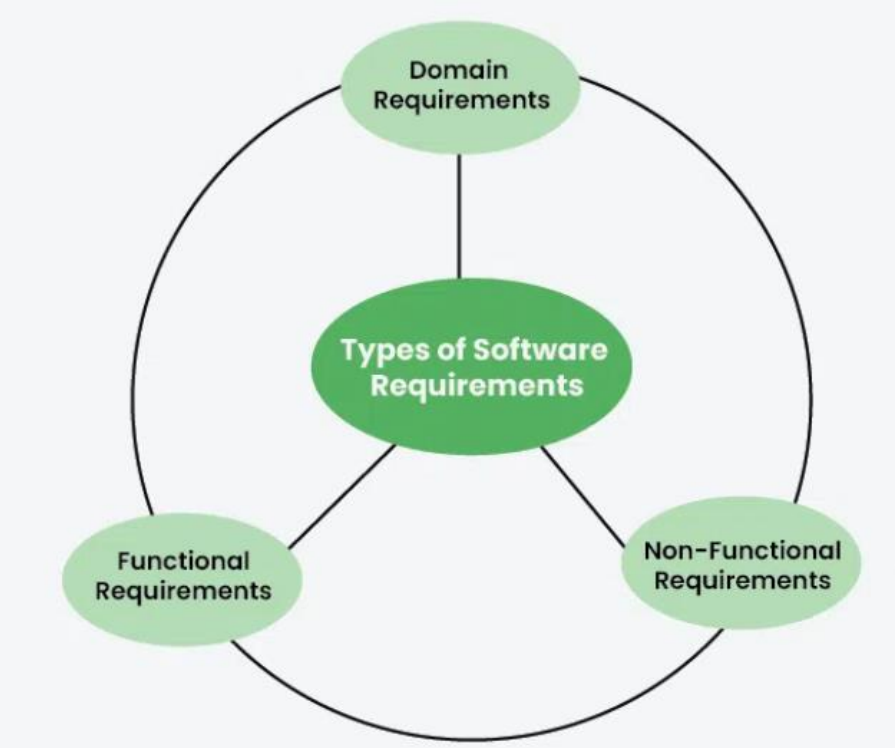
Wishful thinking: This type of problems concern description of aspects which would be difficult to implement.

Noise: The term noise refers to presence of material not directly relevant to the software development process. For example, in the register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8 am and 5 pm, 7 days a week. This information can be called noise as it would hardly be of any use to the software developers and would unnecessarily clutter the SRS document, diverting the attention from the crucial points.

Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE 830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following:

- Functional requirements
- Non-functional requirements
  - *Design and implementation constraints*
  - *External interfaces required*
  - *Other non-functional requirements*



1. Functional Requirements

Definition: Functional requirements describe what the software should do. They define the functions or features that the system must have.

Examples:

- **User Authentication**: The system must allow users to log in using a username and password.
- **Search Functionality**: The software should enable users to search for products by name or category.
- **Report Generation**: The system should be able to generate sales reports for a specified date range.

Explanation: Functional requirements specify the actions that the software needs to perform. These are the basic features and functionalities that users expect from the software.

2. Non-functional Requirements

Definition: Non-functional requirements describe how the software performs a task rather than what it should do. They define the quality attributes, performance criteria, and constraints.

Examples:

- **Performance**: The system should process 1,000 transactions per second.
- **Usability**: The software should be easy to use and have a user-friendly interface.
- **Reliability**: The system must have 99.9% uptime.
- **Security**: Data must be encrypted during transmission and storage.

Explanation: Non-functional requirements are about the system's behavior, quality, and constraints. They ensure that the software meets certain standards of performance, usability, reliability, and security.

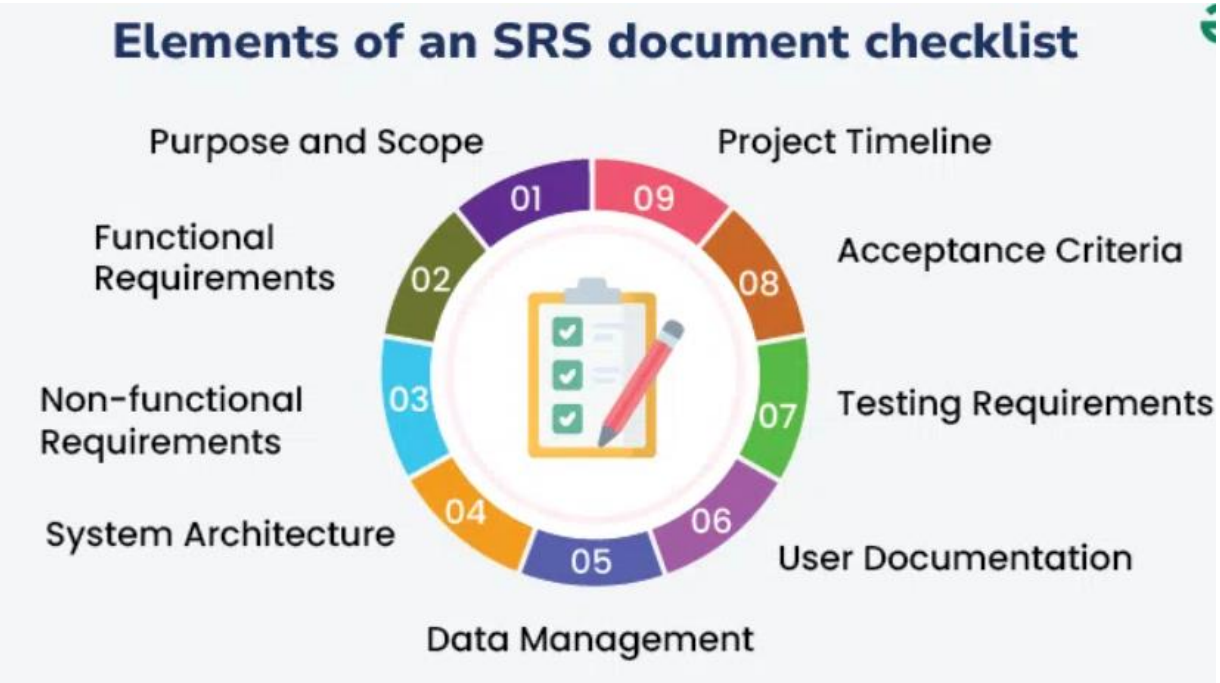
Functional Requirements

VS

Non Functional Requirements

- **Describes** what the system should do, i.e., specific functionality or tasks.
- **Focuses** on the behavior and features of the system.
- **Defines** the actions and operations of the system.
- **User authentication** data input/output, transaction processing

- **Describes** how the system should perform, i.e., system attributes or quality.
- **Focuses** on the performance, usability, and other quality attributes.
- **Defines** constraints or conditions under which the system must operate
- **Scalability** security, response time, reliability, maintainability.


Elements of an SRS document checklist
Purpose and Scope
01
Functional Requirements
02
Non-functional Requirements
03
System Architecture
04
Data Management
05
User Documentation
06
Testing Requirements
07
Acceptance Criteria
08
Project Timeline
09


SRS document checklist
Correctness
01
Ambiguity
02
Completeness
03
Consistency
04
Verifiability
05
Traceability
06
Feasibility
07

FORMAL SYSTEM SPECIFICATION

- In recent years, formal techniques have emerged as a central issue in software engineering.
- The importance of precise specification, modelling, and verification is recognised to be important in most engineering disciplines.
- Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented.
- The specification of a system can be given either as a list of its desirable properties (property-oriented approach) or as an abstract model of the system (model-oriented approach).

What is a Formal Technique?

- A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc.
- The mathematical basis of a formal method is provided by its specification language.
- More precisely, a formal specification language consists of two sets—*syn* and *sem*, and a relation *sat* between them.
- The set *syn* is called the *syntactic domain*, the set *sem* is called the *semantic domain*, and the relation *sat* is called the *satisfaction relation*.
- For a given specification *syn*, and model of the system *sem*, if *sat* (*syn, sem*), then *syn* is said to be the *specification* of *sem*, and *sem* is said to be the *specification* of *syn*.

- The generally accepted paradigm for system development is through a hierarchy of abstractions.
- Each stage in this hierarchy is an implementation of its preceding stage and a specification of the succeeding stage.
- The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc.
- Formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.

Syntactic domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

Semantic domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronisation trees, partial orders, state machines, etc.

Satisfaction relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined—those that *preserve* a system's behaviour and those that *preserve* a system's structure.

AXIOMATIC SPECIFICATION

- In axiomatic specification of a system, first-order logic is used to write the pre- and post- conditions to specify the operations of the system in the form of axioms.
- The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked.
- The pre-conditions capture the requirements on the input parameters of a function.
- The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully.
- Thus, the post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

How to develop an axiomatic specifications?

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Establish the constraints on the input parameters as a predicate.
- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Combine all of the above into pre- and post-conditions of the function.

We now illustrate how simple abstract data types can be algebraically specified through two simple examples.

EXAMPLE 4.11 Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f(x : real) : real$$

$$pre : x \in R$$

$$post : \{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$$

ALGEBRAIC SPECIFICATION

- In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type.
- It was first brought into prominence by Guttag [1980, 1985] in specification of abstract data types.
- Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.
- An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the *signature* of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

Equations section: This section gives a set of *rewrite rules* (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

- The first step in defining an algebraic specification is to identify the set of required operators.
- After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators.
- The definition of these categories of operators is as follows:

Basic construction operators: These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, 'create' and 'append' are basic construction operators.

Extra construction operators: These are the construction operators other than the basic construction operators. For example, the operator 'remove' is an extra construction operator, because even without using 'remove' it is possible to generate all values of the type being specified.

Basic inspection operators: These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified—these are the inspection operators. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1.

Extra inspection operators: These are the inspection operators that are not basic inspectors. A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

Completeness: This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. When the equations are not complete, at some step during the reduction process, we might not be able to reduce the expression arrived at that step by using any of the equations. There is no simple procedure to ensure that an algebraic specification is complete.

Finite termination property: This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.

Unique termination property: This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked—Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique termination property is a very difficult problem.

EXECUTABLE SPECIFICATION AND 4GL

- When the specification of a system is expressed formally or is described by using a programming language, then it becomes possible to directly execute the specification without having to design and write code for implementation.
- Executable specifications are usually slow and inefficient, 4GLs (4th Generation Languages) are examples of executable specification languages.
- 4GLs are successful because there is a lot of large granularity commonality across data processing applications which have been identified and mapped to program code.
- 4GLs get their power from software reuse, where the common abstractions have been identified and parameterized.
- Careful experiments have shown that rewriting 4GL programs in 3GLs results in up to 50 per cent lower memory usage and the program execution time can reduce up to tenfolds.