UNIT II

Software Engineering Principles: SE Principles, Communication Principles, Planning Principles, Modeling Principles, Construction Principles, Deployment.

System Engineering: Computer-based Systems, The System Engineering Hierarchy, Business Process Engineering, Product Engineering, System Modeling.

Requirements Engineering: A Bridge to Design and Construction, Requirements Engineering Tasks, Initiating Requirements Engineering Process, Eliciting Requirements, Developing Use-Cases, Building the Analysis Model, Negotiating Requirements, Validating Requirements.

SOFTWARE ENGINEERING PRINCIPLES

Software engineering is guided by a collection of core principles that help in the application of a software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the practice level, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

General principles that span software engineering process and practice: (1) provide value to end users, (2) keep it simple, (3) maintain the vision (of the product and the project), (4) recognize that others consume (and must understand) what you produce, (5) be open to the future, (6) plan ahead for reuse, and (7) think! To provide a more detailed look at the core principles that guide process and practice, we consider:

Principles That Guide Process: The following set of core principles can be applied to the framework, and by extension, to every software process.

1. ***Be agile:*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.
2. ***Focus on quality at every step:*** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.
3. ***Be ready to adapt:*** Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.
4. ***Build an effective team:*** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.
5. ***Establish mechanisms for communication and coordination:*** Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.
6. ***Manage change:*** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.
7. ***Assess risk:*** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.
8. ***Create work products that provide value for others:*** Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design; the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

Principles That Guide Practice: Software engineering practice has a single overriding goal—to deliver on-time, high quality, operational software that contains functions and features that meet the needs of all stakeholders. To achieve this goal, you should adopt a set of core principles that guide your technical work. The following sets of core principles are fundamental to the practice of software engineering:

1. ***Divide and conquer:*** Analysis and design should always emphasize *separation of concerns* (SoC). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.
2. ***Understand the use of abstraction:*** An abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. In software engineering practice, you use many different levels of abstraction. In analysis and design work, a software team normally begins with models that represent high levels of abstraction and slowly refines those models into lower levels of abstraction. The intent of an abstraction is to eliminate the need to communicate details. Without an understanding of the details, the cause of a problem cannot be easily diagnosed.
3. ***Strive for consistency:*** The principle of consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.
4. ***Focus on the transfer of information:*** Software is about information transfer—from a database to an end user, from an OS to an application etc. In every case, information flows across an interface, and as a consequence, there are opportunities for error, or omission, or ambiguity. The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.
5. ***Build software that exhibits effective modularity:*** Any complex system can be divided into modules (components), but good software engineering practice demands more. Modularity must be *effective* i.e., each module should focus exclusively on one aspect of the system (cohesion). Modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules.
6. ***Look for patterns:*** The goal of patterns within the software community is to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions.
7. ***When possible, represent the problem and its solution from a number of different perspectives:*** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.
8. ***Remember that someone will maintain the software:*** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

Communication Principles: Before customer requirements can be analyzed, modeled, or specified they must be gathered through the communication activity. Communication activity helps you to define your overall goals and objectives. Effective communication is among the most challenging activities that you will confront. The communication principles include:

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.

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. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.
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 that other principles are followed.
4. ***Face-to-face communication is best:*** It usually works better when some other representation of the relevant information is present.
5. ***Take notes and document decisions:*** Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.
6. ***Strive for collaboration:*** Collaboration occurs when the collective knowledge of members of the team is used to describe product or system functions or features. Each collaboration serves to build trust among team members and creates a common goal for the team.
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.
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.
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.*** Communication takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion and that "moving on" is sometimes the best way to achieve communication agility.
10. ***Negotiation is not a contest or a game. It works best when both parties win:*** There are many instances in which stakeholders must negotiate functions and features, priorities, and delivery dates. If team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

Planning Principles: The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives. There are many different planning philosophies. Some people are "minimalists" arguing that change often obviates the need for a detailed plan. Others are "traditionalists" arguing that the plan provides an effective road map and the more detail it has, the less likely the team will become lost. Still others are "agilists" arguing that a quick planning may be necessary. Regardless of the rigor with which planning is conducted, following principles always apply:

1. ***Understand the scope of the project:*** Scope provides the software team with a destination.
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.
3. ***Recognize that planning is iterative:*** As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. Iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.
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.
5. ***Consider risk as you define the plan:*** If you have identified risks that have high impact and high probability, contingency planning is necessary. The project plan should be adjusted to accommodate the likelihood that one or more of these risks will occur.

6. ***Be realistic:*** People don't work 100 percent of every day. Change will occur. Even the best software engineers make mistakes. These realities should be considered as a project plan is established.
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. 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.
8. ***Define how you intend to ensure quality:*** The plan should identify how the software team intends to ensure quality. If technical reviews 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.
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?
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: We create models to gain a better understanding of the actual entity to be built. In software engineering work, two classes of models can be created: requirements models and design models. ***Requirements models*** (also called *analysis models*) represent customer requirements by depicting the software in three different domains: *the information domain, the functional domain, and the behavioral domain*.
Design models represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.
The modeling principles include:

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.
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. Create only those models that make it easier and faster to construct the software.
3. ***Strive to produce the simplest model that will describe the problem or the software:*** Don't overbuild the software. 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).
4. ***Build models in a way that makes them amenable to change:*** Assume that your models will change. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design that will invariably miss important functions and features.
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 justification for existence of model, don't spend time on it.
6. ***Adapt the models you develop to the system at hand:*** It may be necessary to adapt model rules to the application.
7. ***Try to build useful models, but not building perfect models:*** Modeling should be conducted with an eye to next software engineering steps. Iterating endlessly to make a model "perfect" does not serve need for agility.
8. ***Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary:*** The most important characteristic of the model is to communicate information

that enables next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

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. If something tells you that a model is doomed to fail, you have reason to spend additional time examining it or developing a different one.
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.

Requirements Modeling Principles: All analysis methods are related by a set of operational principles:

1. ***The information domain of a problem must be represented and understood:*** The *information domain* encompasses the data that flow into the system, the data that flow out of the system, and the data stores that collect and organize persistent data objects.
2. ***The functions that the software performs must be defined:*** Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. Functions can be described at many different levels of abstraction.
3. ***The behavior of the software must be represented:*** The behavior of software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.
4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion:*** Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into sub problems until each sub problem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns,* and it is a key strategy in requirements modeling.
5. ***The analysis task should move from essential information toward implementation detail:*** The "essence" of the problem is described without any consideration of how a solution will be implemented. Implementation detail indicates how the essence will be implemented.

Design Modeling Principles: The design model created for software provides a variety of different views of the system. Set of design principles that can be applied are:

1. ***Design should be traceable to the requirements model:*** The design model translates the information from requirements model into architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.
2. ***Always consider the architecture of the system to be built:*** Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established component-level issues should be considered.
3. ***Design of data is as important as design of processing functions:*** A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.
4. ***Interfaces must be designed with care:*** A well-designed interface makes integration easier and assists the tester in validating component functions.
5. ***User interface design should be tuned to the needs of the end user. However, it should stress ease of use:*** The user interface is the visible manifestation of the software. A poor interface design often leads to the perception that the software is "bad."

6. ***Component-level design should be functionally independent:*** Functional independence is a measure of "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or sub-function.
7. ***Components should be loosely coupled to one another and to the external environment:*** Coupling is achieved in many ways. As level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of software decreases. Therefore, component coupling should be kept as low as is reasonable.
8. ***Design representations (models) should be easily understandable:*** If the design is difficult to understand, it will not serve as an effective communication medium.
9. ***The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity:*** Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

Construction Principles: The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. The following set of fundamental principles and concepts are applicable to coding and testing:

Coding Principles: The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

Preparation principles: ***Before you write one line of code, be sure you***

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets needs of the software to be built and environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: ***As you begin writing code, be sure you***

- Constrain your algorithms by following structured programming practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards. Write code that is self-documenting.
- Create a visual layout that aids understanding.

Validation Principles: ***After you've completed your first coding pass, be sure you***

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

Testing Principles: Testing is a process of executing a program with the intent of finding an error. If testing is conducted successfully, it will uncover errors in the software. Set of testing principles include**:**

1. ***All tests should be traceable to customer requirements:*** The objective of software testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

2. ***Tests should be planned long before testing begins:*** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
3. ***The Pareto principle applies to software testing:*** In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem is to isolate these suspect components and to thoroughly test them.
4. ***Testing should begin "in the small" and progress toward testing "in the large":*** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. ***Exhaustive testing is not possible:*** It is impossible to execute every combination of paths during testing. It is possible to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Deployment Principles: The deployment activity encompasses three actions: delivery, support, and feedback. A number of key principles should be followed as the team prepares to deliver an increment:

1. ***Customer expectations for the software must be managed:*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. So, a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided).
2. ***A complete delivery package should be assembled and tested:*** All relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in as many different computing configurations as possible.
3. ***A support regime must be established before the software is delivered:*** An end user expects responsiveness and accurate information when a question or problem arises. If support is worse, nonexistent, customer will become dissatisfied immediately. Support should be planned, support materials should be prepared.
4. ***Appropriate instructional materials must be provided to end users.*** The software team delivers more than the software itself. Appropriate training aids should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.
5. ***Buggy software should be fixed first, delivered later.*** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

The delivered software provides benefit for the end user, but it also provides useful feedback for the software team. As the increment is put into use, end users should be encouraged to comment on features and functions, ease of use, reliability, and any other characteristics that are appropriate.

SYSTEM ENGINEERING

Computer-Based Systems: All complex systems can be viewed as being composed of cooperating subsystems. A computer-based system makes use of a variety of system elements.

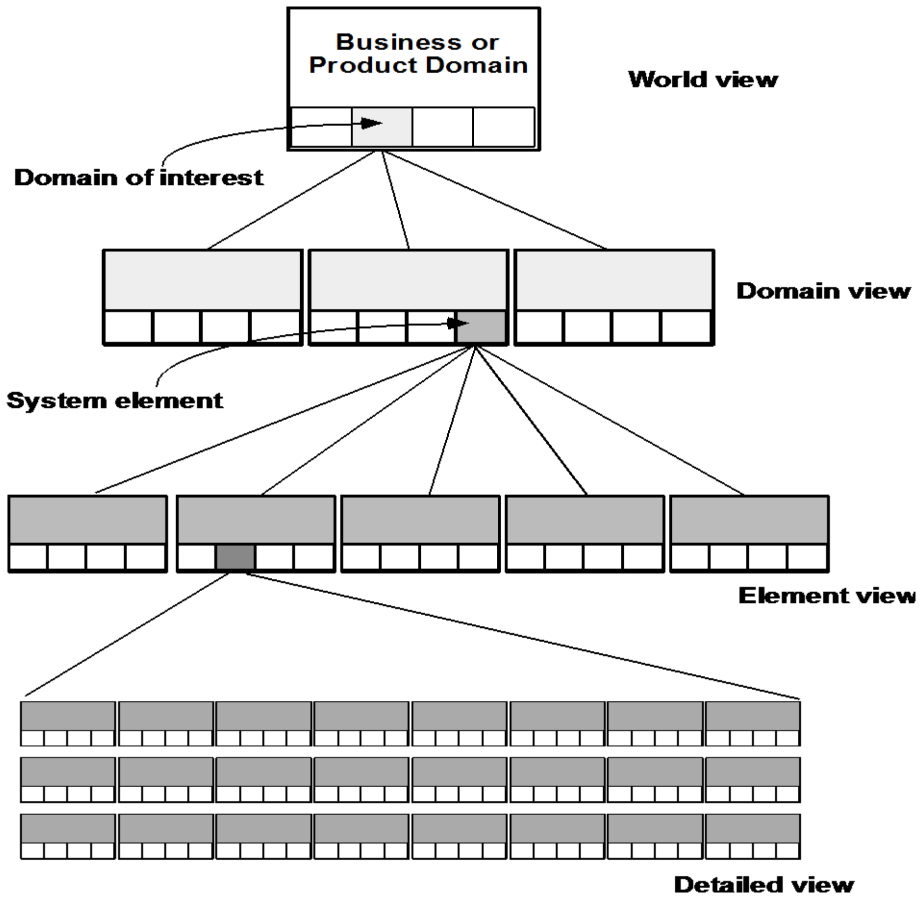
1. **Software**: programs, data structures, and related work products.
2. **Hardware**: electronic devices that provide computing capabilities.
3. **People**: Users and operators of hardware and software.
4. **Database**: A large, organized collection of information that is accessed via S/w and persists over time.
5. **Documentation**: manuals, on-line help files.
6. **Procedures**: the steps that define the specific use of each system element.

One complicating characteristic of computer-based system is that the elements constituting one system may also represent one macro element of a still large system. The *micro-element* is a computer-based system that is one part of a larger computer based system.

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.

In software engineering there is rarely one right way of doing something. Instead designers must consider the tradeoffs present in the feasible solutions and select one that seems advantageous for the current problem. This section lists several factors that need to be examined by software engineers when evaluating alternative solutions (assumptions, simplifications, limitations, constraints, and preferences).

Regardless of its domain of focus, system eng. Encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated below:



The system eng. 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.

System Modeling: System modeling is an important element of the system eng. Process. The Engineer creates models that:

1. Define the processes that serve the needs of the view under consideration.
2. Represent the behavior of the processes and the assumptions on which the behavior is based.
3. Explicitly define both exogenous and endogenous input to the model.

   Exogenous inputs link one constituent of a given view with other constituents at the same level of other levels; endogenous input links individual components of a constituent at a particular view.
4. Represent all linkages (including output) that will enable the engineer to better understand the view.

To construct a system model, the engineers should consider a number of restraining factors:

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.

Objective: Objective is a general statement of direction.
Goal: Goal defines a measurable objective: “reduce manufactured cost of our product”.
✓ Objectives tend to be strategic while goals tend to be tactical.

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.

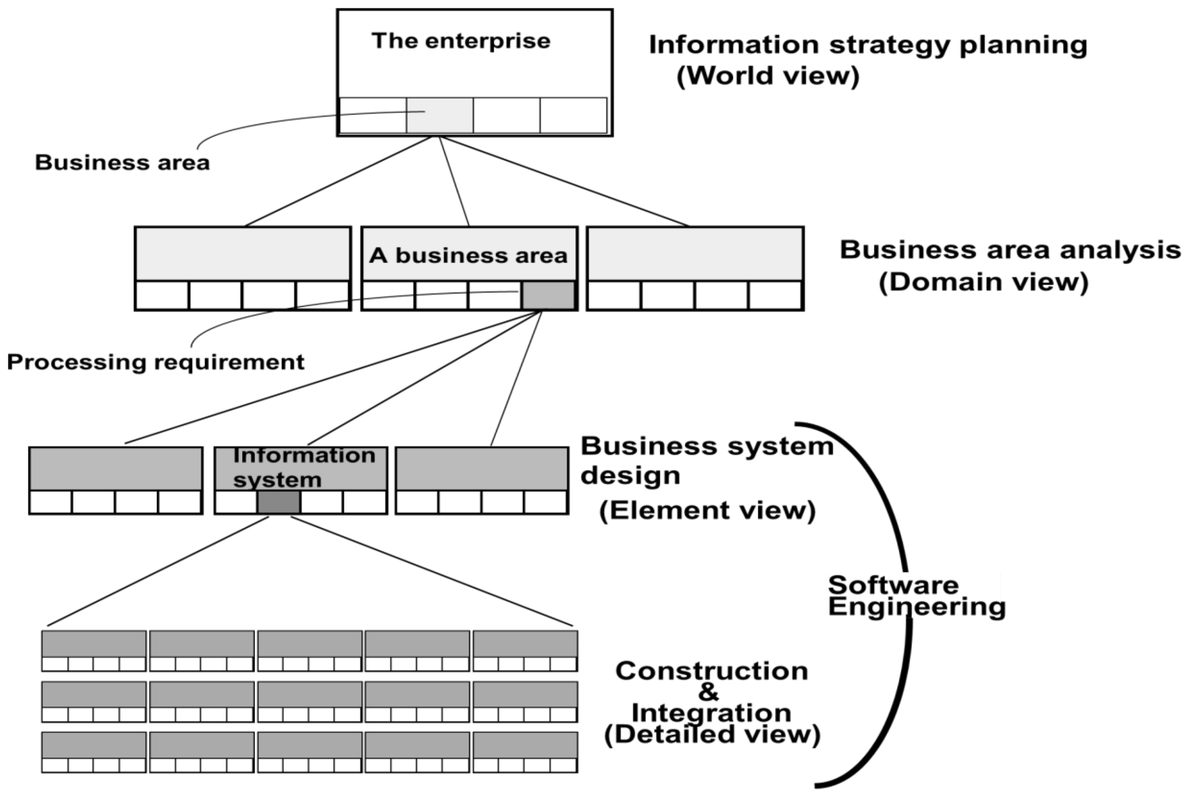
BPE uses an integrated set of procedures, methods, and tools to identify how information systems can best meet the strategic goals of an enterprise. It focuses first on the enterprise and then on the business area. BPE Creates enterprise models, data models and process models. It also creates a framework for better information management distribution, and control.

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

1. ***Data architecture:*** 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.
2. ***Application architecture:*** The *application architecture* encompasses those elements of a system that transform objects within the data architecture for some business purpose.
3. ***Technology infrastructure:*** The *technology infrastructure* provides the foundation for the data and application architectures. The infrastructure encompasses h/w and s/w that are used to support the applications and data.

The BPE Hierarchy: The BPE hierarchy includes four elements. They are:

1. ***Information strategy planning (ISP):*** The strategic goals are defined, success factors/business rules identified, enterprise model created.
2. ***Business area analysis (BAA):*** All processes/services modeled, interrelationships of processes and data.
3. ***Application Engineering*** (Software Engineering): It involves modeling applications/procedures that address (BAA) and constraints of ISP.
4. ***Construction and delivery:*** Done by using CASE and 4GTs, testing.


The enterprise
Information strategy planning (World view)
Business area
A business area
Business area analysis (Domain view)
Processing requirement
Information system
Business system design (Element view)
Software Engineering
Construction & Integration (Detailed view)

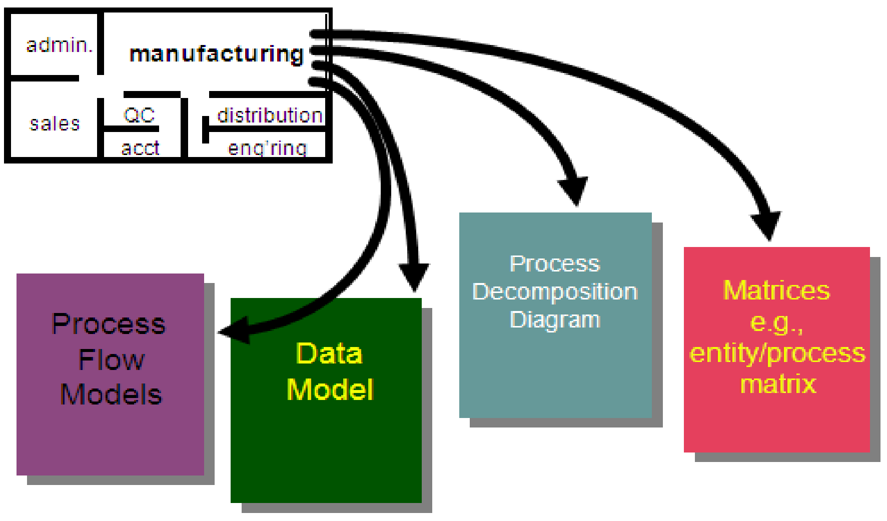
Information Strategy Planning:

- ***Management Issues:***
  - define strategic business goals/objectives
  - isolate critical success factors
  - conduct analysis of technology impact
  - perform analysis of strategic systems
- ***Technical Issues:***
  - create a top-level data model
  - cluster by business/organizational area
  - refine model and clustering

Business Area Analysis: It defines "naturally cohesive groupings of business functions and data"

- ❖ Perform many of the same activities as ISP, but narrow scope to individual business area
- ❖ Identify existing (old) information systems / determine compatibility with new ISP model
  - define systems that are problematic
  - defining systems that are incompatible with new information model
  - begin to establish re-engineering priorities

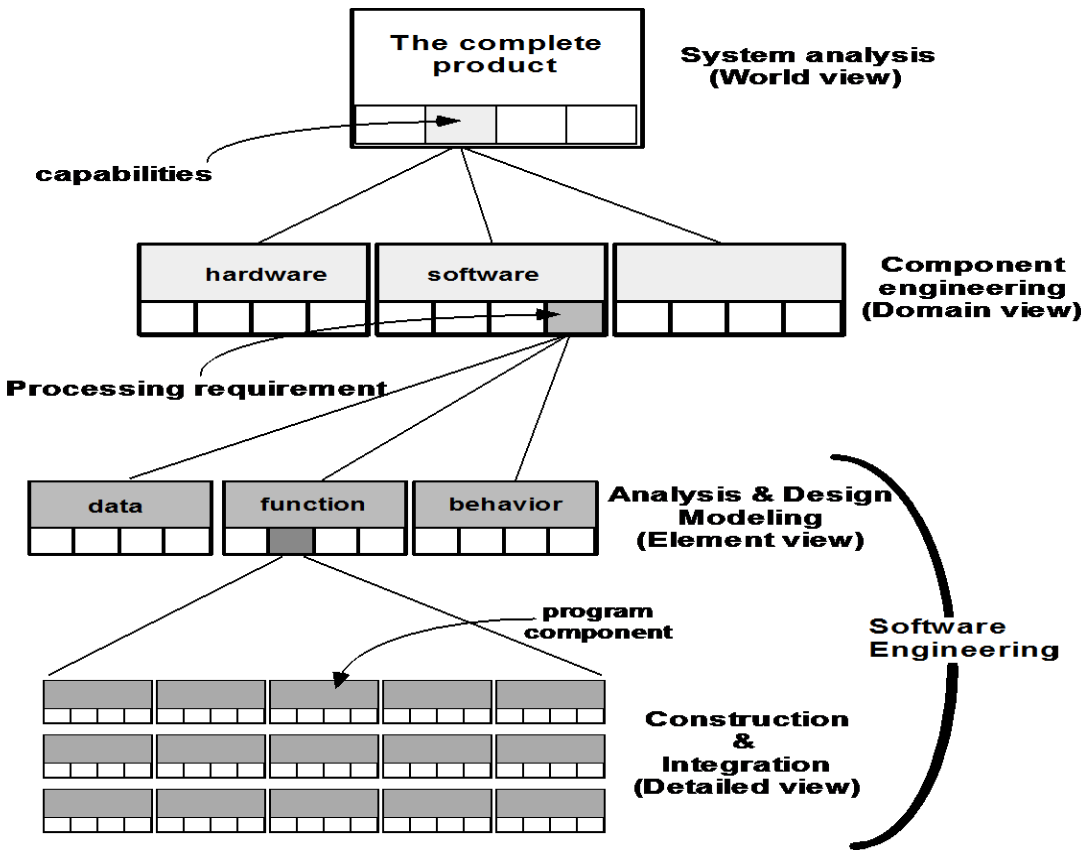
The Business Area Analysis Process:



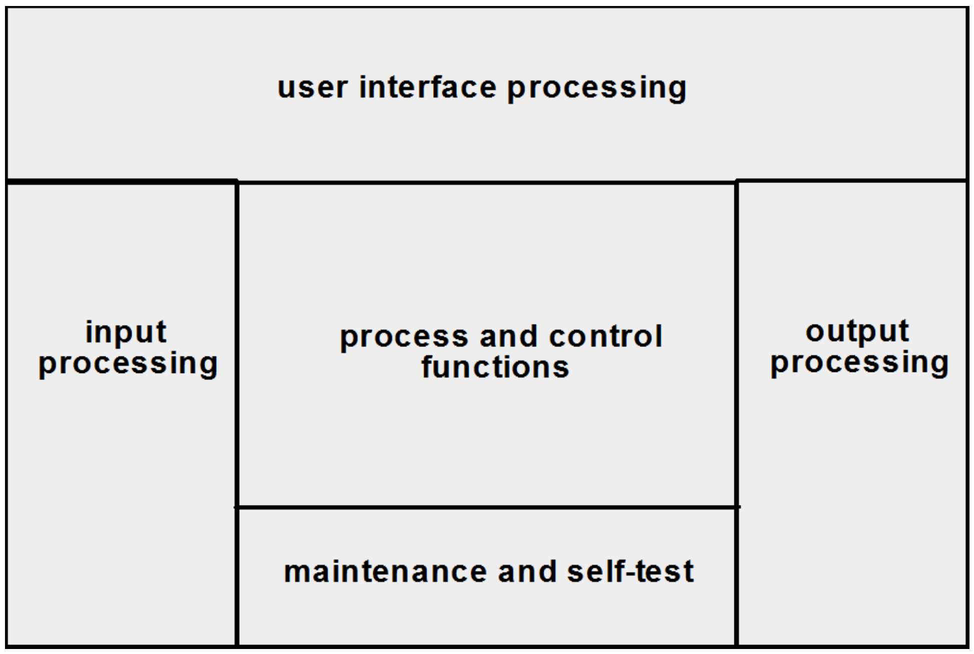
Product Engineering: The goal of product engineering is to translate customer's desire into a working product. It consists of four system components.

- Software
- Hardware
- Data
- People

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.

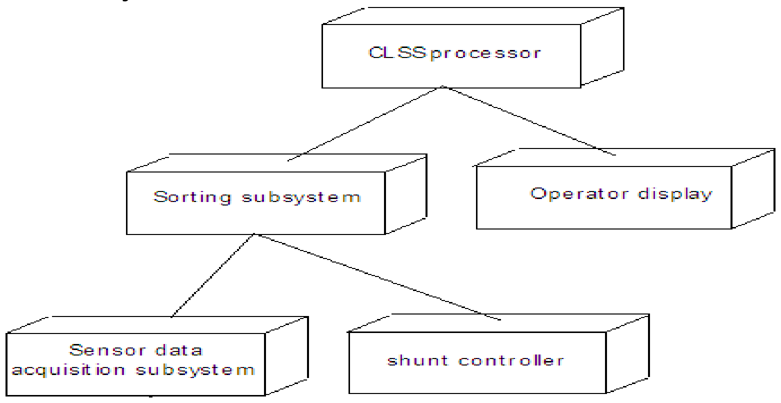

The complete product
System analysis (World view)
capabilities
hardware
software
Component engineering (Domain view)
Processing requirement
data
function
behavior
Analysis & Design Modeling (Element view)
program component
Software Engineering
Construction & Integration (Detailed view)

Product Architecture Template: Proposed by Hatley and Pirbhai, also known as Hatley-Pirbhai modeling.

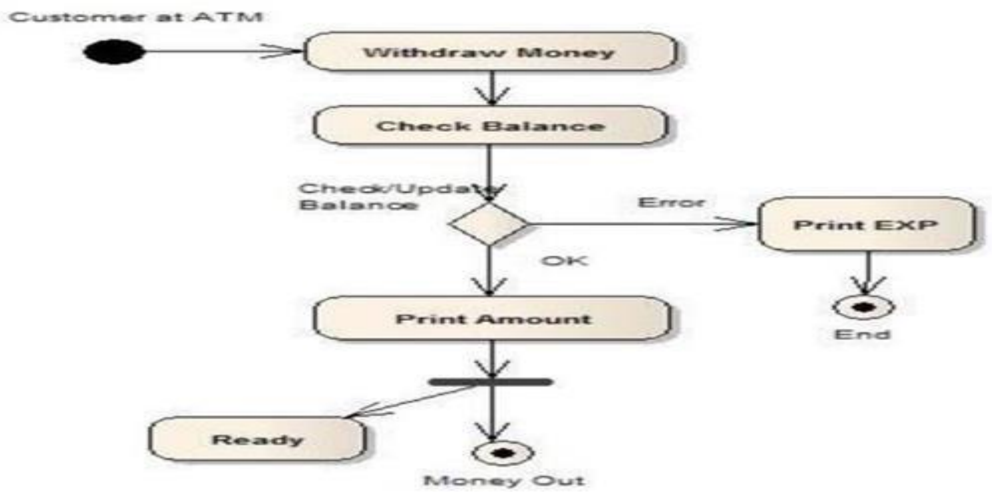

user interface processing
input processing
process and control functions
output processing
maintenance and self-test

System Modeling with UML: In represents the data that describe the element and the operations that manipulate the data.

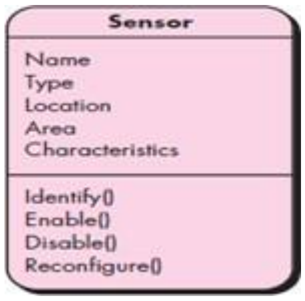
1. ***Deployment diagrams (Modeling hardware):*** Each Cube (3-D box) depicts a hardware element that is part of the physical architecture of the system



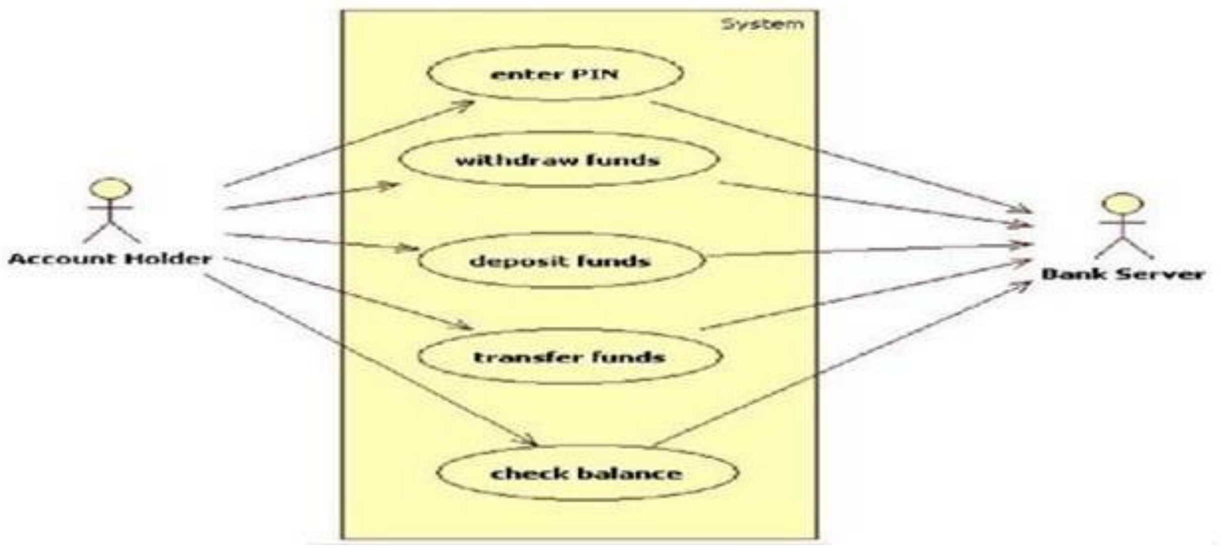
2. ***Activity diagrams (Modeling software):*** Represent procedural aspects of a system element



3. ***Class diagrams (Modeling data):*** Represent system level elements in terms of the data that describe the element and the operations that manipulate the data



4. ***Use-case diagrams (Modeling people):*** Illustrate the manner in which an actor interacts with the system



REQUIREMENTS ENGINEERING

A BRIDGE TO DESIGN AND CONSTRUCTION:

Requirement engineering, like all other Software Engineering activities, must be adapted to the process, project, product and the people doing the work. Requirement Engineering begins during the communication activity and continues into the modeling activity. It is essential that the software team make a real effort to understand the requirements of a problem before the team atoms to solve the problem.

Requirement Engineering builds a bridge to design and construction. It allows a software team, to examine,

1) About the context of the software work to be performed.
2) The specific needs that design the construction must address.
3) The priorities that guide the order in which work is to be completed.
4) The information, functions and behaviors that will have a profound impact on the resultant design.

REQUIREMENTS ENGINEERING TASKS

Requirement Engineering Provides an appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification and managing requirements. Requirement engineering tasks are classified as:

1. Inception
2. Elicitation
3. Elaboration
4. Negotiation
5. Specification
6. Validation
7. Requirements management

1) **Inception:** In some cases, for casual conversation all that is needed is to precipitate in a major software engineering effort. At project Inception, Software Engineers ask a set of questions, which establish:
   - ✓ Basic understanding of the problem
   - ✓ People who want a solution.
   - ✓ Nature of solution that is desired.
   - ✓ Effectiveness of preliminary communication and collaboration between customer and developer.

2) **Elicitation:** Elicitation is about asking the customers, uses and others "what they want", i.e., what the objectives of the product are, what is to be accomplished, how the product/ system fits into business needs, and finally how product is to be used on day-to-day basis. ***But elicitation is difficult because:***

   a. ***Problems of Scope:*** Boundary of the system is ill-defined, or customers/ users specify unnecessary technical detail that make confuse rather than clarify, overall system objectives.
   b. ***Problems of Understanding:*** Customers or users have a poor understanding of their computing environment, the problem domain etc., specify requirements that conflict with other users needs or specify requirements that are ambiguous or untestable.
   c. ***Problems of Volatility:*** Requirements change over time.

✓ Software engineers overcome these three problems by gathering requirements in an organised manner.

3) **Elaboration:** Information obtained from the customer during Inception and elicitation is expanded and refined during elaboration. It focuses on developing a refined technical model of software functions, features and constraints.

Elaboration is driven by creation and reinforcement of <u>user scenarios</u> that describe how end-user will interact with the system. Each user scenario is parsed to extract <u>analysis classes</u> (business entities) that are visible to end user. The relationships and collaboration between classes are identified and UML diagrams are produced. End-result of elaboration is an <u>analysis model</u> that defines informational, functional and behavioural domain of the problem.

4) **Negotiation:** In this task, customers, users and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Risks associated with each requirement are identified and analysed. Rough "guestimates" of development are made and used to assess impact of each requirement on project cost and delivery time. Measures are taken in negotiations (discussions) so that each party achieves some satisfaction.

5) **Specification:** A specification can be written document, a set of graphical models, a formal mathematical model (algorithm), a collection of usage scenario or a prototype or a combination of these. It is necessary to remain flexible when specification is to be developed. For <u>large systems</u>, a <u>written</u> <u>document</u> is the best approach, whereas for <u>smaller products, usage scenarios</u> are best.
   1. Specification is final work product by requirements engineer.
   2. It serves as foundation for subsequent Software Engineering activities and describes function, performance and constraints in the product.

6) **Validation:** Work products produced as a result of Requirement Engineering are assessed for quality during validation step. It examines,
   1. Specification to ensure all software requirements are stated unambiguously.
   2. That inconsistencies, omissions, errors have been detected and corrected.
   3. The work products conform to standards established for the process, project and product.

❖ Primary requirement validation mechanism is the Formal Technical Review. The review team consists of Software Engineers, customers, users other stakeholders. Review team examines specification for errors, missing information, inconsistencies, conflicting requirements or unrealistic requirements.

7) **Requirements Management**: "It is a set of activities that help the project team identify, control and track requirements and changes two requirements at any time as the project proceeds."
   ❖ It begins with identification; each requirement is assigned a unique identifier. Once requirement have been identified, traceability/ feasibility tables are developed. Each traceability table relates requirements to one or more aspects of the system or its environment.

   Possible traceability tables are:
   1. <u>Features traceability table:</u> shows how requirements relate to important customer observable system/ product features.
   2. <u>Source traceability table:</u> Identifies source of each requirement.
   3. <u>Dependency traceability table:</u> Indicates how requirements are related to one another.
   4. <u>Subsystem traceability table:</u> Categorises requirements by the subsystem(s) that they govern.
   5. <u>Interface traceability table:</u> Shows how requirements relate to both internal and external system interfaces.

These traceability tables are maintained in Requirements Database to understand how a change in requirements will affect different aspects of the system to be built.

INCEPTION (OR) INITIATING REQUIREMENT ENGINEERING PROCESS:

To get the project started and forward towards a successful solution, we need the following steps to initiate Requirement Engineering:

1. **Identifying the Stakeholders:** stakeholder is defined as "anyone who benefits in a direct or indirect way from the system which is being developed." Each stakeholder has a different view of the system, achieves different benefits when system is successfully developed and is open to different risks.

At inception, requirement engineer should create list of people who will input as requirements are elicited. The initial list will grow as stakeholders are contacted further.

2. **Recognizing Multiple Viewpoints:** As different stakeholders exist; requirements of the system will be explored from many different points of the view. Each of various constituencies such as marketing groups, business managers, end users, Software Engineers will contribute information to the Requirements Engineering process. For ex. support engineer may focus on the software maintainability.

The job of requirements engineer is to categorize all stakeholder information including inconsistent and conflicting requirements in a way that will allow decision makers to <u>choose a consistent set of requirements for the system.</u>

3. **Working towards Collaboration:** Customers should collaborate among themselves and with Software Engineers to result a successful system. <u>The job of requirement engineer is to identify areas of commonality and areas of conflict or inconsistency.</u>

In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (Ex: business manager) may make the final decision about which requirements make the cut.

4. **Asking the first question:** The questions asked at the Inception of the project should be "context free". The first set of questions focus on customer and other stakeholders, overall goals, and benefits.
   - ❖ For ex: requirements engineering might ask:
     - Who is behind the request for this work?
     - Who will use the solution?
     - What will be the economical benefit of the successful solution?
     - Is there another source for the solution that you need?

   These questions help to identify all stakeholders who will have interest in software to be built. These also identify measurable benefits of successful implementation and alternatives for software development.
   - ❖ Next set of questions include:
     - What problems will this solution address?
     - How would you characterize "good output"?
     - Can you show me the environment where solution will be used?
     - Will special performance issues or constraints affect the way the solution is approached?

   These questions enable software team to gain better understanding of the problems and allows customer to say his/her perceptions.
   - ❖ Final set of questions are:
     - Are you the right person and are your answers "official"?
     - Are my questions relevant to your problem?
     - Am I asking too many questions?
     - Can anyone else provide additional information?
     - Should I be asking you anything else?

   These questions focus on effectiveness of communication. These are also called as meta- questions. All these questions will help to "break the ice" and initiate the communication that is essential for successful elicitation.

ELICITING REQUIREMENTS

The Q&A session should be used for the first encounter only and then replaced by requirements elicitation format.

Collaborative Requirements Gathering: Many different approaches to collaborative requirements gathering have been proposed and each follows the basic guidelines:

1. Meetings are conducted and attended by both Software Engineers, customers along with stakeholders.
2. Rules for preparation and participation are established.
3. An agenda is suggested that is formal enough to cover all important points but informal enough to encourage free flow of ideas.
4. A "facilitator" (customer/ developer/ outsider) controls the meeting.
5. A "definition mechanism" (worksheets etc) can be used.
6. The goal is to
   a. Identify the problem.
   b. Propose elements of the solution
   c. Negotiate different approaches
   d. Specify preliminary set of solution requirements.

During Inception the stakeholders write a "one or two page request". A meeting place, time, date and a facilitator are selected. Then the product request is distributed to all attendees before the meeting date, and ask to go through the product request and make suggestions in the meeting.

Quality Function Deployment: QFD is a technique that translates the needs of customer into the technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the Software Engineering process". QFD identifies three types of requirements:

1. **Normal Requirements:** These reflect objectives and goals stated for a product during meetings with the customer. If these requirements are present then the customer is satisfied.
2. **Expected Requirements:** These are implicit to the product and customer does not explicitly state them. Their absence will cause significance dissatisfaction.
3. **Exciting Requirements:** These reflect features that go beyond customer's expectations and prove to be very satisfying when present.

- ✓ In meetings with the customer, Function Deployment determines value of each function that is required for the system.
- ✓ Information Deployment identifies both data objects and events that system must consumer and produce.
- ✓ Task Deployment examines behaviour of the system within context of its environment.
- ✓ Value Analysis is conducted to determine relative priority of the requirements determined during each of three deployments.

- ❖ QFD uses customer interviews, observations, surveys and examination of historical data as raw data for requirements gathering activity. These data are then translated into a table of requirements, called as the Customer voice table that is reviewed with customer.

User Scenarios: Developers and users create a set of scenarios that identify a trend of usage for the system to be constructed. These scenarios often called use cases provide a description of how the system will be used.

Elicitation Work Products: These depend on the size of the system/product to be built. These include:

- ✓ A statement of need or feasibility.
- ✓ A bounded statement of scope for system or product.
- ✓ A list of customers, users and stakeholders.
- ✓ A description of system's technical environment.
- ✓ A list of requirements and constraints that apply.
- ✓ A set of usage scenarios that provide insight into use of the system.
- ✓ Any prototypes developed to better define requirements.

ELABORATION

Developing Use Cases: "Use case is defined as set of sequence of actions performed by an actor to achieve a specific result". An actor refers to various people that use system or product within context of the function.

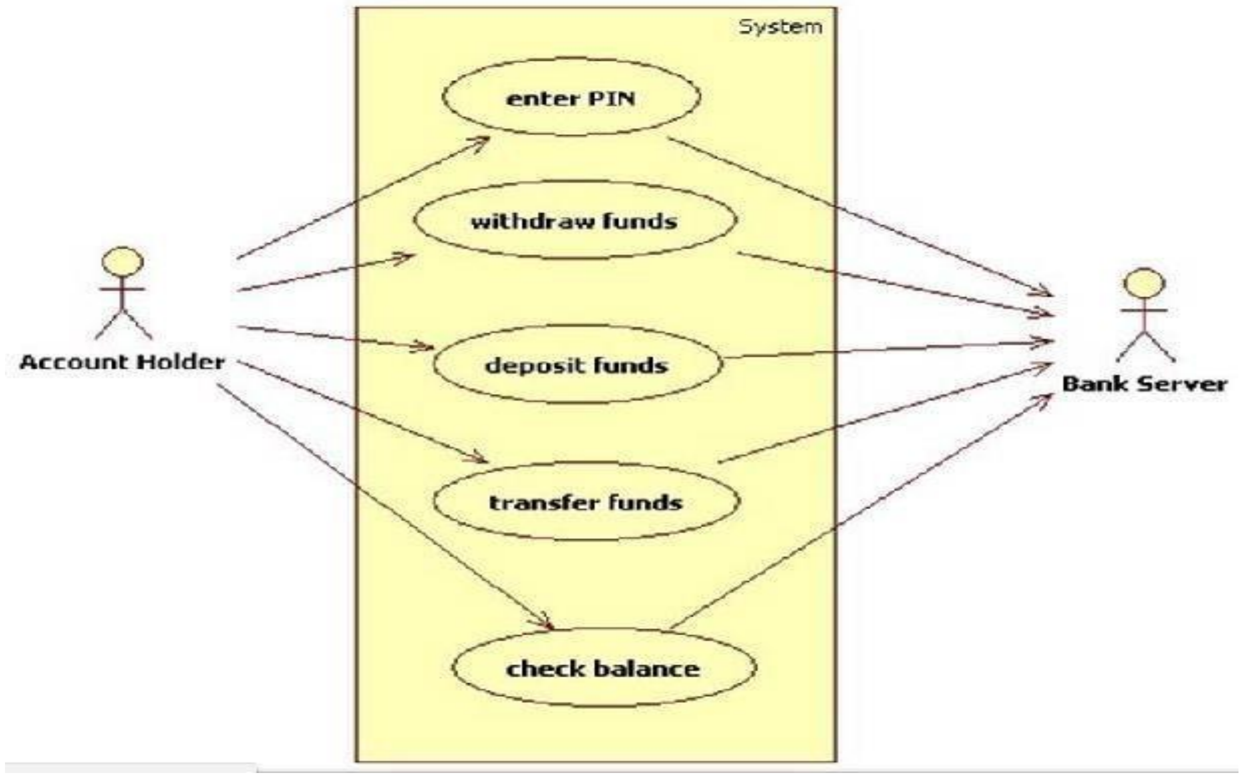


Fig: Use Case Diagram for ATM

Use Case is a collection of user scenarios that describe the thread of usage of a system. Each scenario is described from the point-of-view of an "actor"—a person or device that interacts with the software in some way. Each scenario answers the following questions:

- Who is the primary actor, the secondary actor (s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What extensions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Building the Analysis Model: The intent of the analysis model is to provide a description of the functional, informational and behavioural requirements for a computer-based system.

- ✓ Analysis model is a snapshot of requirements at any given time. We expect it to change. As the analysis model evolves, certain elements will become relatively stable and other elements may be more volatile, indicating customer does not yet understand requirements for system.

Elements of Analysis Model: The specific elements of the analysis model are dictated by analysis modeling method. These elements include:

1. **Scenario-based elements:** These are often the first part of analysis model that is developed. They serve as input (or) informational requirements for creation of other modeling elements.

- A variation in scenario based modeling depicts activities (functions/operations) that have been defined as a part of requirement elicitation task, i.e., sequence of activities is defined as part of analysis model. Activities can be represented iteratively at different levels of abstraction by using activity diagrams (or) use case diagrams.

As an example, consider UML diagrams for eliciting requirements.

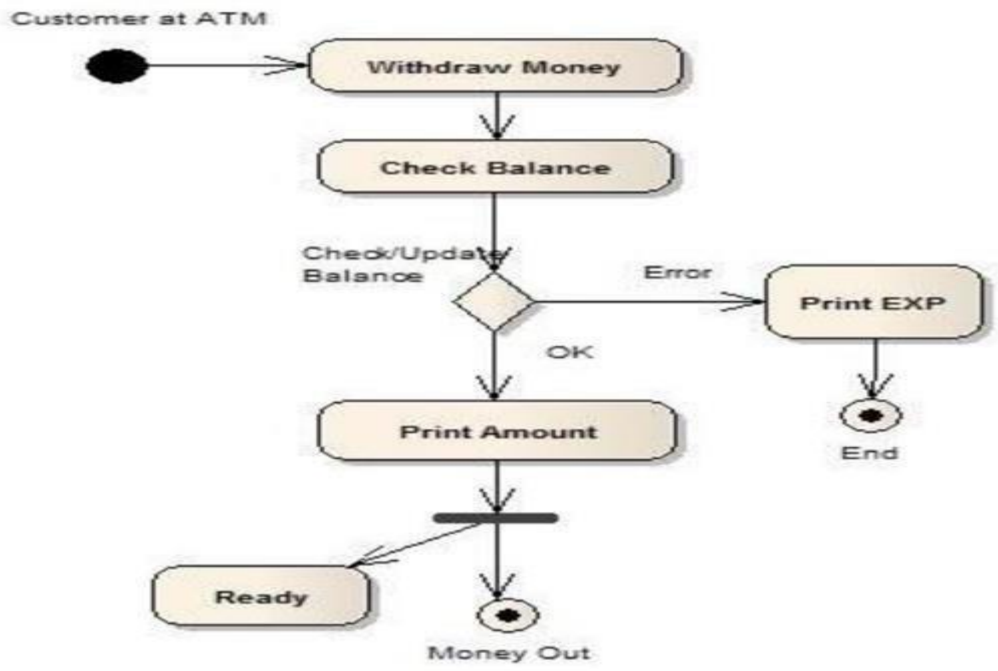
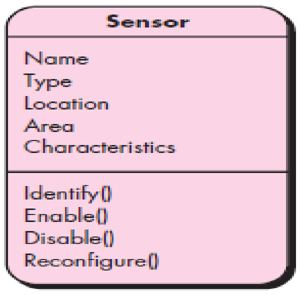


Fig: Activity Diagram

2. **Class-based elements:** Each usage scenario implies a set of "Objects" that are manipulated as an actor interacts with the system. These objects are categorised into "Classes"- a collection of things that have similar attributes and common behaviour.

- A class diagram usually represents the functional requirements in analysis model. Analysis model may also depict the manner in which classes collaborate with one another and relationships between them. An example for class diagram is,



3. **Behavioral Elements:** The behaviour of a product can have a profound effect on design and implementation approach. [i.e., static or dynamic].

- State diagram is used to represent behaviour of a system buy depicting its states and events that cause the system to change state. *A state is any observable mode of behaviour.*
- A state diagram indicates what actions are taken as a consequence of a particular event. A state diagram is given as,

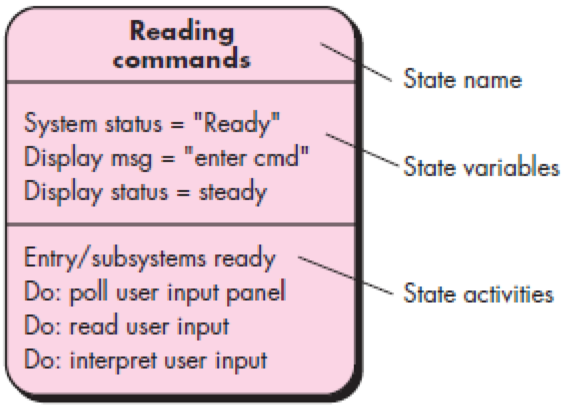


Fig: State Diagram Notation

4. **Flow oriented elements:** Information/Data is transformed as it flows through a computer-based system. System accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. This data flow is depicted using DFDs (Data Flow Diagrams).

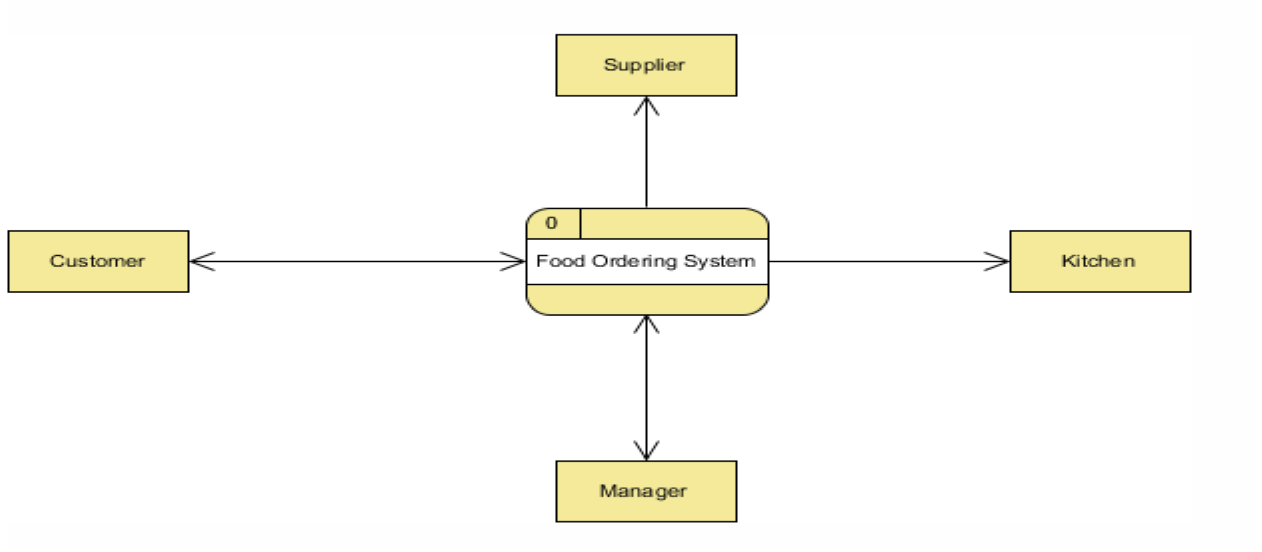


Fig: Data Flow Diagram

Analysis Patterns: Analysis patterns represent the things (class, function or behaviour) that can be reused when modeling many applications. Analysis patterns are integrated into analysis model by reference to the pattern name. They are also stored in a repository so that Requirement Engineers can reuse them.

Analysis pattern template includes:

1. Pattern name: A descriptive that captures essence of pattern.
2. Intent: Describes what pattern accomplishes or represents and/or what problem is addressed.
3. Motivation: A scenario that illustrates how pattern can be used to address the problem.
4. Forces and context: Description of external issues that can affect how pattern is used and how they will be resolved.
5. Solution: Description of how pattern is applied to solve problem.

6. Consequences: Address what happens when pattern is applied.
7. Design: Discusses how analysis pattern can be achieved through use of known design patterns.
8. Known uses: Examples of uses within actual systems.
9. Related patterns: One or more analysis patterns that are related to named pattern because the analysis pattern,
    a. is commonly used with named pattern
    b. is structurally similar to the named pattern
    c. is a variation of named pattern.

NEGOTIATING REQUIREMENTS

Customer and developer enter into a process of negotiation, where they will have a discussion about balancing functionality, performance, and other product or system characteristics against cost and time to market.

❖ The best negotiations strive for a **"win-win"** result. i.e., customer wins by getting system/ product that satisfies the needs, and software team wins by working to realistic and achievable budget and deadlines.

Boehm defines a set of negotiation activities:

1. Identification of system/ subsystem key stakeholders.
2. Determination of stakeholders "Win conditions".
3. Negotiate stakeholders win conditions to reconcile them into a set of **win-win** conditions for all concerned (including software team).

VALIDATING REQUIREMENTS

Requirements are validated in this task by a review of customer requirements. A review of analysis model addresses the following questions:

✓ Is each requirement consistent with overall objective for the system/ product?
✓ Have all requirements been specified at proper level of abstraction?
✓ Is the requirement really necessary (or) does it represent and add-on feature that may not be essential?
✓ Is each requirement bounded and unambiguous?
✓ Does each requirement have attribution? That is, is a source noted for each requirement?
✓ Do any requirements conflict with other requirements?
✓ Is each requirement testable?
✓ Is each requirement achievable in technical environment?
✓ Does requirements model properly reflect information, function and behaviour of system to be built?
✓ Are all patterns consistent with customer requirements?
✓ Have all patterns been properly validated?
✓ Have requirements patterns been used to simplify the requirements model?
✓ Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?