# UNIT 2:

## System engineering:

Systems engineering is an interdisciplinary field of engineering and engineering management that focuses on how to design and manage complex systems over their life cycles.

S/W Engineering occurs as a consequence of a process called System engineering. System Engineering focuses on analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information.

## Computer Based Systems

The elements of computer-based systems are defined as software, hardware, people, database, documentation, and procedures.

**Software**: programs, data structures, and related work products.

**Hardware**: electronic devices that provide computing capabilities.

**People**: Users and operators of hardware and software.

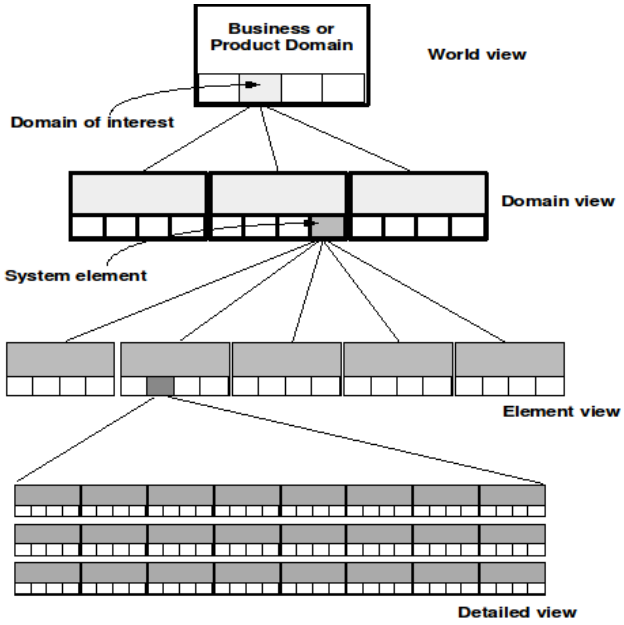
**Database**: A large, organized collection of information that is accessed via S/w and persists over time.

**Documentation**: manuals, on-line help files.

**Procedures**: the steps that define the specific use of each system element.

## The System Engineering Hierarchy

The key to system engineering is a clear understanding of context. For software development this means creating a "world view" and progressively narrowing its focus until all technical detail is known.




The system engineering Process usually begins with a “world view.” The entire business or product domain is examined to ensure that the proper business or technology context can be established.

The world view is refined to focus more fully on a specific domain of interest.

Within a specific domain, the need for targeted system elements (data, S/W, H/W, and people) is analyzed.

Finally, the analysis, design, and construction of a targeted system element are initiated.

**Business Process Engineering:**

The goal of Business Process Engineering (BPE) is to define architectures that will enable a business to use information effectively.

BPE is one process for creating an overall plan for implementing the computing architecture.

Three different architectures must be analyzed and designed within the context of business objectives and goals:

- Data architecture
- Application architecture
- Technology infrastructure

The data architecture provides a framework for the information needs of a business. The building blocks of the architecture are the data objects that are used by the business.

Once a set of data objects is defined, their relationships are identified. A relationship indicates how objects are connected to one another.

The application architecture encompasses those elements of a system that transform objects within the data architecture for some business purpose.

The technology infrastructure provides the foundation for the data and application architectures. The infrastructure encompasses the h/w and s/w that are used to support the applications and data.

**Check Hierarchy from Class copy**

**Product Engineering:**

Emphasize that software engineers participate in all levels of the product engineering process that begins with requirements engineering. The analysis step maps requirements into representations of data, function, and behavior. The design step maps the analysis model into data, architectural, interface, and software component designs.

Product engineering translates the customer's desire for a set of defined capabilities into a working product

It achieves this goal by establishing a product architecture and a support infrastructure

1. Product architecture components consist of people, hardware, software, and data
2. Support infrastructure includes the technology required to tie the components together and the information to support the components.

3. Requirements engineering elicits the requirements from the customer and allocates function and behavior to each of the four components.

   System component engineering happens next as a set of concurrent activities that address each of the components separately

a. Each component takes a domain-specific view but maintains communication with the other domains
b. The actual activities of the engineering discipline takes on an element view

   Analysis modeling allocates requirements into function, data, and behavior

   Design modeling maps the analysis model into data/class, architectural, interface, and component design.

**Check Hierarchy from Class copy**

**System Modeling:**

Defines the processes (e.g., domain classes in OO terminology) that serve the needs of the view under consideration

Represents the behavior of the processes and the assumptions on which the behavior is based

Explicitly defines intra-level and inter-level input that form links between entities in the model

Represents all linkages (including output) that will enable the engineer to better understand the view

Factors to Consider when Constructing a Model:

1. Assumptions that reduce the number of possible permutations and variations, thus enabling a model reflect the problem in a reasonable manner.

2. Simplifications that enable the model to be created in a timely manner.

3. Limitations that help to bound the system.

4. Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented.

5. Preferences that indicate the preferred architecture for all data, functions, and technology.

**Requirements Engineering**

Requirements engineering (RE) refers to the process of defining, documenting and maintaining requirements and to the sub fields of systems engineering and software engineering concerned with this process.

The process of collecting the software requirement from the client then understand, evaluate and document it is called as requirement engineering.

Requirement engineering constructs a bridge for design and construction.

**Requirement engineering consists of seven different tasks as follow:**

**1. Inception**

- Inception is a task where the requirement engineering asks a set of questions to establish a software process.
- In this task, it understands the problem and evaluates with the proper solution.
- It collaborates with the relationship between the customer and the developer.
- The developer and customer decide the overall scope and the nature of the question.

**2. Elicitation**

Elicitation means to find the requirements from anybody.
The requirements are difficult because the **following problems occur in elicitation**.
**Problem of scope:** The customer give the unnecessary technical detail rather than clarity of the overall system objective.
**Problem of understanding:** Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.
**Problem of volatility:** In this problem, the requirements change from time to time and it is difficult while developing the project.

**3. Elaboration**

- In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.
- Its main task is developing pure model of software using functions, feature and constraints of a software.

**4. Negotiation**

- In negotiation task, a software engineer decides the how will the project be achieved with limited business resources.
- To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

**5. Specification**

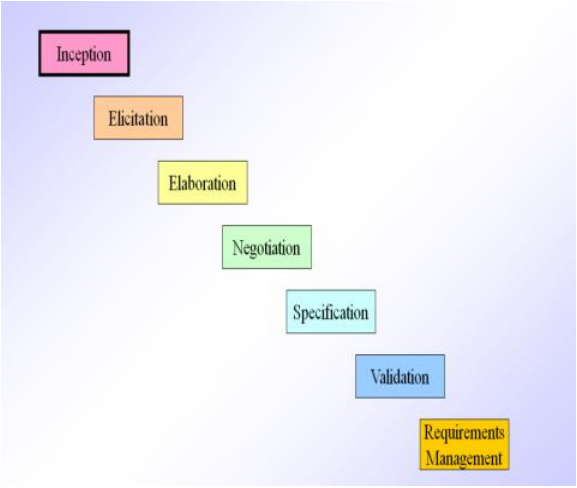
- In this task, the requirement engineer constructs a final work product.
- The work product is in the form of software requirement specification.
- In this task, formalize the requirement of the proposed software such as informative, functional and behavioral.

**6. Validation**

- The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.
- The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

**7. Requirement management**

- It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project.
- These tasks start with the identification and assign a unique identifier to each of the requirement.
- After finalizing the requirement traceability table is developed.
- The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement.



Software Requirements Characteristics:

Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct and well-defined.

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

**Software Requirements Types:**

Broadly software requirements should be categorized in two categories:

**Functional Requirements**

Requirements, which are related to functional aspect of software fall into this category.

They define functions and functionality within and from the software system.

examples -

- Search option given to user to search from various invoices.
- Users can be divided into groups and groups can be given separate rights.

**Non-Functional Requirements**

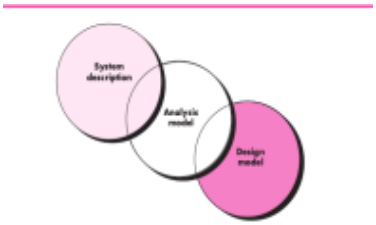
Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements include -

- Security
- Logging
- Storage
- Configuration
- Performance
- Cost
- Interoperability
- Flexibility
- Disaster recovery
- Accessibility

**Requirements Analysis:**

- **Software engineering task bridging the gap between system requirements engineering and software design.**




**Analysis Objectives:**

- Identify customer's needs.
- Evaluate system for feasibility.
- Perform economic and technical analysis.
- Allocate functions to system elements.
- Establish schedule and constraints.
- Create system definitions.

**Communication Principles**

**Principle 1.** ***Listen.*** Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. *Never* become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

**Principle 2.** ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

**Principle 3.** ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.

**Principle 4.** ***Face-to-face communication is best.*** But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

**Principle 5.** ***Take notes and document decisions.*** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

**Principle 6.** ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

**Principle 7.** ***Stay focused; modularize your discussion.*** The more people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved

**Principle 8.** ***If something is unclear, draw a picture.*** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

**Principle 9.** ***(a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on.*** C

Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

**Principle 10.** ***Negotiation is not a contest or a game. It works best when both parties win.*** There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

Planning Principles-

**Principle 1.** ***Understand the scope of the project.*** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

**Principle 2.** ***Involve stakeholders in the planning activity.*** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

**Principle 3.** ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

**Principle 4.** ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

**Principle 5.** ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

**Principle 6.** ***Be realistic.*** People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

**Principle 7.** ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project

time line moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require high granularity (too much can change).

**Principle 8.** ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality. If technical reviews3 are to be conducted, they should be scheduled. If pair programming is to be used during construction, it should be explicitly defined within the plan.

**Principle 9.** ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

**Principle 10.** ***Track the plan frequently and make adjustments as required.***

Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

**Modeling Principles-**

Principle 1. *The primary goal of the software team is to build software, not create models.* Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

Principle 2. *Travel light—don't create more models than you need.* Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on

construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

Principle 3. *Strive to produce the simplest model that will describe the problem or the software.* Don't overbuild the software [Amb02b]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to

maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. *Build models in a way that makes them amenable to change.* Assume that your models will change, but in making this assumption don't get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.

**Principle 5.** ***Be able to state an explicit purpose for each model that***

***is created.*** Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

**Principle 6.** ***Adapt the models you develop to the system at hand.*** It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

**Principle 7.** ***Try to build useful models, but forget about building perfect***

***models.*** When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth

the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.

**Principle 8.** ***Don't become dogmatic about the syntax of the model. If***

***it communicates content successfully, representation is secondary.***

Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

**Principle 9.** ***If your instincts tell you a model isn't right even though it***

***seems okay on paper, you probably have reason to be concerned.*** If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though

you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

**Principle 10.** ***Get feedback as soon as you can.*** Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

***Facilitated application specification technique*** **(FAST).**

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

• Meetings are conducted and attended by both software engineers and other

stakeholders.

• Rules for preparation and participation are established.

• An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.

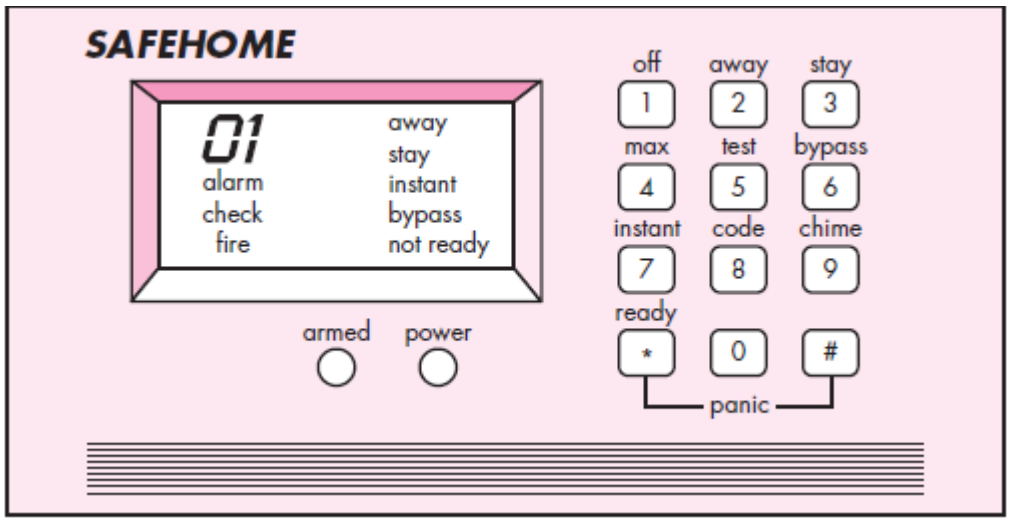
• A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.

• A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur, I present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting. While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business

rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system. Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating,

must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented). The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board, at an internal website, or posed in a chat room environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be modified, and additions can be made. At this stage, critique and debate are strictly prohibited. After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built. In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for entries on the lists.11 Each mini-specification is an elaboration of an object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be: The control panel is a wall-mounted unit that is approximately 9 _ 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 _ 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions. The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.




**Quality Function Deployment**

*Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software

QFD identifies three types of requirements-

**Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

**Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

**Exciting requirements.** These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.