# CS8494 SOFTWARE ENGINEERING
# CHAPTER - 1
# INTRODUCTION TO SOFTWARE ENGINEERING

## 1.1 INTRODUCTION

### 1.1.1 Software

- Software is a set of instructions that provides desired features, function, and performance on execution.
- Software is a data structure that enable the programs to adequately manipulate information

### 1.1.2 Characteristics

1. Software is developed or engineered and not manufactured.
2. Software doesn't wear out.
3. Although the industry is moving toward component-based construction, most software continues to be custom built.

### 1.1.3 Categories of Computer Software

- System software is a set of programs that serve other programs.
- Application software is stand-alone programs that solve a specific business need.
- Engineering software has been characterized by "number crunching" algorithms. Eg: CAD, CAM
- Embedded software is the software that resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Eg: microwave oven, Air Conditioner, washing machine etc
- Product-line software provide a specific capability for use by many different customers. Eg: word processor.
- Web applications called "WebApps," provide a wide array of applications.
- Artificial intelligence software makes use of non numerical algorithms to solve complex problems. Eg: Expert systems, Robotics, Neural networks etc.

### 1.1.4 Software Myths

**Management myth**

Myth: Is the existing standards and procedures are enough for building software?

Reality: It is not used, not up to date, not complete, not focused on quality, time, and money

Myth: If we are behind the schedule, more programmers can be added to meet the deadline

Reality: Adding people to a late software project makes it later. Training time, increased communication lines

Myth: By outsourcing the project to a third party, I can just relax

Reality: Software projects need to be controlled and managed. Outsourced project need proper support for development

**Customer myth**

Myth: A general objective is enough to begin the code and the details can be filled later

Reality: Ambiguous statement of objectives spells disaster

Myth: Project requirements continuously change, but change can be easily accommodated because software is flexible

Reality: Impact of change depends on where and when it occurs in the software life cycle

**Practitioner's myth**

Myth: Once the program is executed then the job is done

Reality: 60% to 80% of all effort expended on software occurs after it is delivered

Myth: The only deliverable work product for a successful project is the working program

Reality: A working program is only one part of a software configuration that includes many elements.

### 1.1.5 Layered Technology

- **Software Engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.**
- Software engineering is a layered technology

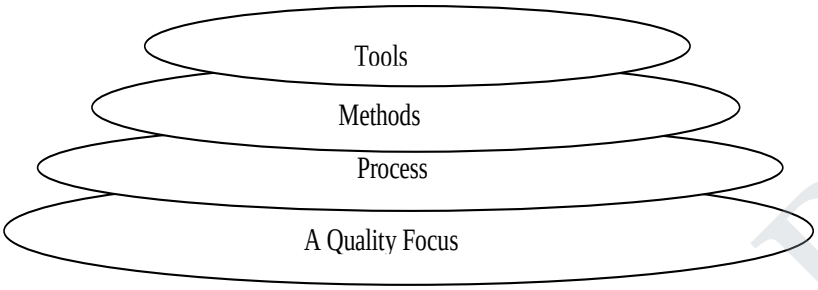



**Fig: Software Engineering Layers**

**Process**

- Process defines a framework for effective delivery of software engineering technology.
- The software process is the basis for the control of software projects and provides the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

**Methods**

- Software engineering methods provide the technical how-to's for building software.
- Methods include tasks like communication, requirements analysis, design modeling, program construction, testing, and support.

**Tools**

- Software engineering tools provide automated support for the process and the methods. When tools are integrated, the information created by one tool can be used by another

## 1.2 SOFTWARE PROCESS

- A software process is a collection of activities, actions, and tasks that are required to build high-quality software.
- A process defines who is doing what when and how to reach a certain goal.

- The aim of software process is effective on-time delivery of software with quality.

### 1.2.1 Elements of a software Process

**Activity:**

- An activity helps to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size, complexity and degree of rigor with which software engineering is to be applied.

**Action**

- Actions consist of the set of tasks that is used to produce the product.

**Task**

- A task focuses on a small, but well-defined objective that produces a tangible outcome.

### 1.2.2 Process framework

- A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects.

**Process framework activities**:

**Communication**

- Before starting any technical work, it is important to communicate and collaborate with the customer and other stakeholders.
- The main objective is to understand stakeholders' objectives for the project and to gather requirements that helps to define software features and functions.

**Planning**

- Software project plan defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

**Modeling**

- Model helps to better understand software requirements and the design that will achieve those requirements.

**Construction**

- This activity combines code generation and the testing that is required to uncover the errors in the code.

**Deployment**

- The software is delivered to the customer, who evaluates the product and provides feedback based on the evaluation.

- Generally, the framework activities are applied iteratively as a project progresses.
- Each project iteration produces a software increment that provides stakeholders with a subset of overall software features and functionality.
- As each increment is produced, the software becomes more and more complete.
- The Process framework activities are complemented by a number of umbrella activities which helps to manage and control progress, quality, change, and risk.

**Umbrella activities**:

- **Software project tracking and control** helps to assess progress against the project plan and take any necessary action to maintain the schedule.
- **Risk management** assesses risks that may affect the outcome and quality of the project.
- **Software quality assurance** defines the activities required to ensure software quality.
- **Technical reviews** assess the products to uncover and remove errors before they are propagated to the next activity.
- **Measurement** defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs.
- **Software configuration management** manages the effects of change throughout the software process.
- **Reusability management** defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
- **Work product preparation and production** encompasses the activities required to create work products such as models, documents, and lists.

- So a process adopted for one project might be significantly different than a process adopted for another project.

## 1.3 PROCESS MODEL

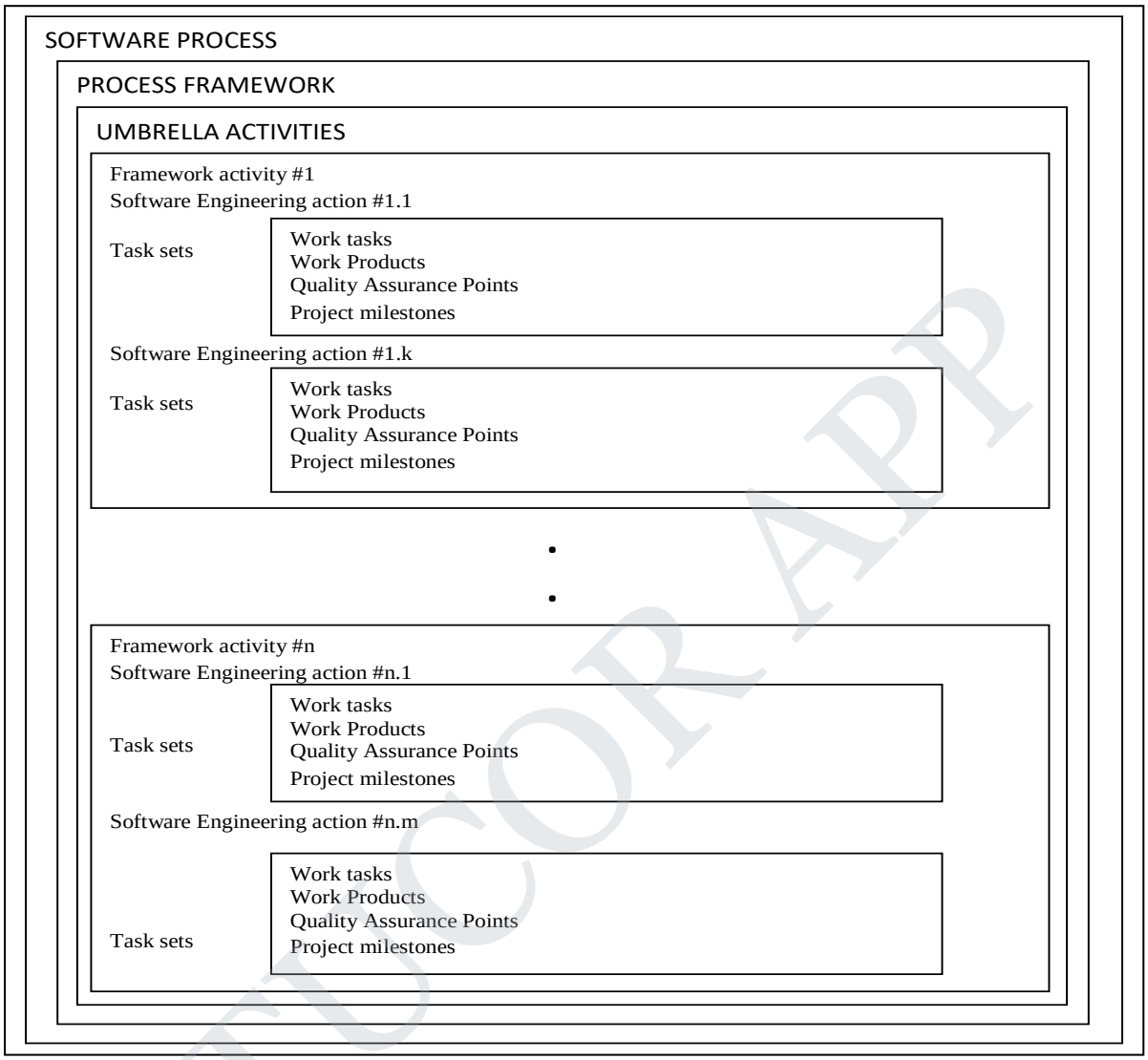


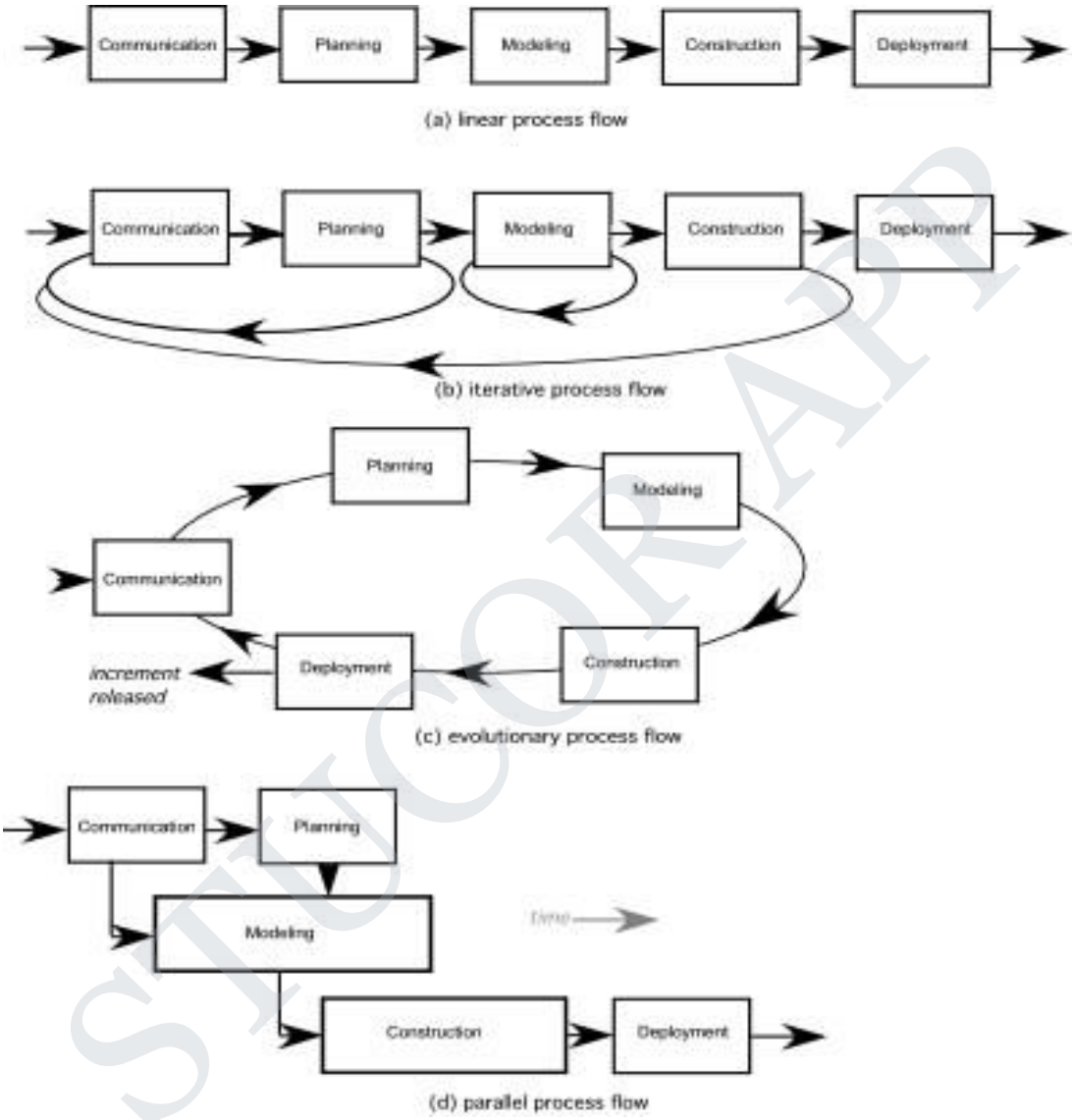

**Fig: A Generic Process Model**

- Process Flow:

  Process flow describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time

  - Linear process flow executes each of the five activities in sequence.
  - An iterative process flow repeats one or more of the activities before proceeding to the next.

- An evolutionary process flow executes the activities in a circular manner. Each evolution leads to a more complete version of the software.
- A parallel process flow executes one or more activities in parallel with other activities.




### 1.3.1 Defining a Framework Activity:

- Defining refers to specifying what actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders.

- For a small software project, the communication activity includes contacting the stakeholder, discuss the requirements, generate a brief statement of the requirement and get the review and approval of it from the stakeholder.
- For complex Projects, the communication activity might have six distinct actions: inception, elicitation, elaboration, negotiation, specification, and validation.

### 1.3.2 Identifying a Task Set

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
    - A list of the task to be accomplished
    - A list of the work products to be produced
    - A list of the quality assurance filters to be applied
- Eg: For a small project, the task set may include:
    - Make a list of stakeholders for the project.
    - Informal meeting with stake holders to identify the functions required
    - Discuss requirements and build a final list.
    - Prioritize requirements.
- Choose the task sets that achieve the goal and still maintain quality and agility.

### 1.3.3 Process Patterns

- A process pattern provides a template for describing problem solutions that is encountered during software engineering work
- The template for describing a process pattern:
    - **Pattern Name. - a** meaningful name describing the pattern within the context (e.g., **Technical Reviews**).
    - **Forces -** The environment where the pattern is encountered.
    - **Type -** The pattern type is specified. They are three types of patterns:

1. **Stage patterns** define a problem associated with a framework activity for the process. A stage pattern has multiple related task patterns. Eg: Establishing Communication includes the task pattern Requirements Gathering and others.
2. **Task patterns** define a problem associated with a software engineering action or work task and relevant to successful software engineering practice.

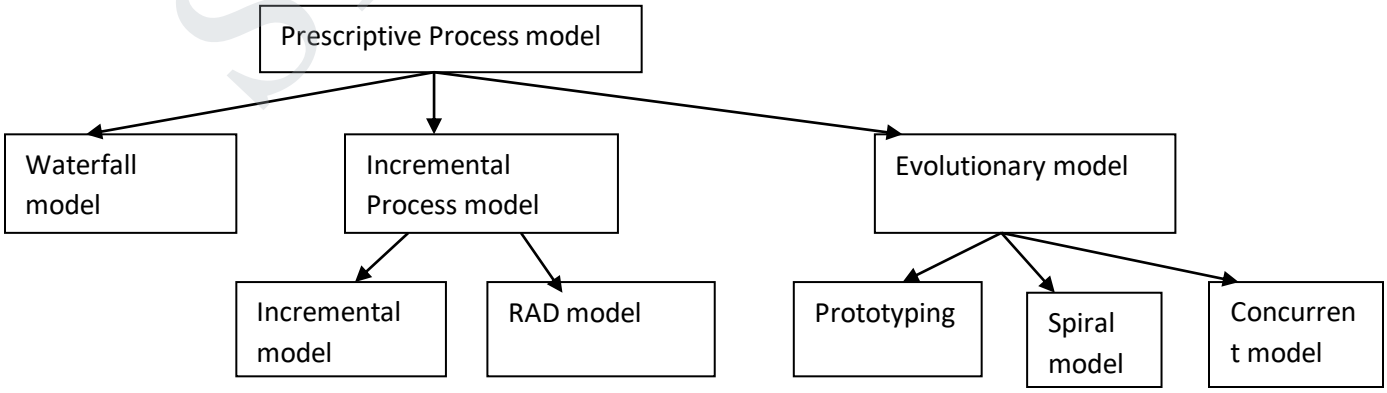
3. **Phase patterns** define the sequence of framework activities that occur with the process. Eg: Sprial Model or Prototyping

- **Initial context -** This describes the conditions under which the pattern applies.
- **Problem -** The specific problem to be solved by the pattern.
- **Solution -** Describes how to implement the pattern successfully and how the initial state of the process is modified as a consequence of the initiation of the pattern.
- **Resulting Context -** Describes the conditions that will result once the pattern has been successfully implemented.
- **Related Patterns - Provide** a list of all process patterns that are directly related to this one. Eg: the stage pattern Communication encompasses the task patterns: Project Team, Collaborative Guidelines, Scope Isolation, Requirements Gathering, Constraint Description, and Scenario Creation.
- **Known Uses and Examples - Indicate** the specific instances in which the pattern is applicable.

- Process patterns provide an effective mechanism for addressing problems associated with any software process.
- After developed the process patterns can be reused for the definition of process variants.

## 1.4 PRESCRIPTIVE PROCESS MODELS

- Prescriptive process models stress detailed definition, identification, and application of process activities and tasks.
- Their intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system.



### 1.4.1 Waterfall Model

- This is also referred as classic life cycle or linear - sequential model
- It is the oldest paradigm for SE.
- This model is used for developing projects where the requirements are well defined and reasonably stable, it leads to a linear fashion.
- Waterfall model suggests a systematic, sequential approach to software development.
- Phases:
  - Communication: Customer specification of requirements
  - Planning - identifying the work task, analysing the risk involved, scheduling the project and estimating the effort and time needed to complete the project etc
  - Modelling which involves translating the requirements gathered into a design
  - Construction – converting the design into the executable code and testing to uncover the errors.
  - Deployment includes delivery of the product and evaluation of the software.

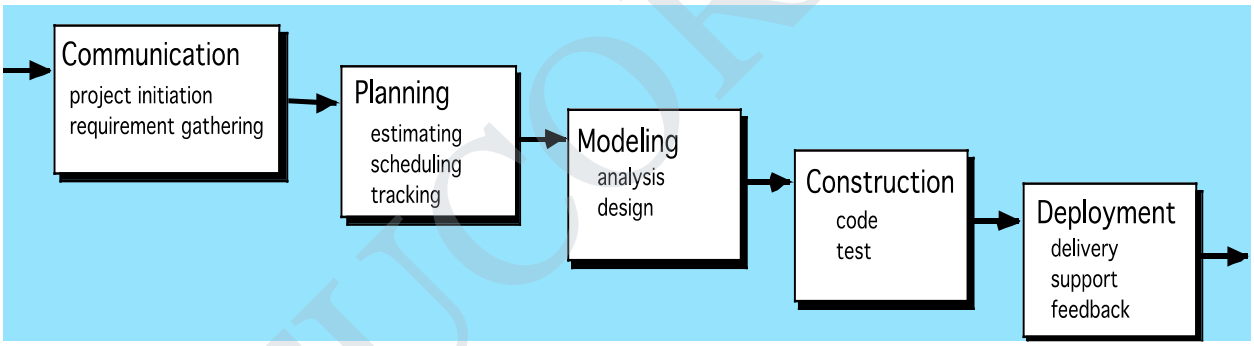



**Fig: Waterfall Model**

- Problems/ Drawbacks:
  1. Real projects rarely follow the sequential process flow.
  2. The linear model can accommodate iteration, but it does so indirectly which can cause confusion as the project team proceeds.
  3. It is difficult for the customer to state all requirements explicitly.
  4. The customer must have patience. A working version of the program will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.
- The linear nature of the classic life cycle leads to ***"blocking states"*** in which some project team members must wait for other members of the team to complete dependent

tasks. The time spent waiting can exceed the time spent on productive work. The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

- It can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

### 1.4.2 V-Model

- A variation of waterfall model is referred as V model which includes the quality assurance actions associated with communication, modeling and early code construction activities.
- Team first moves down the left side of the V to refine the problem requirements.
- Once code is generated, the team moves up the right side of the V, performing a series of tests that validate each of the models created.
- The V-model provides a way of visualizing how verification and validation actions are applied at the different stages of development.

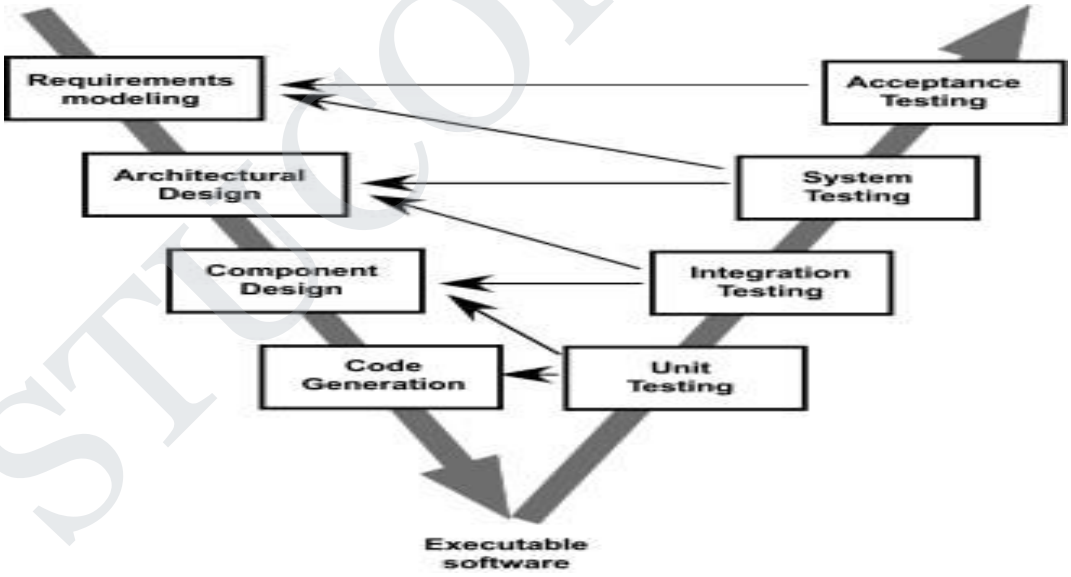



Fig: V - Model

### 1.4.3 Incremental Process Models

- Incremental models construct a partial implementation of a total system and then slowly add increased functionality
- The incremental model prioritizes requirements of the system and then implements them in groups.

- Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented.
- It combines elements of both linear and parallel process flows.
- Each linear sequence produces deliverable increments of the software.
- The first increment is the core product with many supplementary features.
- After implementation and evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and includes additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.

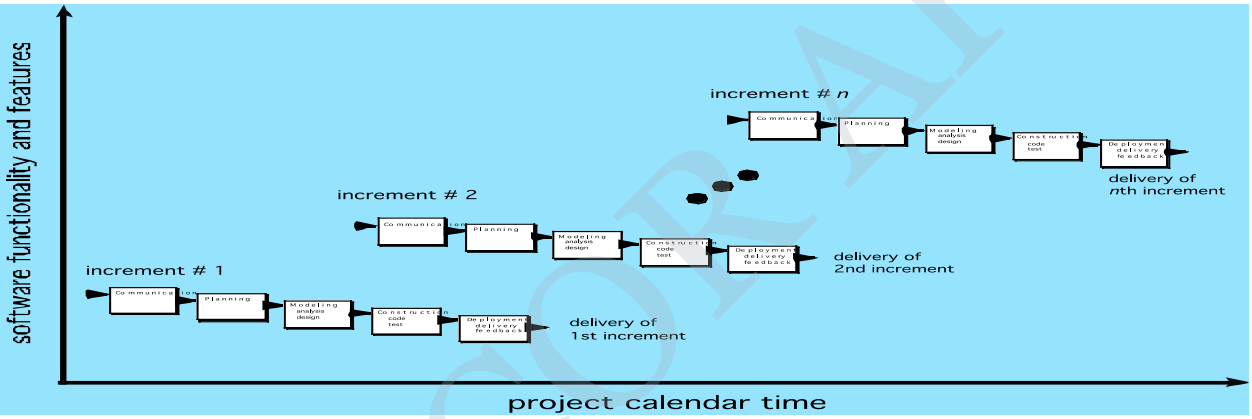



Fig: Incremental Model

- For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.
- It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- The incremental process model focuses on the delivery of an operational product with each increment.
- Incremental development can be used when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. If the

core product is well received, then additional staff if required can be added to implement the next increment.

- Increments can be planned to manage technical risks.

**RAD (Rapid Application Development)**

- RAD model is based on prototyping and iterative development with no specific planning involved.
- Different phases of RAD model include

1. **Business Modeling**

   On basis of the flow of information and distribution between various business channels, the product is designed.

2. **Data Modeling**

   The information collected from business modeling is refined into a set of data objects that are significant for the business

3. **Process Modeling**

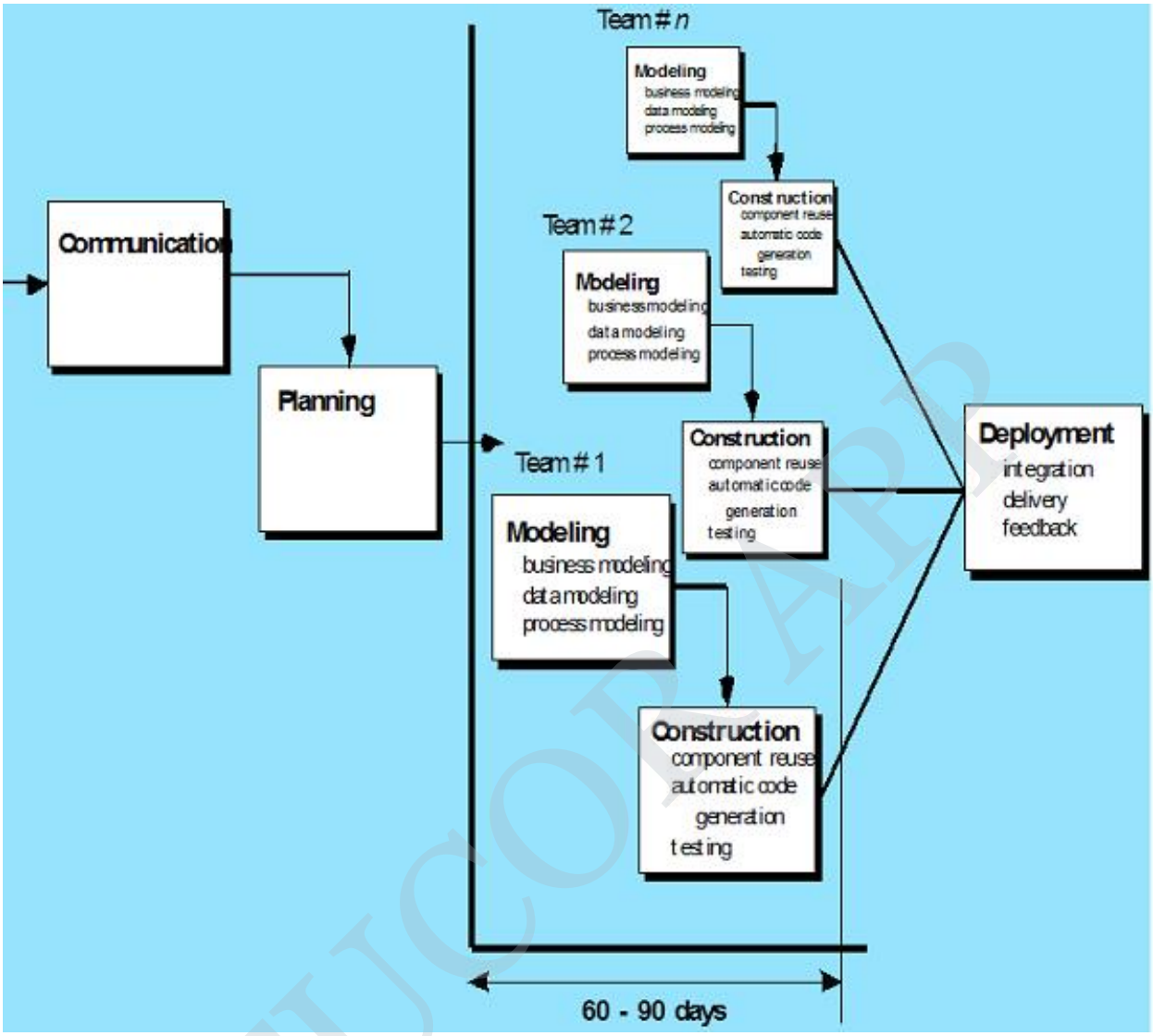
   The data object that is declared in the data modeling phase is transformed to achieve the information flow necessary to implement a business function

4. **Application Generation**

   Automated tools are used for the construction of the software, to convert process and data models into prototypes

5. **Testing and Turnover**

   As prototypes are individually tested during every iteration, the overall testing time is reduced in RAD.




**Advantages**

- Use of reusable components helps to reduce the cycle time of the project.
- Encourages user involvement
- Reduced cost.
- Flexible and adaptable to changes

**Disadvantages**

- For large scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- If developers and customers are not committed to the rapid-fire activities, then project will fail.

- If a system cannot properly be modularized, building the components necessary for RAD will be problematic.
- The use of powerful and efficient tools requires highly skilled professionals.
- Customer involvement is required throughout the life cycle.

### 1.4.4 Evolutionary Models

- In some cases the requirement changes over time, for such projects evolutionary approach can be used where a limited version is delivered to meet competitive pressure.
- In this model, a set of core product and requirements is well understood, but the details and extension have yet to be defined.
- It is iterative that enables to develop increasingly more complete version of the software.
- Two types of evolutionary approaches are there namely
    - Prototyping and
    - Spiral models.

**Prototyping**

- The process of building a preliminary design, trying it out, refining it, and trying again has been called an iterative process of systems development.
- Prototyping is more explicitly iterative than the conventional life cycle, and it actively promotes system design changes.
- Prototyping consists of building an experimental system rapidly and inexpensively for end users to evaluate.
- By interacting with the prototype, users can get a better idea of their information requirements.
- The prototype can be used as a template to create the final system.
- The prototype is a working version of the system or part of the system, but it is meant to be only a preliminary model. After implementing, the prototype will be further refined until it conforms precisely to users' requirements.
- **When to use:**

Customer defines a set of general objectives but does not identify detailed requirements or the developer may not be sure of the efficiency of an algorithm.

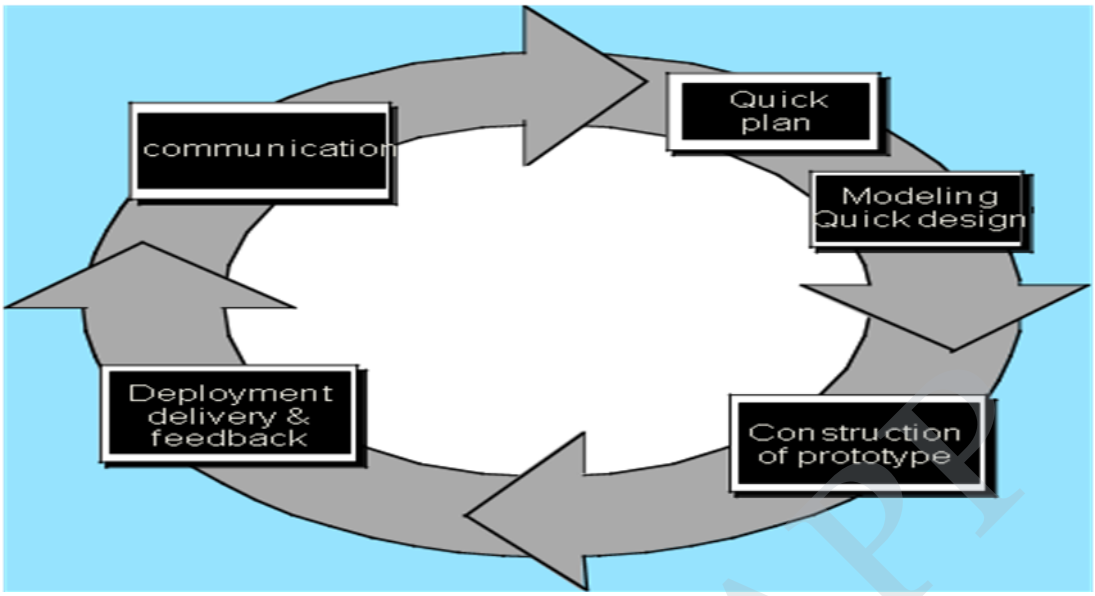



Fig: Prototype model

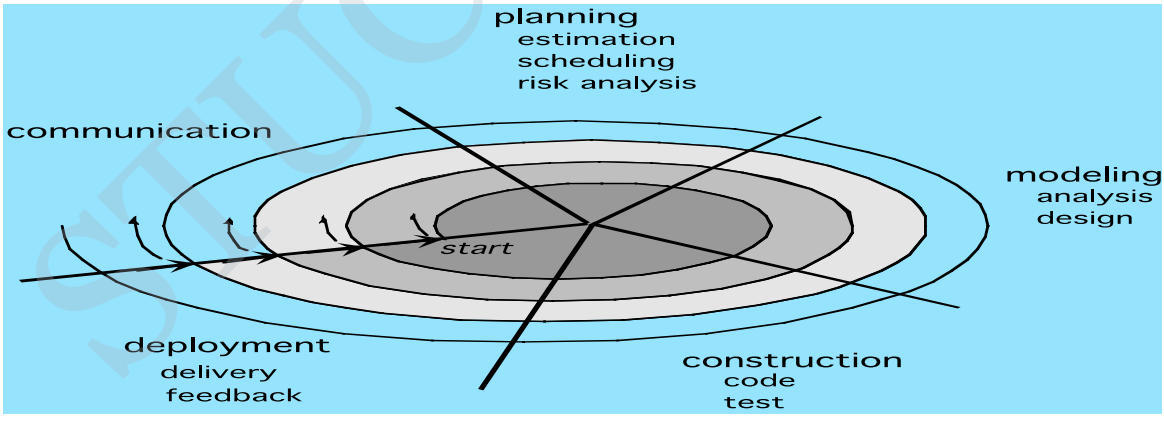
- **Phases:**
  - Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is necessary.
  - A quick plan for prototyping and modeling is determined.
  - Quick design focuses on a representation of those aspects the software that will be visible to end users.
  - Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder's comments will be used to refine requirements.

**Advantages and Disadvantages of Prototyping:**

- Prototyping is most useful when there is some uncertainty about requirements or design solutions.
- Because prototyping encourages end-user involvement throughout the systems development life cycle, it is more likely to produce systems that fulfill user requirements.
- If the completed prototype works reasonably well, there is no need for reprogramming and redesign.

**Spiral Model**

- Spiral model couples the iterative nature of prototyping with the systematic aspects of the waterfall model
- It is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- Two main distinguishing features:
  - Cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
  - Set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.
- A series of evolutionary releases are delivered.
- During the early iterations, the release might be a model or prototype. During later iterations, increasingly more complete version of the engineered system are produced.
- The first circuit in the clockwise direction might result in the product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- After each iteration, the project plan has to be refined. Cost and schedule are adjusted based on feedback. Also, the number of iterations will be adjusted by project manager.




Advantages & disadvantages:

- Good to develop large-scale system as software evolves as the process progresses and risk should be understood and properly reacted to.
- Prototyping is used to reduce risk.

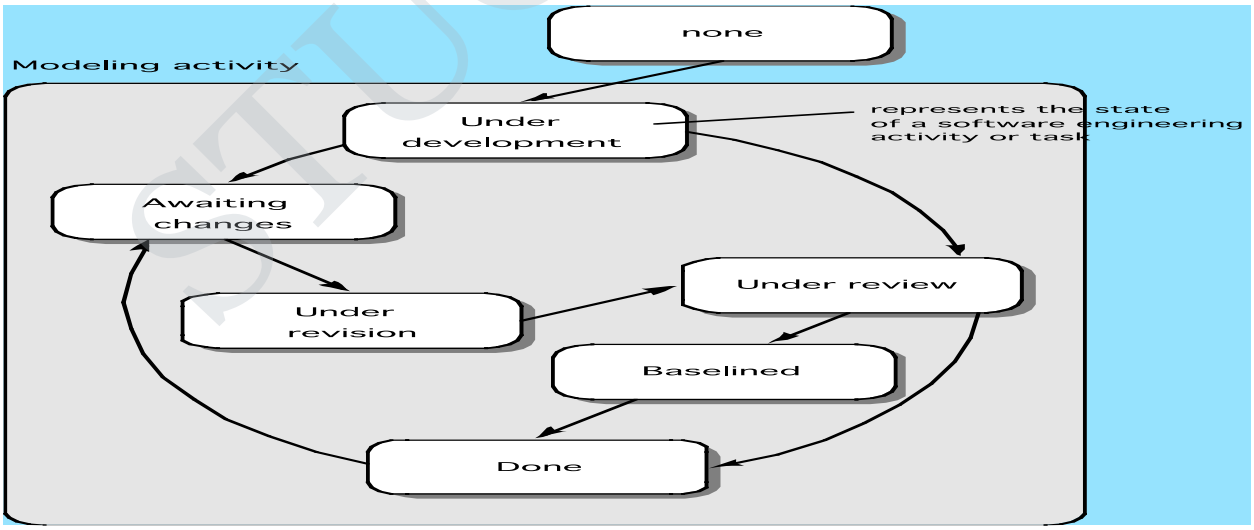
- It is difficult to convince customers that it is controllable as it demands considerable risk assessment expertise.

**Three Concerns on Evolutionary Processes**

- First concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, it does not establish the maximum speed of the evolution. If the evolution occurs too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand, if the speed is too slow then productivity could be affected.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality. The speed of the development should be prioritized over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product when the opportunity niche has disappeared.

### 1.4.5 Concurrent Model

- Allow a software team to represent iterative and concurrent elements of any of the process models. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following actions: prototyping, analysis and design.
- The Figure shows modeling may be in any one of the states at any given time.




- For example, communication activity has completed its first iteration and in the awaiting changes state. The modeling activity was in inactive state, now makes a transition into the

under development state. If customer indicates changes in requirements, the modeling activity moves from the under development state into the awaiting changes state.

- Concurrent modeling is applicable to all types of software development
- This provides an accurate picture of the current state of a project.
- Rather than confining software engineering activities, actions and tasks to a sequence of events, it defines a process network.
- Each activity, action or task on the network exists simultaneously with other activities, actions or tasks.
- Events generated at one-point trigger transitions among the states.

## 1.5 SPECIALIZED PROCESS MODEL

- These models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

### 1.5.1 Component-Based Development

- The component-based development model incorporates many of the characteristics of the spiral model.
- It is evolutionary in nature, uses an iterative approach to the creation of software.
- The component-based development model constructs applications from prepackaged software components.
- Modeling and construction activities begin with the identification of candidate components.
- These components can be designed as either conventional software modules or object-oriented classes or packages of classes.
- The component-based development model incorporates the following steps to develop the software:

**1.** Available component-based products are researched and evaluated for the application domain in question.

**2.** Component integration issues are considered.

**3.** A software architecture is designed to accommodate the components.

**4.** Components are integrated into the architecture.

**5.** Comprehensive testing is conducted to ensure proper functionality.

- The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.
- This model helps to reduce the development cycle time as well as the project cost

### 1.5.2 The Formal Methods Model

- Every software engineering methodology is based on a recommended development process proceeding through several phases:
    - Requirements, Specification, Design
    - Coding, Unit Testing
    - Integration and System Testing, Maintenance
- Formal methods can
    - Be a foundation for designing safety critical systems
    - Be a foundation for describing complex systems
    - Provide support for program development
- The formal methods model has a set of activities that leads to formal mathematical specification of computer software.
- Formal methods help to specify, develop, and verify a computer-based system by applying a mathematical notation.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily
- When formal methods are used during design, they help in program verification and therefore enable to discover and correct errors.
- Thus the formal methods model provides defect-free software.
- Drawbacks:
    - The development of formal models is quite time consuming and expensive.
    - Because few software developers have the necessary background to apply formal methods, extensive training is required.
    - It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

### 1.5.3 Aspect-Oriented Software Development

- Aspect-oriented software development (AOSD) aims to address crosscutting concerns by providing means for systematic identification, separation, representation and composition.
- Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. This results in better support for modularization hence reducing development, maintenance and evolution costs.
- Some Crosscutting concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).
- Aspect-oriented software development (AOSD) is often referred as aspect-oriented programming (AOP), provides a process and methodological approach for defining, specifying, designing, and constructing aspects.
- An aspect-oriented process is likely to adopt both evolutionary and concurrent process models.
- The evolutionary model is appropriate as aspects are identified and then constructed.
- The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components.

Advantages:

- Provides better modularization support of software designs, reducing software design, development and maintenance costs
- Because concerns are encapsulated into different modules, localization of crosscutting concerns is better promoted and handled.
- Promotes reusability of code
- Smaller code size, due to tackling cross cutting concerns

**Aspect-oriented component engineering (AOCE):**

- AOCE uses a concept of horizontal slices through vertically-decomposed software components, called “aspects,” to characterize cross-cutting functional and non-functional properties of components.

- Systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on.
- Components may provide aspect details relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on.

## 1.6 INTRODUCTION TO AGILITY

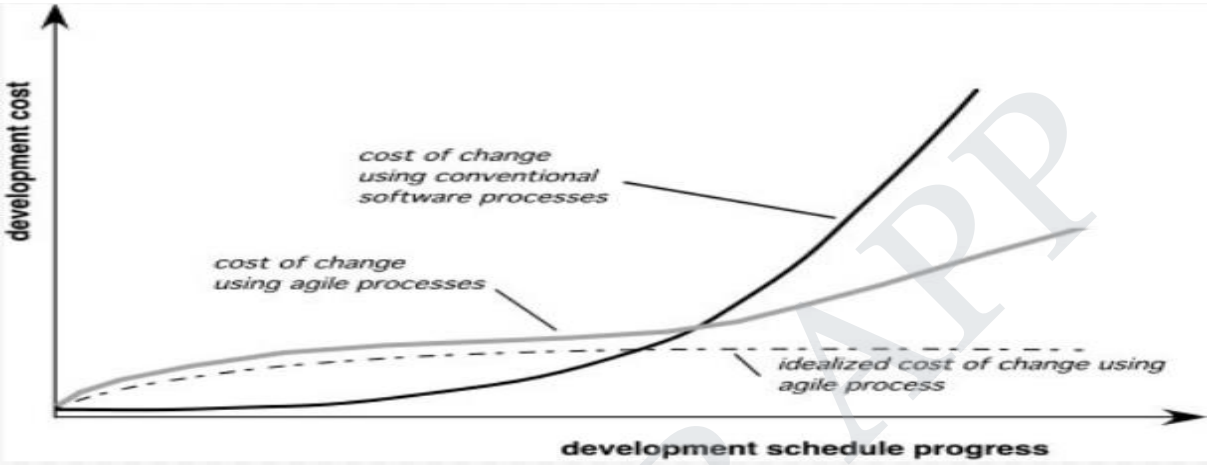
### 1.6.1 What is Agility?

- Agility is effective (rapid and adaptive) response to change (changes in team members, new technology, requirements etc) that may have an impact on the product being developed.
- It encourages effective communication in structure and attitudes among all team members, technological and business people, software engineers and managers.
- It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing);
- It involves the customer in the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects
- It recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.
- Agility can be applied to any software process but it is essential that the process should support incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

### 1.6.2 Agility and the Cost of Change

- The cost of change **increases nonlinearly** as a project progresses.
- It is easy to accommodate a change in the early stages of a project and cost will be minimal but in case of major functional change, it requires a modification to the architectural design, construction, testing etc. so the Costs escalate quickly.

- A well-designed agile process may "flatten" the cost of change curve by coupling the incremental delivery with agile practices such as continuous unit testing and pair programming. Thus team can accommodate changes late in the software project without dramatic cost and time impact.




### 1.6.3 Agility Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face–to–face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.

11. The best architectures, requirements, and designs emerge from self–organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

### 1.6.4 Human Factors

- **"Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." i.e. The process molds to the needs of the people and team, not the other way around.**
- The key traits that must exist among the people on an agile team are:
  - Competence.
  - Common focus.
  - Collaboration.
  - Decision-making ability.
  - Fuzzy problem-solving ability.
  - Mutual trust and respect.
  - Self-organization.

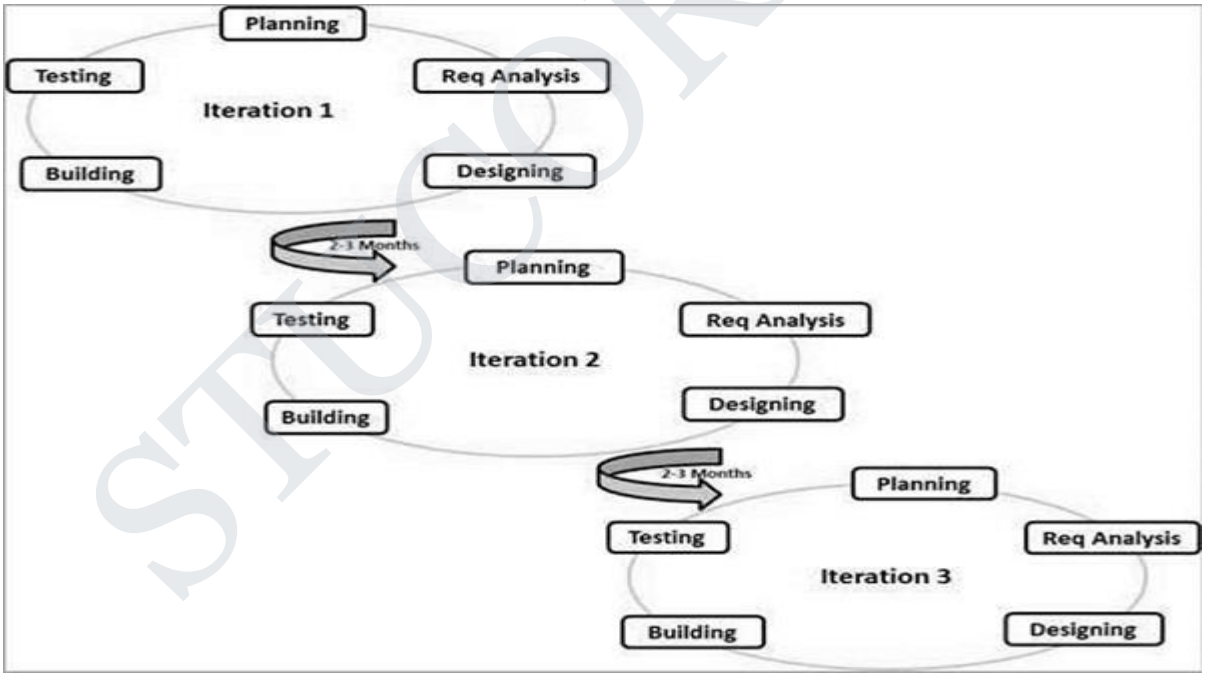
### 1.6.5 Agile Process

- Agile software process addresses a number of key assumptions say:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

- An agile process must be **adaptable** in order to manage the unpredictability.
- An agile software process must **adapt incrementally**.
- Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product.
- Agile Methods break the product into small incremental builds.

- Each build is incremental in terms of features and the final build holds all the features required by the customer.
- Every iteration involves cross functional teams working simultaneously on various areas like –
  - Planning
  - Requirements Analysis
  - Design
  - Coding
  - Testing
- In Agile methodology, there is no detailed planning and there is clarity only in respect of what features need to be developed. The team adapts to the changing product requirements dynamically.
- The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future.




- Program specification, design, and implementation are interleaved.
- The system is developed as a series of frequent versions or increments.
- Stakeholders involved in version specification and evaluation.
- Extensive tool support (e.g. automated testing tools) used to support development.
- **Minimal documentation** - focus on working code.

**Advantages:**

- Rapid Functionality development
- Promotes team work
- Adaptable to changing requirements
- Delivers early partial working solutions.
- Flexible and Easy to manage.

**Disadvantages**

- Not suitable for complex projects/dependencies.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- Prioritizing changes can be difficult where there are multiple stakeholders.

**Note:**

Agile methods include Rational Unified Process, Scrum, Extreme Programming, Adaptive Software Development, Crystal clear, Feature Driven Development, and Dynamic Systems Development Method. These are now collectively referred to as Agile Methodologies.

## 1.7 Extreme Programming (XP)

**What is Extreme Programming?**

**XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software.**

- Extreme Programming (XP), the most widely used approach to agile software development proposed by Kent Beck.
- Recently, a variant of XP, called Industrial XP (IXP) has been proposed which refines XP and targets the agile process specifically for use within large organizations.

### 1.7.1 XP Values

- XP is based on the 5 values:
    1. Communication
    2. Simplicity
    3. Feedback
    4. Courage
    5. Respect.

- Each of these values is used as a driver for specific XP activities, actions, and tasks.

**Communication**

- To achieve effective communication, XP emphasizes close, informal (verbal) collaboration between customers and developers.

**Simplicity**

- To achieve simplicity, XP restricts developers to design only for immediate needs, rather than consider future needs. The design can be refactored later if necessary.

**Feedback**

- Feedback is derived from three sources: the implemented software itself, the customer, and other software team members.

**Courage**

- Team should be prepared to make hard decisions that support the other principles and practices.

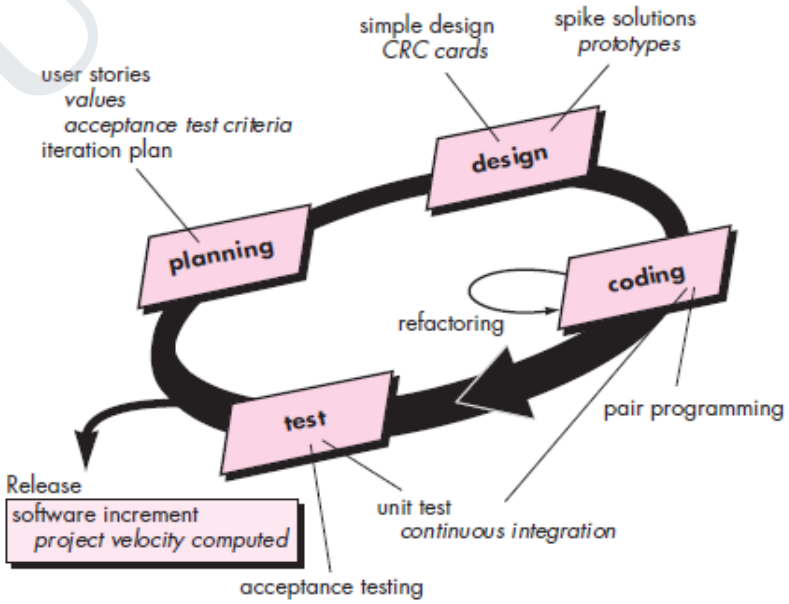
**Respect**

By following each of these values, the agile team inculcates respect among its members, between other stakeholders and team members, and indirectly, for the software itself.

### 1.7.2 XP Process

Extreme Programming uses an object-oriented approach that encompasses a set of rules and practices that occur within the context of four framework activities:

- Planning
- Design
- Coding
- Testing



**XP Planning**

- The planning activity is also called as planning game.
- Planning begins with a requirements gathering where customers and developers negotiate requirements in the form of User stories captured on index cards.
- User stories describe the required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card.
- The customer assigns a value (i.e., a priority) to the story based on the overall business value of the function.
- Agile team assesses each story and assigns a cost measured in development weeks.
- If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again.
- Stories are grouped to for a deliverable increment and a commitment is made on delivery date.
- Stories are developed in one of three ways:

  1.All stories will be implemented immediately

  2.The stories with highest value will be moved up in the schedule and implemented first

  3.The riskiest stories will be moved up in the schedule and implemented first.

**Fig: Sample Index card with User stories**

| |
|---|
| #5 |
| Customer can change his/her personal information in |
| the system |
| |
| Priority: 2 (high/low/medium) |
| Estimate: 4 (development weeks) |

- After the first increment, the XP team computes project velocity.
- Project velocity is the number of customer stories implemented during the first release.
- Project velocity can be used to
  1. help estimate delivery dates and schedule for subsequent releases.
  2. determine whether an over commitment has been made for all stories across the entire development project.

- During development, the customer can add new stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all and modifies its plans accordingly.

**XP Design**

- XP design follows the KIS (keep it simple) principle.
- A simple design is preferred than complex representation.
- Design provides implementation guidance for the story.
- XP encourages the use of CRC cards. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment.
- Eg: sample CRC for ATM system

| **Class :** User Menu | |
|---|---|
| **Responsibility** | **Collaborators** |
| Display Main menu<br>Ask PIN<br>Validate PIN<br>Select transaction type<br>Debit amount<br>Print balance etc | Bank System<br>.<br><br>Printer |

- The CRC cards are the only design work product produced as part of the XP process.
- For difficult design problems, XP recommends creation of an intermediate operational prototype called a spike solution.
- XP encourages "refactoring"—an iterative refinement of the internal program design

**XP Coding**

- XP recommends the construction of a unit test for a store *before* coding commences.
- A key concept during the coding activity is **"pair programming"**
- This provides a mechanism for real-time problem solving and real-time quality assurance.
- As pair programmers complete their work, the code they develop is integrated with the work of others.
- This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

**XP Testing**

- XP uses Unit and Acceptance Testing.
- All unit tests are executed daily
    - "Acceptance tests" are defined by the customer and executed to assess customer visible functionality
- The unit tests that are created should be implemented using a framework that enables them to be automated.
- This encourages a regression testing strategy whenever code is modified.
- The individual unit tests are organized into a "universal testing suite", integration and validation testing of the system can occur on a daily basis.
- This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry.
- XP acceptance tests (customer tests), are specified by the customer and focus on overall system features and functionality. Acceptance tests are derived from user stories.

### 1.7.3 Industrial XP

IXP incorporates **six new practices** that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

1. **Readiness assessment:**

   The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.

2. **Project community**:

   Classic XP suggests that the right people be used to populate the agile team to ensure success. The people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team.

3. **Project chartering:**

   Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

4. **Test-driven management:**

   Test-driven management establishes a series of measurable "destinations "and then defines mechanisms for determining whether or not these destinations have been reached.

5. **Retrospectives:**

   An IXP team conducts a specialized technical review after a software increment is delivered. Called a retrospective, the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release.

6. **Continuous learning:**

   Because learning is a vital part of continuous process improvement, members of the XP team are encouraged to learn new methods and techniques that can lead to a higher quality product.

**CS8494 SOFTWARE ENGINEERING**
**UNIT-II**
**REQUIREMENTS ANALYSIS AND SPECIFICATION**

## 2.1 SOFTWARE REQUIREMENTS

- Requirements of a system are the descriptions of the services provided by the system and its operational constraints.
- The requirements reflect the needs of customers for a system that helps solve the problem
- Requirements specify what the system is supposed to do but not how the system is to accomplish the task.
- Requirements should be precise, complete, and consistent
  - Precise - They should state exactly what is desired of the system
  - Complete - They should include descriptions of all facilities required
  - Consistent - There should be no conflicts in the descriptions of the system facilities
- Requirement analysis
  - specifies software's operational characteristics
  - indicates software's interface with other system elements
  - establishes constraints that software must meet
- Requirements analysis allows the software engineer to:
  - elaborate on basic requirements established during earlier requirement engineering tasks
  - Build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.
- Requirement can be
  - Functional
  - Non-functional
  - Domain

### 2.1.1 FUNCTIONAL REQUIREMENTS

- Functional requirements describe the services provided by the system, how the system should react to particular inputs and how the system should behave in particular situations.
- The functional requirements may also explicitly state what the system should not do.
- They depend on the type of the system being developed, expected user of the system and approach used for writing the requirements.
- Eg:
    - Compute the transactions (deposit, withdraw, print report etc) in a bank ATM system
    - Compute the salary of the employee in a payroll processing system etc
- The functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions.

### 2.1.2 NON-FUNCTIONAL REQUIREMENTS

- Non functional requirements are the constraints on the services offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Non functional requirements are not directly concerned with specific functions delivered by the system.
- They specify system performance, security, availability, and other emergent properties.

**Types of non-functional requirements**:

1. **Product requirements:**
    - These requirements specify the product behavior.
    - Ex: execution speed, memory requirement, reliability requirements that set out the acceptable failure rate; portability requirements; and usability requirements etc
2. **Organisational requirements**
    - These requirements are derived from policies and procedures of the organization.

- Ex: process standards that must be used; implementation requirements such as the programming language or design method used and delivery requirements that specify when the product and its documentation are to be delivered.

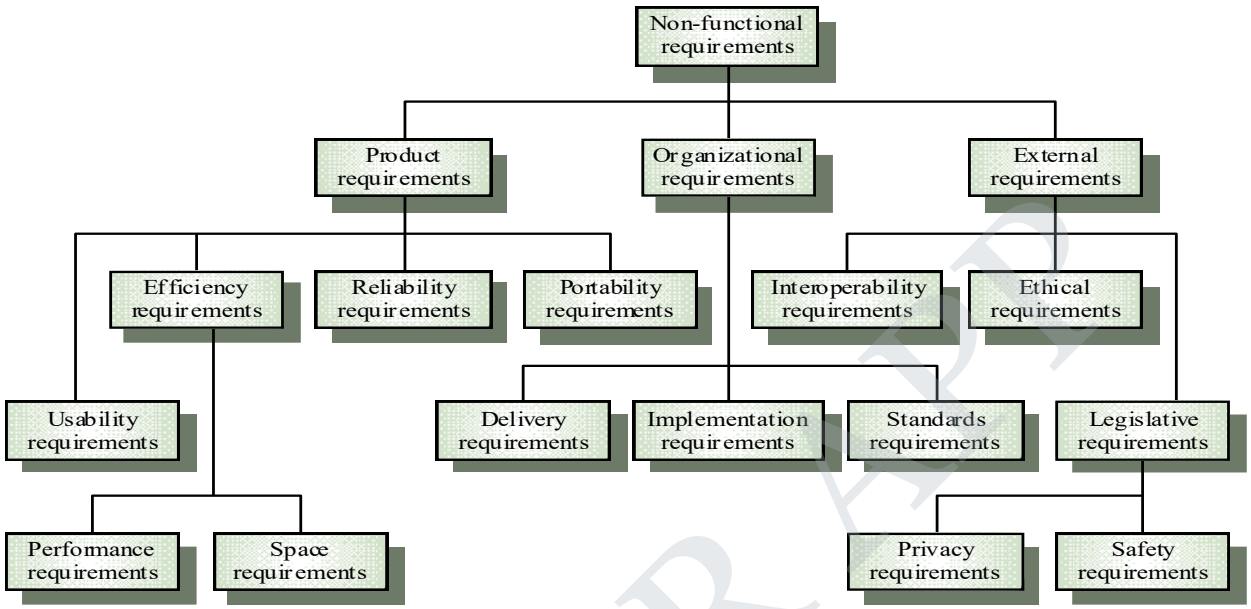



Fig: Non- functional requirements

3. **External requirements**
   - These requirements are derived from factors external to the system and its development process.
   - Eg: interoperability requirements that define how the system interacts with systems in other organizations.

- Non functional requirements are more critical than individual functional requirements. if these are not met, the system is useless
- Eg: if an aircraft system does not meet its reliability requirement, it will not be certified as safe for operation. If a real time control system fails to meet performance requirement, the control functions will not operate properly.
- Problem with non functional requirements is that they are difficult to verify.
- Non-functional requirements often conflict and interact with other functional or non-functional requirements.
- It's better to differentiate functional and non-functional requirements in the requirements document. But it's difficult to do so.

- Metrics used to specify the non functional requirements are:

| Property | Measure |
|---|---|
| Speed | Processed transactions/second<br>User/Event response time<br>Screen refresh time |
| Size | K Bytes<br>Number of RAM chips |
| Ease of use | Training time<br>Number of help frames |
| Reliability | Mean time to failure<br>Probability of unavailability<br>Rate of failure occurrence<br>Availability |
| Robustness | Time to restart after failure<br>Percentage of events causing failure<br>Probability of data corruption on failure |
| Portability | Percentage of target dependent statements<br>Number of target systems |

Fig: Metrics for specifying non- functional requirements

- If the non-functional requirements are stated separately from the functional requirements, it is sometimes difficult to see the relationships between them.
- If they are stated with the functional requirements, then it may be difficult to separate functional and nonfunctional considerations and to identify requirements that relate to the system as a whole
- So the requirements that are clearly related to emergent system properties, such as performance or reliability can be explicitly highlighted by putting them in a separate section of the requirements document or distinguishing them from other system requirements

**Domain requirements**

- These requirements come from the application domain of the system.
- They reflect the characteristics and constraints of that domain
- They may be new functional requirement or non functional requirementss) or set out how particular computations must be carried out
- Eg: LIBSYS
  - There shall be a standard user interface to all databases that shall be based on the Z39.50 standards. → This is a design constraint. So the developer has to find out about the standard before starting the interface design.
  - The deceleration of the train shall be computed as:

$$D_{train} = D_{control} + D_{gradient}$$

- This uses a domain specific terminology. To understand this you need understanding of operation of railway system and train characteristics

- Drawback: domain requirements are written in the language of application domain which is difficult for the software engineer to understand.

### 2.1.3 USER RQUIREMENTS

- User requirements describe what services the system is expected to provide and the constraints under which it must operate.
- These requirements should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge
- They should specify only the external behaviour and avoid the system design characteristics
- User requirements are defined using natural language, tables, and diagrams
- Problems faced with natural language:
  - **Lack of clarity**: It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
  - **Requirements confusion:** Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.
  - **Requirements amalgamation:** Several different requirements may be expressed together as a single requirement.
- This requirement includes both conceptual and detailed information.
- The user requirement should simply focus on the key facilities to be provided.
- A rationale can be associated with each user requirement which explains why the requirement has been included and is particularly useful when requirements are changed
- Guidelines to be followed to minimize misunderstandings when writing user requirements:
  - Use a standard format to define all the requirement.
  - Use language consistently.
  - Use text highlighting (bold, italic or color) to pick out key parts of the requirement.
  - Avoid using computer jargon /technical terms

### 2.1.4 SYSTEM REQUIREMENTS

- System requirements describe the external behavior of the system and its operational constraints.
- This explains how the user requirements should be provided by the system
- System requirement should be a complete and consistent specification of the whole system
- Natural language is often used to write system requirements specifications
- Drawbacks of using natural language specifications for specifying system requirements:
  1. Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstandings because of the ambiguity of natural language.
  2. A natural language requirements specification is over flexible.
  3. It is difficult to modularize natural language requirements. It may be difficult to find all related requirements.

**Notations used for System requirements specification**

- **Structured natural language**
  - This approach depends on defining standard forms or templates to express the requirements specification.
- **Design description languages**
  - This approach uses a language like a programming language but with more abstract feature to specify the requirements by defining an operational model of the system
- **Graphical notations**
  - A graphical language, supplemented by text annotations is used to define the functional requirements for the system. Eg: SADT(structured analysis and design techniques), use case diagram, sequence diagrams etc
- **Mathematical specifications**
  - These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality

**Structured natural language specifications**

- Structured natural language is a way of writing system requirements in a standard format.
- Advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification.
- Structured language notations limit the terminology that can be used and use templates to specify system requirements.
- They may include control constructs derived from programming languages and graphical highlighting to partition the specification.
- When a standard form is used for specifying functional requirements, the following information should be included:
    1. Description of the function or entity being specified
    2. Description of its inputs and where these come from
    3. Description of its outputs and where these go to
    4. Indication of what other entities are used (the *requires* part)
    5. Description of the action to be taken
    6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called
    7. Description of the side effects (if any) of the operation.

**Advantages:**

- Variability in the specification is reduced and requirements are organized more effectively.
- To avoid ambiguity, add extra information to natural language requirements using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system and how sequences of actions are performed.

- Eg : **To compute the insulin dose for a person**

| Insulin Pump/Control Software/SRS/3.3.2 | |
|---|---|
| **Function** | Compute insulin dose: Safe sugar level |
| **Description** | Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units |
| **Inputs** | Current sugar reading (r2), the previous two readings (r0 and r1) |
| **Source** | Current sugar reading from sensor. Other readings from memory. |
| **Outputs** | CompDose–the dose in insulin to be delivered |
| **Destination** | Main control loop |
| **Action:** | CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered. |
| **Requires** | Two previous readings so that the rate of change of sugar level can be computed. |
| **Pre-condition** | The insulin reservoir contains at least the maximum allowed single dose of insulin. |
| **Post-condition** | r0 is replaced by r1 then r1 is replaced by r2 |
| **Side effects** | None |

Fig: system requirements to compute the insulin dosage for a person

- **Tables** are useful when there are a *number of possible alternative situations and actions to be taken*.
- Eg:

| Condition | Action |
|---|---|
| Sugar level falling (r2 < r1) | CompDose = 0 |
| Sugar level stable (r2 = r1) | CompDose = 0 |
| Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0)) | CompDose = 0 |
| Sugar level increasing and rate of increase stable or increasing. ((r2 – r1) > (r1 – r0)) | CompDose = round ((r2 – r1)/4)<br>If rounded result = 0 then<br>CompDose = MinimumDose |

**Fig: sample table for system requirement specification**

- **Graphical models** are most useful to show how state changes and to describe a sequence of actions
- Eg: sequence diagram for ATM withdrawal

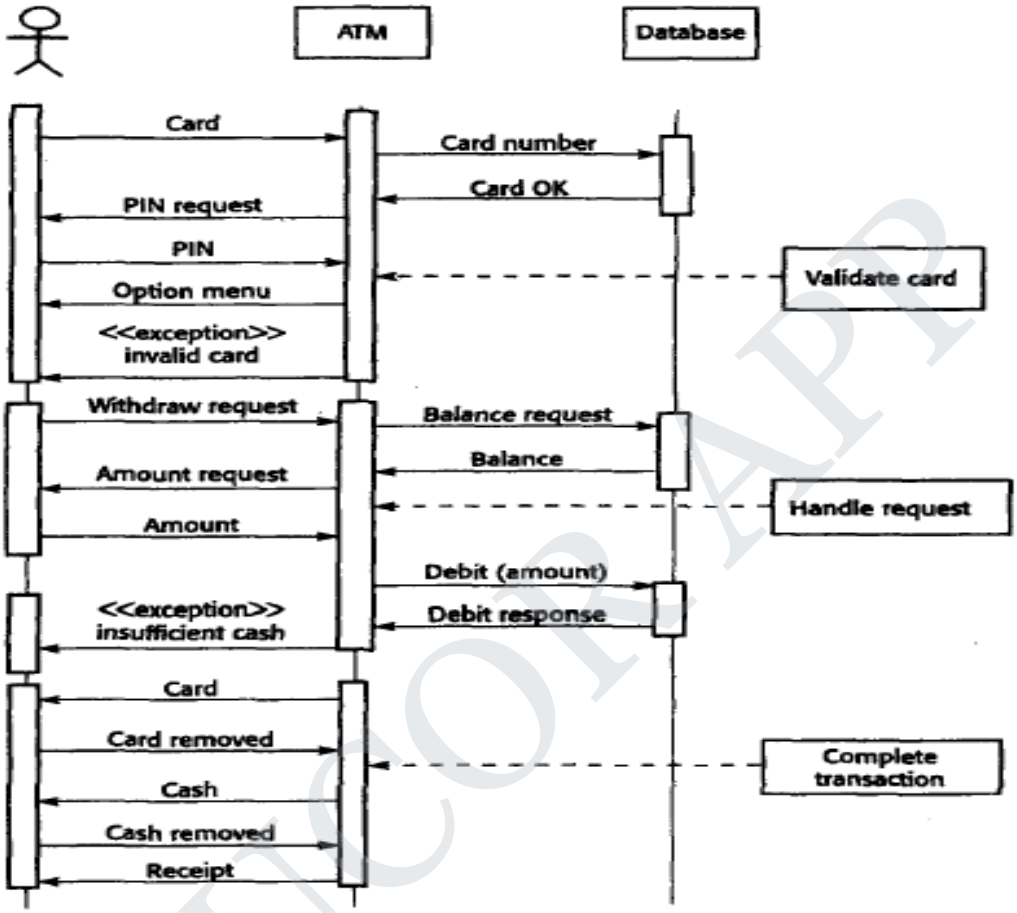



Fig: Sequence diagram for withdraw()

## 2.2 REQUIREMENTS ENGINEERING

- Requirements engineering (RE) is the process of identifying, analyzing, documenting and checking the services and constraints.
- Requirements engineering process is to create and maintain a system requirements document.
- Requirements engineering includes the task and techniques used to understand the basic requirements of the system.

- Process involved in Requirements engineering:

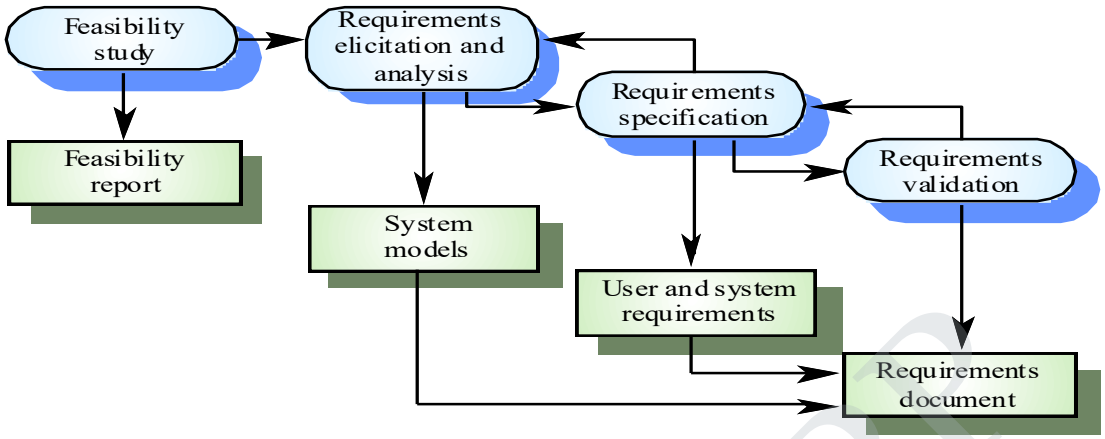



**Fig: Requirements Engineering Process**

1. **Feasibility Study**- concerned with assessing whether the system is useful to the business
2. **Requirements elicitation and analysis** - discovering requirements
3. **Requirements specification** - converting these requirements into some standard form
4. **Requirements Validation** - checking that the requirements actually define the system that the customer wants.

- The requirements engineering process is an iterative process around a spiral.

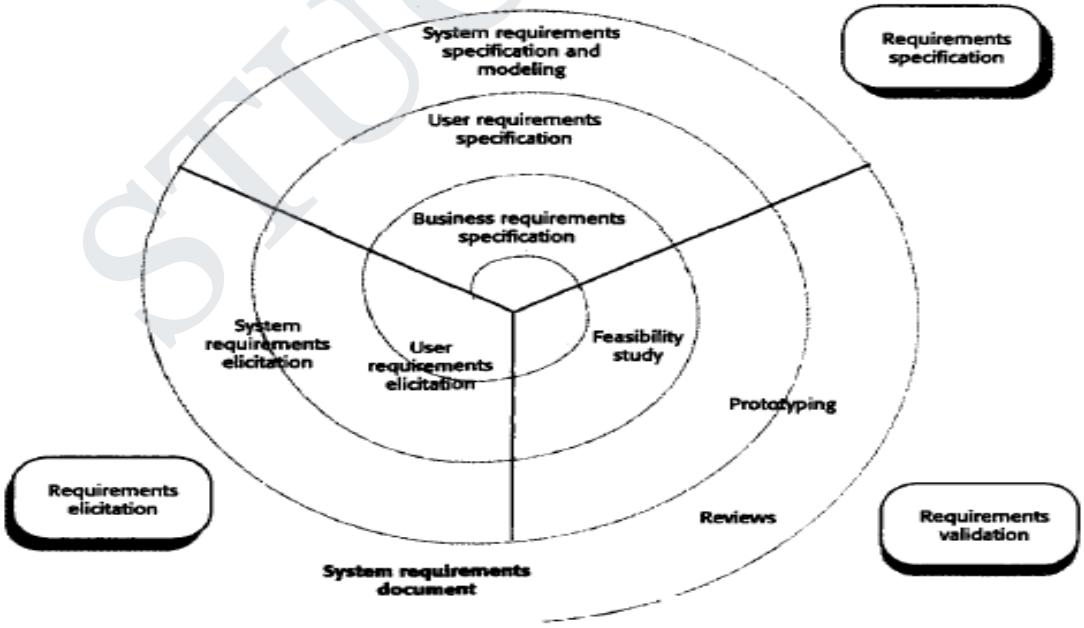



Fig: Requirements Engineering

- The amount of time and effort for each activity in the iteration depends on the stage of the overall process and the type of system being developed.
- Early stage includes understanding the non-functional requirements and the user requirements. Later stages are devoted to system requirements engineering and system modeling.
- The number of iterations around the spiral can vary
- Requirements engineering is the process of applying a structured analysis method such as object-oriented analysis
- This involves analyzing the system and developing a set of graphical system models, such as use-case models, that then serve as a system specification.

### 2.2.1 FEASIBILITY STUDY

- Feasibility study is used to determine if the user needs can be satisfied with the available technology and budget
- Feasibility study checks the following:
  - Does the system contribute to organisational objectives?
  - Can the system can be implemented using current technology and within budget
  - Can the system can be integrated with other systems that are used
- If a system does not support these objectives, it has no real value to the business.
- Feasibility study is based on the information assessment, information collection and report writing.
- Sample Questions that may be asked for information collection are:
  1. What if the system wasn't implemented?
  2. What are current process problems?
  3. How will the proposed system help?
  4. What will be the integration problems?
  5. Is new technology needed? What skills?
  6. What facilities must be supported by the proposed system?
- Information sources are the managers of the departments, software engineers, technology experts and end-users of the system.
- The feasibility study should be completed in two or three weeks.
- After collecting the information, the feasibility report is created.

- In the report, changes to the scope, budget and schedule of the system can be proposed and also suggest further high-level requirements for the system.

### 1.2.2 REQUIREMENTS ELICITATION AND ANALYSIS

- Requirements elicitation is nothing but identifying the application domain, the services that the system should provide and the constraints.
- Requirements are gathered from the *stakeholders*
- Stakeholders are any person or group who will be affected by the system, directly or indirectly. Eg: end-users, managers, engineers, domain experts etc
- Drawbacks of Eliciting and understanding stakeholder requirements:
    1. Stakeholders don't know what they really want.
    2. Stakeholders express requirements in their own terms.
    3. Different stakeholders may have conflicting requirements.
    4. Organisational and political factors may influence the system requirements.
    5. The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

**Steps involved in requirements elicitation and analysis**

1. *Requirements discovery*

   This is the process of interacting with stakeholders in the system to collect their requirements. Domain requirements are also identified.

2. *Requirements classification and organisation*

   This activity takes the unstructured collection of requirements, groups related requirements and organises them into coherent clusters.

3. *Requirements prioritisation and negotiation*

   Since multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing the requirements, and finding and resolving requirements conflicts through negotiation.

4. *Requirements documentation*

   The requirements are documented and input into the next round of the spiral for further requirements discovery. Formal or informal requirements documents may be produced.

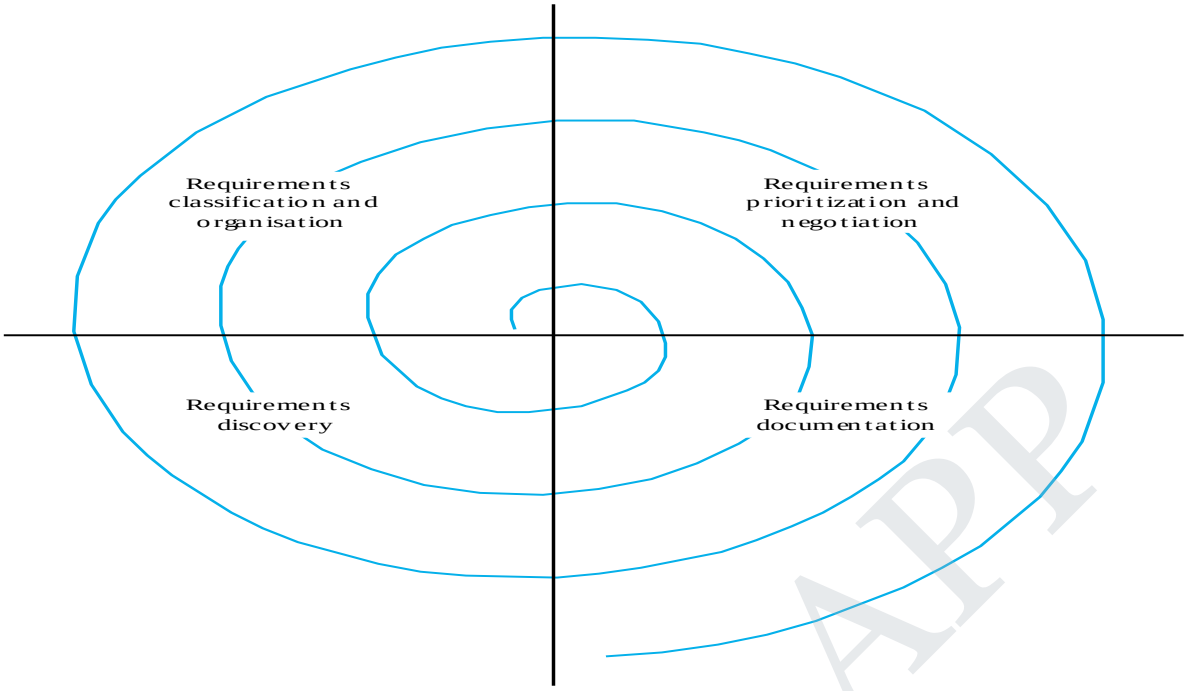



Fig: Requirement Elicitation and Analysis

**Requirements discovery**

- Requirements discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information during the requirements discovery phase include documentation, system stakeholders and specifications of similar systems.
- Stakeholders are interacted through interviews and observation, and may use scenarios and prototypes to help with the requirements discovery.
- Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system.
- For example,

  System stakeholders for a bank ATM include:

  Current bank customers, Representatives from other banks, Managers of bank branches, Counter staff at bank branches, Database administrators, Bank security managers, bank's marketing department, Hardware and software maintenance engineers, National banking regulators etc
- The requirements may also come from the application domain and from other systems that interact with the system being specified.

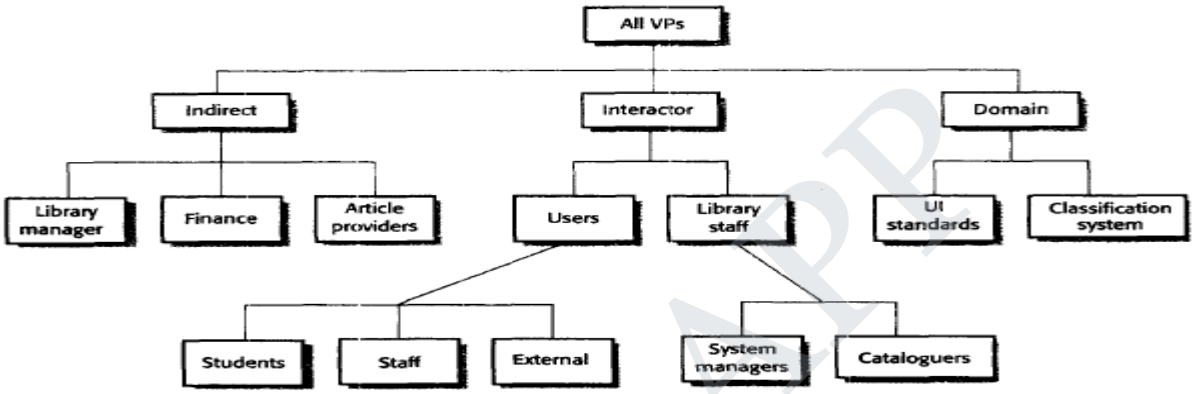
- These requirements sources (stakeholders, domain, systems) can all be represented as **system viewpoints**
- Each viewpoint presents a sub-set of the requirements for the system. Each viewpoint provides a fresh perspective on the system, but these perspectives are not completely independent--they usually overlap so that they have common requirements.

**Viewpoints**

- Viewpoint-oriented approach recognizes multiple perspectives of the stakeholders and provides a framework for discovering conflicts in the requirements proposed by different stakeholders.
- Types of viewpoint:

1. *Interactor viewpoints*
    - This viewpoint represents people or other systems that interact directly with the system.Eg: In bank ATM system, the interactor viewpoints are the bank's customers and the bank's account database.
    - Interactor viewpoints provide detailed system requirements covering the system features and interfaces.
2. *Indirect viewpoints*
    - This represents the views of stakeholders who do not use the system themselves but who influence the requirements in some way.
    - Eg: In the bank ATM system, the indirect viewpoints are the management of the bank and the bank security staff.
    - Indirect viewpoints are more likely to provide higher-level organisational requirements and constraints
3. *Domain viewpoints*
    - *This represents* the domain characteristics and constraints that influence the system requirements.
    - *Eg:* In the bank ATM system, the domain viewpoint would be the standards that have been developed for interbank communications.
    - Domain viewpoints normally provide domain constraints that apply to the system

- Almost all organisational systems must interoperate with other systems in the organisation. When a new system is planned, the interactions with other systems must be planned which may place requirements and constraints on the new system.
- Finally organise and structure the viewpoints into a hierarchy.
- Eg: viewpoint hierarchy for LIBSYS




- Once viewpoints have been identified and structured, identify the most important viewpoints and start with them to discover the system requirements.

**Interviewing**

- RE team asks questions to stakeholders about the system they are using and the system to be developed and derive the requirements from their answers.
- Interviews may be of two types:

  1. **Closed interviews** where the stakeholder answers a predefined set of questions.

  2. **Open interviews** where there is no predefined agenda and a range of issues are explored with stakeholders.
- Completely open-ended discussions rarely work well; most interviews require some questions to get started and to keep the interview focused on the system to be developed.
- Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the system and the difficulties that they face with current systems.
- Interviews are not good for understanding the domain requirements
- Interviews are not an effective technique for eliciting knowledge about organisational requirements and constraints
- It is hard to elicit domain knowledge during interviews for two reasons:
  - Requirements engineers cannot understand specific domain specific terminology;

- Some domain knowledge is so familiar that people find to explain or they think it is so fundamental that it isn't worth mentioning.

**Scenarios**

- Scenarios are real-life examples of how a system can be used.
- Scenarios are useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions.
- Each scenario covers one or more possible interactions.
- The scenario starts with an outline of the interaction, and, during elicitation, details are added to create a complete description of that interaction.
- Scenarios should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong and how it is handled;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.
- Scenarios may be written as text, diagrams, screen shots etc
- Alternatively, a more structured approach such as event scenarios or usecases may be adopted

**Use cases**

- Use-cases are a scenario based technique in the UML (Unified Modelling Language) notation for describing object oriented system model.
- Use cases identify the actors involved and the type of interaction between them.
- Eg:

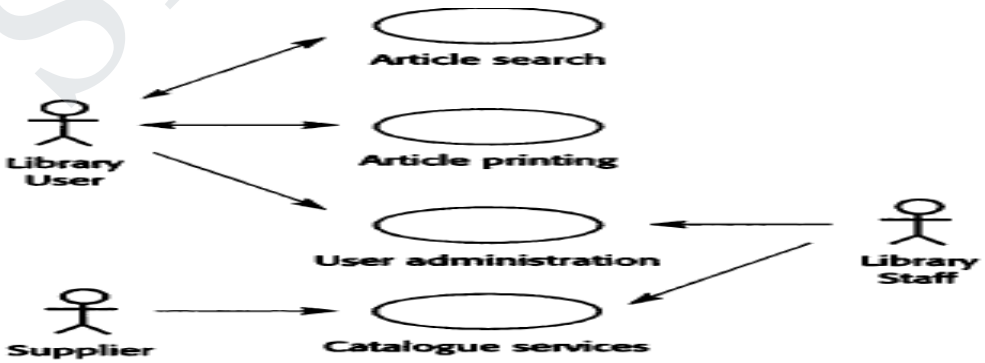



Fig: Use case diagram for article printing

- A set of use cases should describe all possible interactions with the system.
- Actors in the process are represented as stick figures, and each class of interaction is represented as a named ellipse.

**Sequence diagrams**

- Sequence diagrams are used to add detail to use-cases by showing the sequence of event processing in the system.
- Eg:

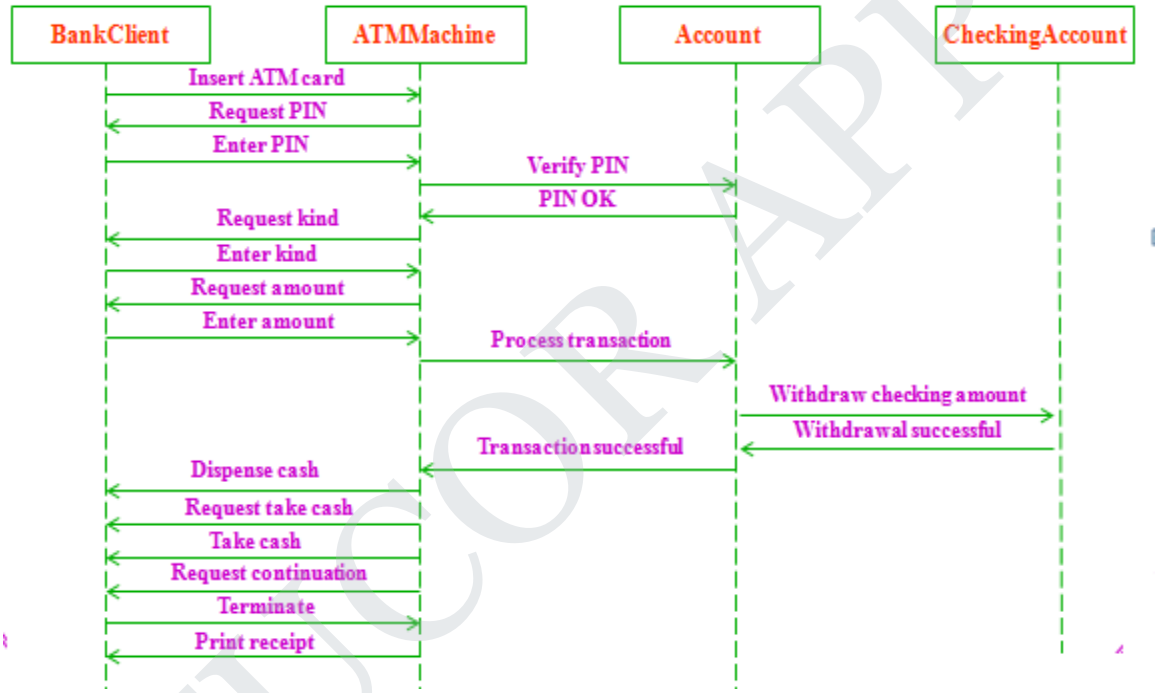



Fig: Sequence diagram for withdraw from ATM

**Ethnography**

- *Ethnography* is an observational technique that can be used to understand social and organisational requirements.
- Ethnography helps analysts to discover implicit system requirements that reflect the actual rather than the formal processes in which people are involved.
- Ethnography is particularly effective at discovering two types of requirements:
  - Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work.
  - Requirements that are derived from cooperation and awareness of other people's activities

- Ethnography may be combined with prototyping.

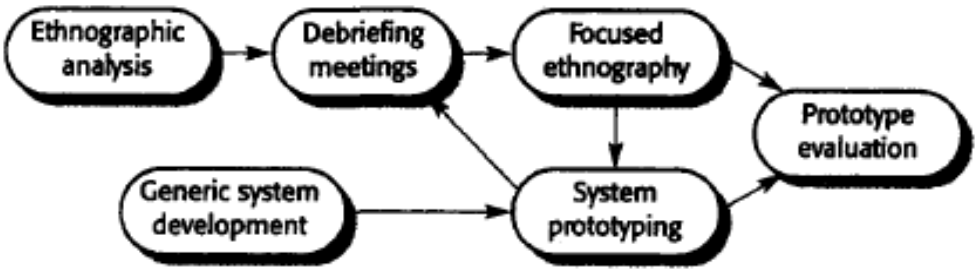



Fig: Ethnography process

- The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required.
- Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer.
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques.
- This approach is not appropriate for discovering organisational or domain requirements as it focuses on the end users.

### 2.2.3 SOFTWARE REQUIREMENTS DOCUMENT/ SOFTWARE SPECIFICATION

- Also referred as software requirements specification or SRS
- The software requirements document is the official statement of what the system developers should implement.
- It should include both the user requirements for a system and a detailed specification of the system requirements.
- The user and system requirements may be integrated into a single description.
- The user requirements are defined in an introduction to the system requirements specification.
- If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

- Users of the requirements document are:

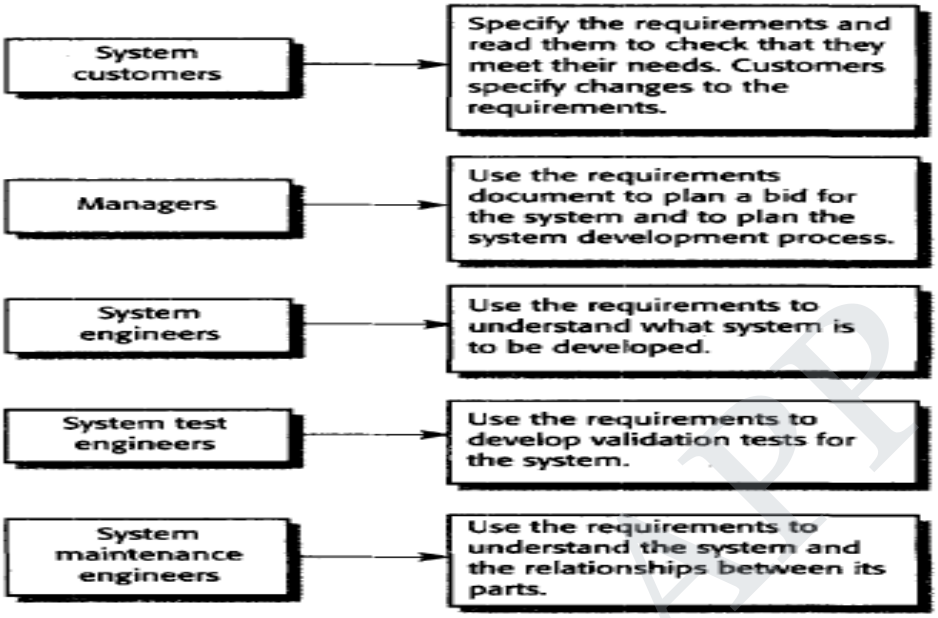



Fig: Users of SRS

- Requirements document helps in communicating the requirements to customers, defining the requirements in detail for developers and testers, and including information about possible system evolution.
- Requirements information can help system designers to avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.
- The level of detail to be included in a requirements document depends on the type of system that is being developed and the development process used.
- When the system will be developed by an external contractor, the system specifications need to be precise and very detailed.
- When there is more flexibility in the requirements and an iterative development process is used, the requirements document can be much less detailed
- **IEEE standard suggests the following structure for requirements documents:**

1. **Introduction**

    1.1 Purpose of the requirements document

    1.2 Scope of the product

    1.3 Definitions, acronyms and abbreviations

1.4 References

1.5 Overview of the remainder of the document

2. **General description**

2.1 Product perspective

2.2 Product functions

2.3 User characteristics

2.4 General constraints

2.5 Assumptions and dependencies

3. **Specific requirements** cover functional, non functional and interface requirements.

The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. **Appendices**

5. **Index**

- It is a general framework that can be tailored and adapted to define a standard to the needs of a particular organization
- Requirements documents are essential when an outside contractor is developing the software system.
- The focus will be on defining the user requirements and high-level, non-functional system requirements.
- When the software is part of a large system engineering project that includes interacting hardware and software systems, it is essential to define the requirements to a fine level of detail.

### 2.2.4 REQUIREMENTS VALIDATION

- Requirements validation is concerned with showing that the requirements actually define the system that the customer wants.
- Requirements error costs are high so validation is very important
- When the requirements change that the system design and implementation must also be changed and then the system must be tested again. So the cost of fixing a requirement problem is greater than repairing the design or coding errors

- Requirements checking:

1. *Validity checks*
   - A user needs the system to perform certain functions.
   - Does the system provide the functions which best support the customer's needs?
2. *Consistency checks*
   - Are there any requirements conflicts?
3. *Completeness checks*
   - The requirements document should include requirements, which define all functions, and constraints intended by the system user.
   - Are all functions required by the customer included?
4. *Realism checks*
   - Can the requirements be implemented given available budget and technology?
5. *Verifiability*
   - To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.
   - Can the requirements be checked?

**Requirements validation techniques**

1. *Requirements reviews*

   The requirements are analysed systematically by a team of reviewers.
2. *Prototyping*

   An executable model of the system is demonstrated to end-users and customers to see if it meets their real needs
3. *Test-case generation*

   Requirements should be testable. This approach is for developing tests for requirements to check testability.

   If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, then the requirements will be difficult to implement and should be reconsidered.

**Requirements reviews**

- A requirements review is a manual process that involves people from both client and contractor organisations to check the requirements document for anomalies and omissions.
- Requirements reviews can be informal or formal.
- Informal reviews involve contractors discussing requirements with the system stakeholders. Good communications can help to resolve problems at an early stage.
- In a formal requirements review, the development team explains the implications of each requirement to the client. The review team should check each requirement for consistency as well as completeness.
- Reviewers may also check for:
  1. **Verifiability**. Is the requirement realistically testable?
  2. **Comprehensibility**. Is the requirement properly understood?
  3. **Traceability**. Is the origin of the requirement clearly stated? Traceability is important as it allows the impact of change on the rest of the system to be assessed
  4. **Adaptability**. Can the requirement be changed without a large impact on other requirements?
- Conflicts, contradictions, errors and omissions in the requirements should be pointed out by reviewers and formally recorded in the review report.
- It is then up to the users, the system procurer and the system developer to negotiate a solution to these identified problems.

## 1.3 REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements management is the process of understanding and controlling changes to system requirements.
- Requirements are incomplete and inconsistent because:
  - New requirements arise during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.

- Requirements may change over time for the following reasons:
  - The priority of requirements from different viewpoints changes during the development process.
  - System customers may specify requirements from a business perspective that conflict with end-user requirements.
  - The business and technical environment of the system changes during its development.

**Types of requirements:**

- Enduring requirements.

  These are stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models

- Volatile requirements.

  These are requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

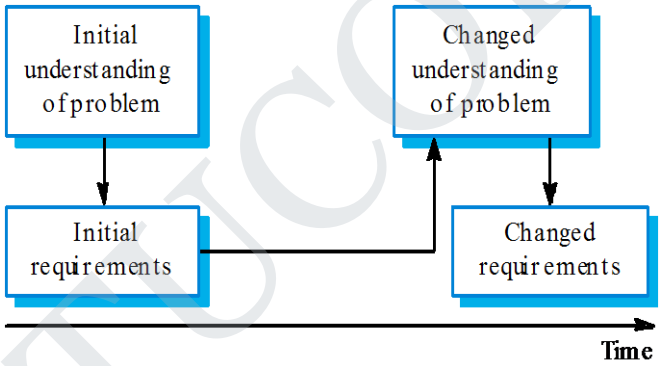



**Fig: Requirement Evolution**

- **Mutable requirements**

  Requirements that change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.

- **Emergent requirements**

  Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements

- **Consequential requirements**

  Requirements that result from the introduction of the computer system. Introducing the computer system may change the organisations processes and open up new ways of working which generate new system requirements

- **Compatibility requirements**

  Requirements that depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

**Requirements management planning:**

- During the requirements engineering process, you have to plan:
  - Requirements identification
    - Each requirement must be uniquely identified so that it can be cross-referenced by other requirements and so that it may be used in traceability assessments.
  - A change management process
    - This is the set of activities that assess the impact and cost of changes
  - Traceability policies
    - These policies define the relationships between requirements, and between the requirements and the system design that should be recorded and how these records should be maintained.
  - CASE tool support
    - The tool support required to help manage requirements change;

**Traceability**

- Traceability is concerned with the relationships between requirements, their sources and the system design
- **Types of traceability information**:
  - **Source traceability**
    - Links from requirements to stakeholders who proposed these requirements;
  - **Requirements traceability**
    - Links between dependent requirements;

- This information helps to assess how many requirements are likely to be affected by a proposed change

- **Design traceability**
  - Links from the requirements to the design;
  - This information helps to assess the impact of proposed requirements changes on the system design and implementation.

➢ **Fig: Traceability Matrix**

| Req. id | 1.1 | 1.2 | 1.3 | 2.1 | 2.2 | 2.3 | 3.1 | 3.2 |
|---|---|---|---|---|---|---|---|---|
| 1.1 | | D | R | | | | | |
| 1.2 | | | D | | | D | | D |
| 1.3 | R | | | R | | | | |
| 2.1 | | | R | | D | | | D |
| 2.2 | | | | | | | | D |
| 2.3 | | R | | D | | | | |
| 3.1 | | | | | | | | R |
| 3.2 | | | | | | | R | |

**CASE tools support for:**

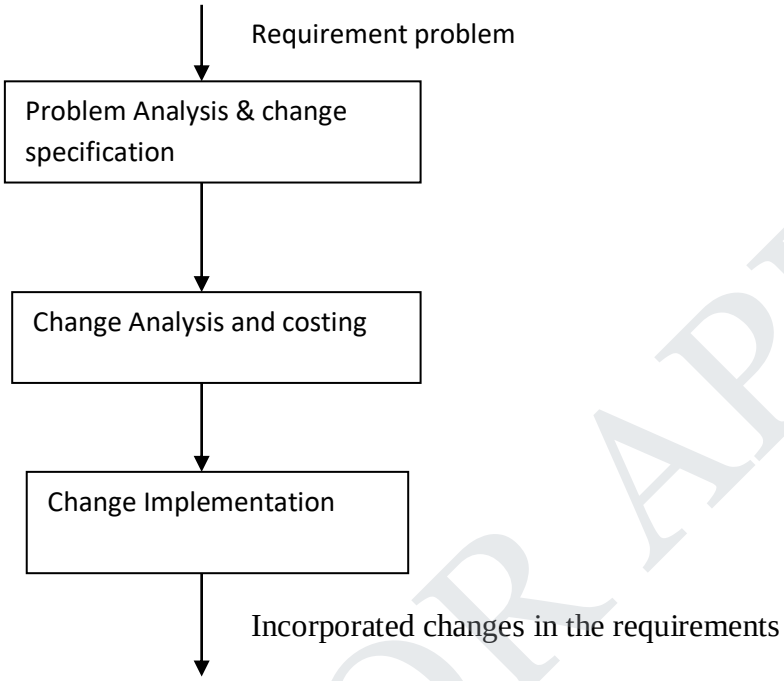
- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is simplified using CASE tools.
- Traceability management
  - Tools help to discover possible relationship between the requirements.

## 2.4 REQUIREMENTS CHANGE MANAGEMENT

- Requirements change management should be applied to all proposed changes to the requirements
- Stages
  - **Problem analysis**. Identifying the requirements problem and propose change;
  - **Change analysis and costing**. Assess effects of change on other requirements and estimate the cost of the impact using the traceability information

- **Change implementation**. Modify requirements document and other documents to reflect change

**Fig: Requirements change management process**




## 2.5 SYSTEM MODELING

### 2.5.1 STRUCTURED SYSTEM ANALYSIS

- System models are graphical representations that describe business processes, the problem to be solved and the system that is to be developed.
- Graphical representations are often more understandable than detailed natural language descriptions of the system requirements
- **Different models present the system from different perspectives**
  - External perspective showing the system's context or environment
  - Behavioural perspective showing the behaviour of the system
  - Structural perspective showing the system or data architecture

**Types of analysis models:**

- Structured analysis
- Object-oriented analysis

**Three primary objectives of the analysis model:**

- to describe what the customer requires

- to establish a base for the creation of a software design
- to define a set of requirements that can be validated

**Elements of Analysis model:**

- **The core - data dictionary**

  - this contains the descriptions of all data objects in the software

- **The entity-relationship diagram (ERD)**

  - ERD specifies the attributes of data objects and also depicts the relationships between data objects.

- **The data flow diagram (DFD)**

  - DFD provides an indication of how data are transformed as they move through the system.

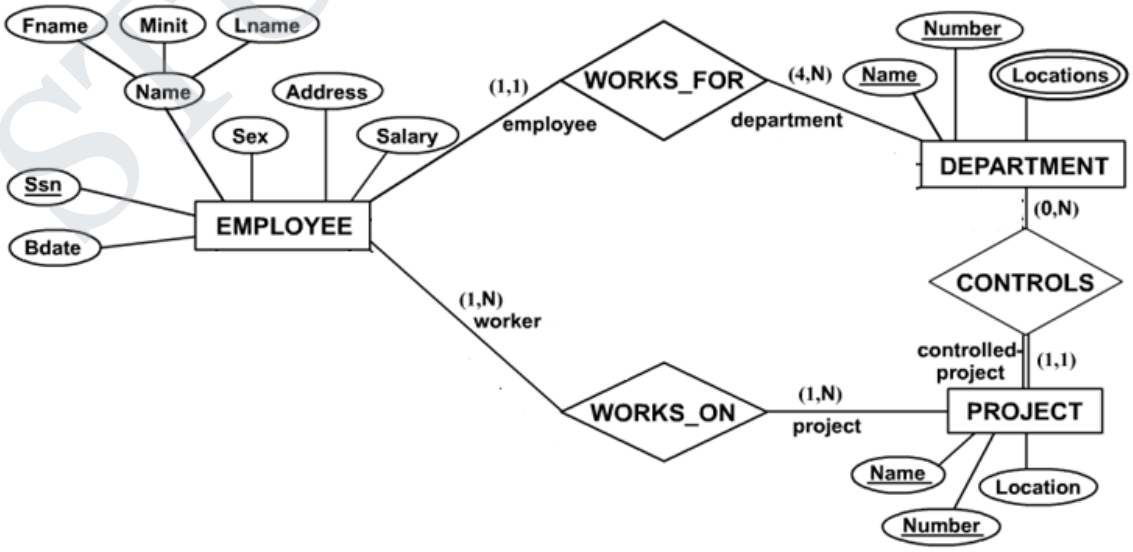
  - DFD depicts the functions that transform the data flow.

- **The state-transition diagram(STD)**

  -This indicate how the system behaves as a consequence of external events

**DATA MODELLING:**

- Entity-Relationship Diagram is a very useful method for data modeling.
- It represents:
  - data objects, object attributes, and relationships between objects
- **Eg:**



**PROCESS MODELLING**

- Process model represents the system's function.
- This graphically represent the process that capture, manipulate, store and distribute data between the system and its environment
- Eg: Data Flow Diagram

**DATA FLOW DIAGRAM**

- A Data Flow Diagram (DFD) is a graphical representation of the flow of data through an information system, modeling its *process* aspects.
- This is used to create an overview of the system

**Elements of DFD:**

- **external entity** - people or organisations that send data into the system or receive data from the system
- **process** - models what happens to the data i.e. transforms incoming data into outgoing data
- **data store** - represents permanent data that is used by the system
- **data flow** - models the actual flow of the data between the other elements

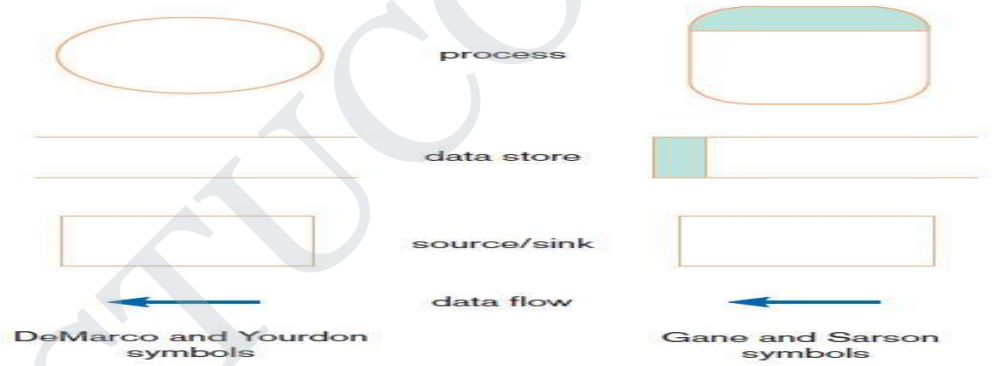



Fig: Symbols used in DFD

**DFD Diagramming rules:**

- Data flow can take place between processes from
  - data store to a process and a process to a data store
  - external entity to a process and from process to an external entity
- Data flows can't take place between 2 data stores or two external entities
- Data flows are unidirectional
- A process must have at least one input and one output

**Developing DFDs**

- **Context diagram** is an overview of an organizational system that shows:
  - The system boundaries.
  - External entities that interact with the system.
  - Major information flows between the entities and the system.
- Note: only one process symbol, and no data stores shown

**Eg:**

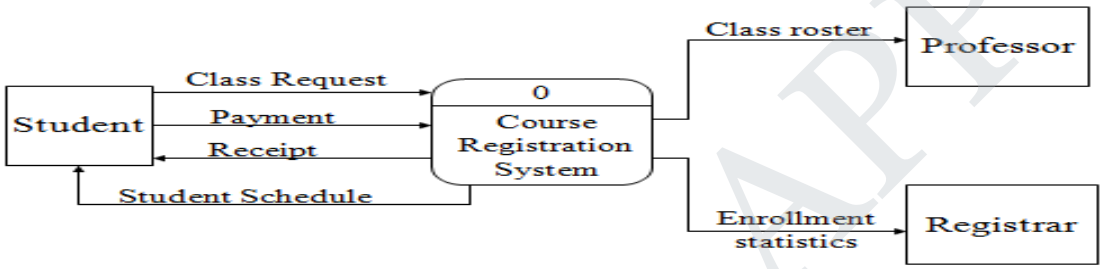



**Fig: Context level DFD for course registration system**

- **Level-1 diagram** is a data flow diagram that represents a system's major processes, data flows, and data stores at a high level of detail.

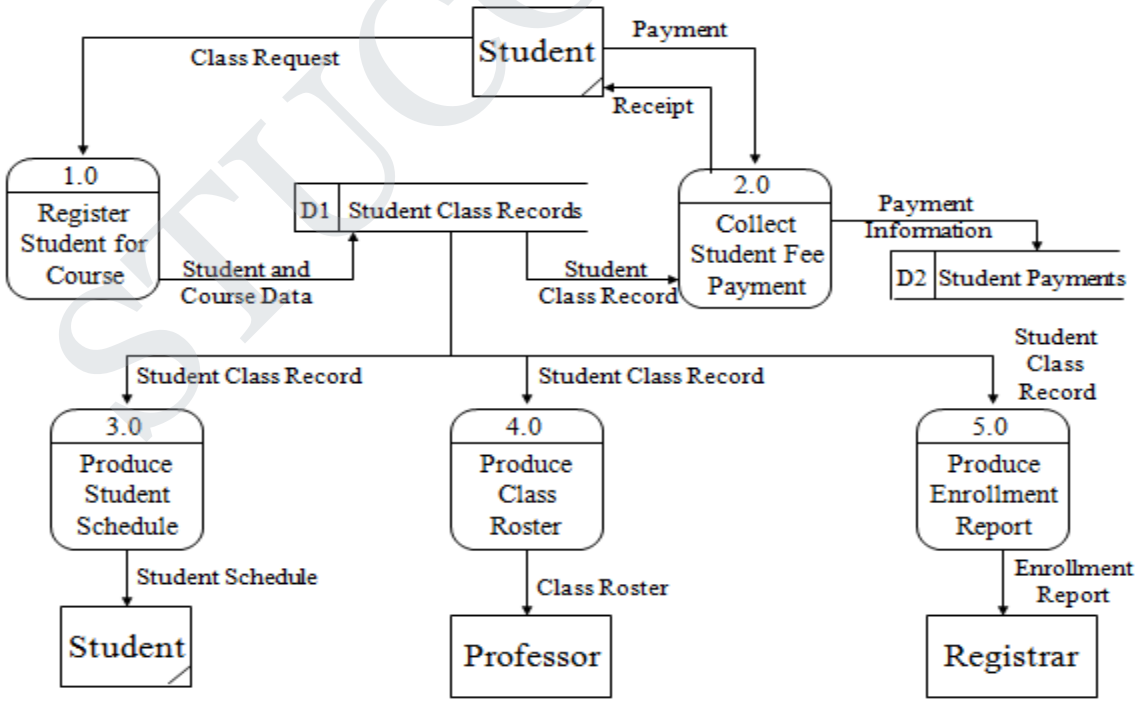



**Fig: Level – 1 DFD**

- **Level-n Diagram** shows how the system is divided into sub-systems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole.
- **Eg:**

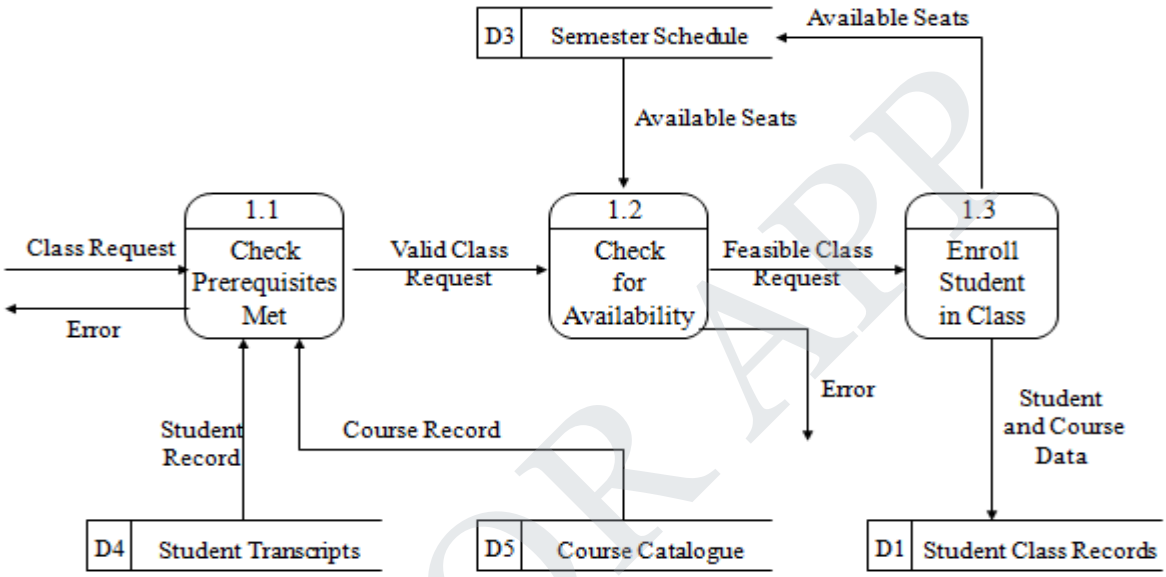



Fig: **level 2 DFD for Process 1: Register student for the course**

**Class Diagram**

- A class is a collection of objects with common structure, common behavior, common relationships and common semantics
- Classes are found by examining the objects in sequence and collaboration diagram
- A class is drawn as a rectangle with three compartments
- A class diagram shows the existence of classes and their relationships in the logical view of a system
- UML modeling elements in class diagrams
  - Classes and their structure and behavior
  - Association, aggregation, dependency, and inheritance relationships
  - Multiplicity and navigation indicators
  - Role names

- Eg: ATM system

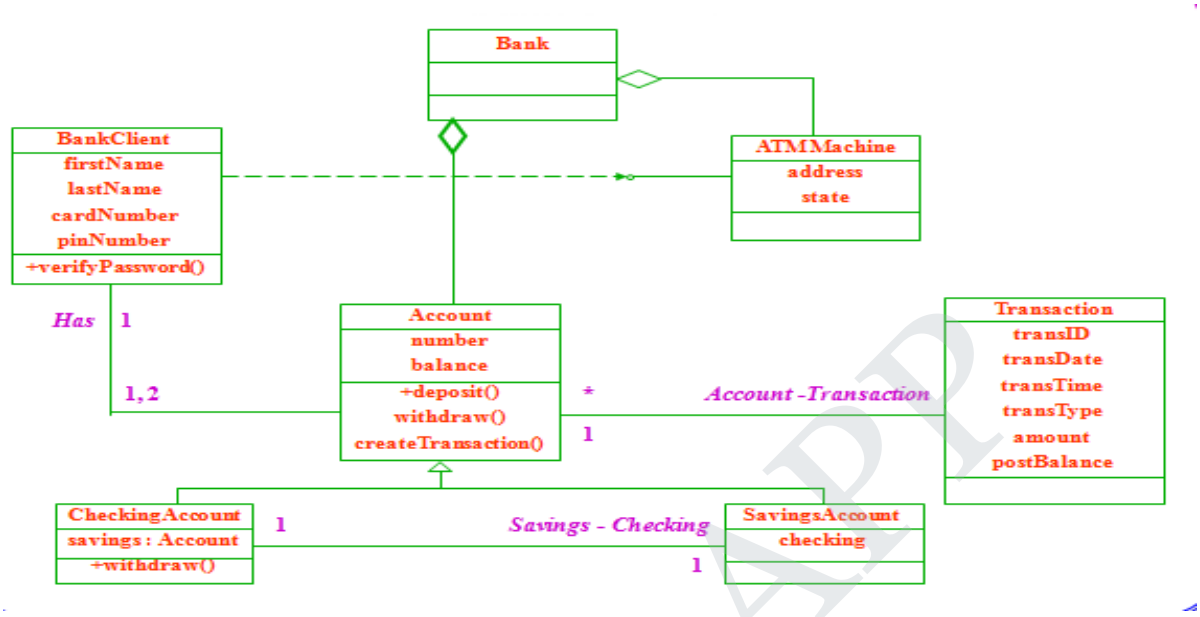



Fig: Class diagram for ATM System

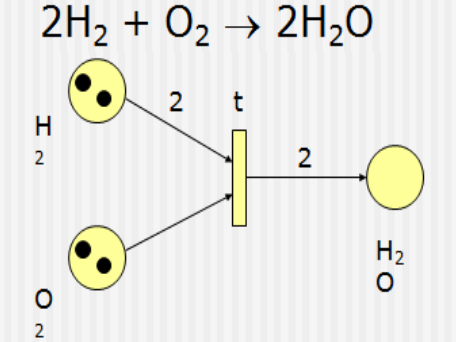
### 2.5.2 PETRI NETS

- A **Petri net** is also known as a **place/transition net**
- It is a diagrammatic tool to model concurrency and synchronization in distributed systems.
- It is similar to State Transition Diagrams.
- This is used as a visual communication aid to model the system behaviour.
- This is a mathematical modeling language for the description of distributed systems
- Petri nets can be used to model complex processes
- Petri nets can be simulated in order to illustrate and test system behaviour, benchmark its speed etc.

**Components:**

- **Places** (circles): Places represent possible states of the system;
- **Transitions** (rectangles): events or actions which cause the change of state;
- **Arcs** (arrows): Used to connect a place with a transition or a transition with a place.
- Change of state is denoted by a movement of ***tokens*** (black dots) from place to place and is caused by the ***firing*** of a transition.
- The firing represents an occurrence of the event or an action taken.
- The firing is subject to the input conditions, denoted by token availability.

- A transition is *firable* or *enabled* when there are sufficient tokens in its input places.
- After firing, tokens will be transferred from the input places (old state) to the output places, denoting the new state.
- Eg: firing event



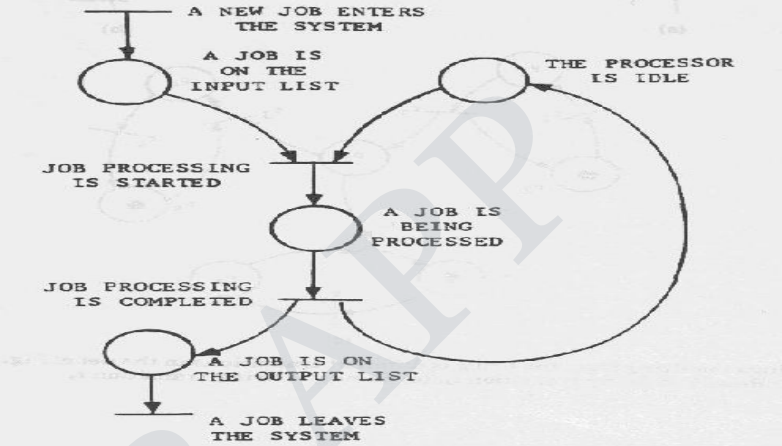
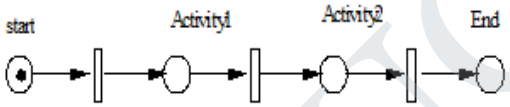



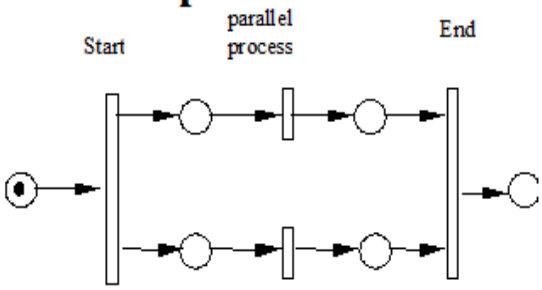

FIGURE 6. Modeling of a simple computer system.

Primitive structures that occur in real systems:

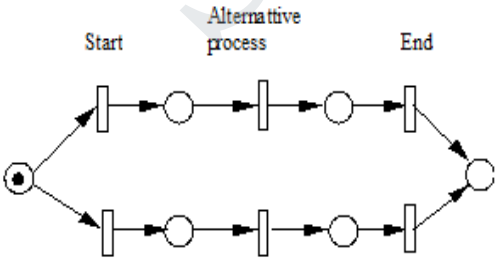
**Precedence relation:**



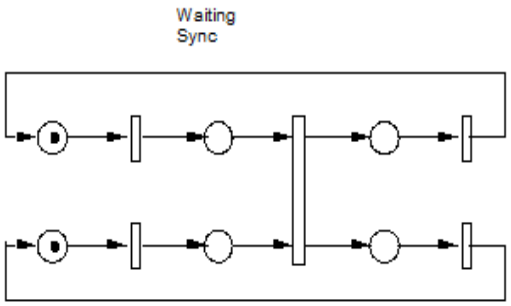

**Parallel processes:**




**Alternative processes:**




**Synchronization:**



## 2.3 DATA DICTIONARIES

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included
- Collects and coordinates data terms, and confirms what each term means to different people in the organization
- Many CASE workbenches support data dictionaries
- The data dictionary may be used for the following reasons:
  - Provide documentation
  - Eliminate redundancy
  - Validate the data flow diagram
  - Provide a starting point for developing screens and reports
  - To develop the logic for DFD processes
- Data dictionaries contain
  - Data flow
  - Data structures
  - Elements
  - Data stores

**Defining Data Flow**

- Each data flow should be defined with descriptive information and it's composite structure or elements

Eg:

| | |
|---|---|
| Name | Customer Order |
| Description | Contains customer order information and is used to update the customer master and item files and to produce an order record. |
| Source | Customer External Entity |
| Destination | Process 1, Add Customer Order |
| Type | Screen |
| Data Structure | Order Information |
| Volume/Time | 10/hour |
| Comments | An order record contains information for one customer order. The order may be received by mail, fax, or by telephone. |

Fig: Data flow definition for customer order

- Include the following information:
  - ID - identification number

- Label, the text that should appear on the diagram
- A general description of the data flow
- The source and destination of the data flow
- Type of data flow
- The name of the data structure describing the elements
- The volume per unit time
- An area for further comments and notations

**Defining Data Structures**

- Data structures are a group of smaller structures and elements
- An algebraic notation is used to represent the data structure
- The symbols used are
  - Equal sign, meaning “consists of”
  - Plus sign, meaning "and”
  - Braces {} meaning repetitive elements, a repeating element or group of elements
  - Brackets [] for an either/or situation- The elements listed inside are mutually exclusive
  - Parentheses () for an optional element
- A repeating group may be
  - A sub-form
  - A screen or form table
  - A program table, matrix, or array
- There may be one repeating element or several within the group
- Data structure may be logical or physical data structure
  - **Logical:** Show what data the business needs for its day-to-day operations
  - **Physical**: Include additional elements necessary for implementing the system

Eg: Data structure for customer details

Customer name = First name + (middle name) + last name

Address= Street + (Apartment) + City + State +Zip

**Data Element**

Data element includes the following:

- Element ID

- The name of the element
- Aliases -Synonyms or other names for the element
- A short description of the element
    - Element is base or derived
        - A base element is one that has been initially keyed into the system
        - A derived element is one that is created by a process, usually as the result of a calculation or a series of decision-making statements
- Element length
- Type of data - Alphanumeric or text data
- Input and output formats - using coding symbols:
    - Z - Zero suppress
    - 9 - Number
    - X - Character
    - X(8) - 8 characters
    - . , - Comma, decimal point, hyphen
- Validation criteria - Ensure that accurate data are captured by the system
- Default value - Include any default value the element may have. The default value is displayed on entry screens
- An additional comment or remark area

**Data Stores**

- Data stores are created for each different data entity being stored
- When data flow base elements are grouped together to form a structural record, a data store is created for each unique structural record
- Because a given data flow may only show part of the collective data that a structural record contains, many different data flow structures may need to be examined to arrive at a complete data store description
- A data store has:
    - The data store ID
    - The data store name
    - An alias for the table
    - A short description of the data store

- The file type
- File format

Eg:

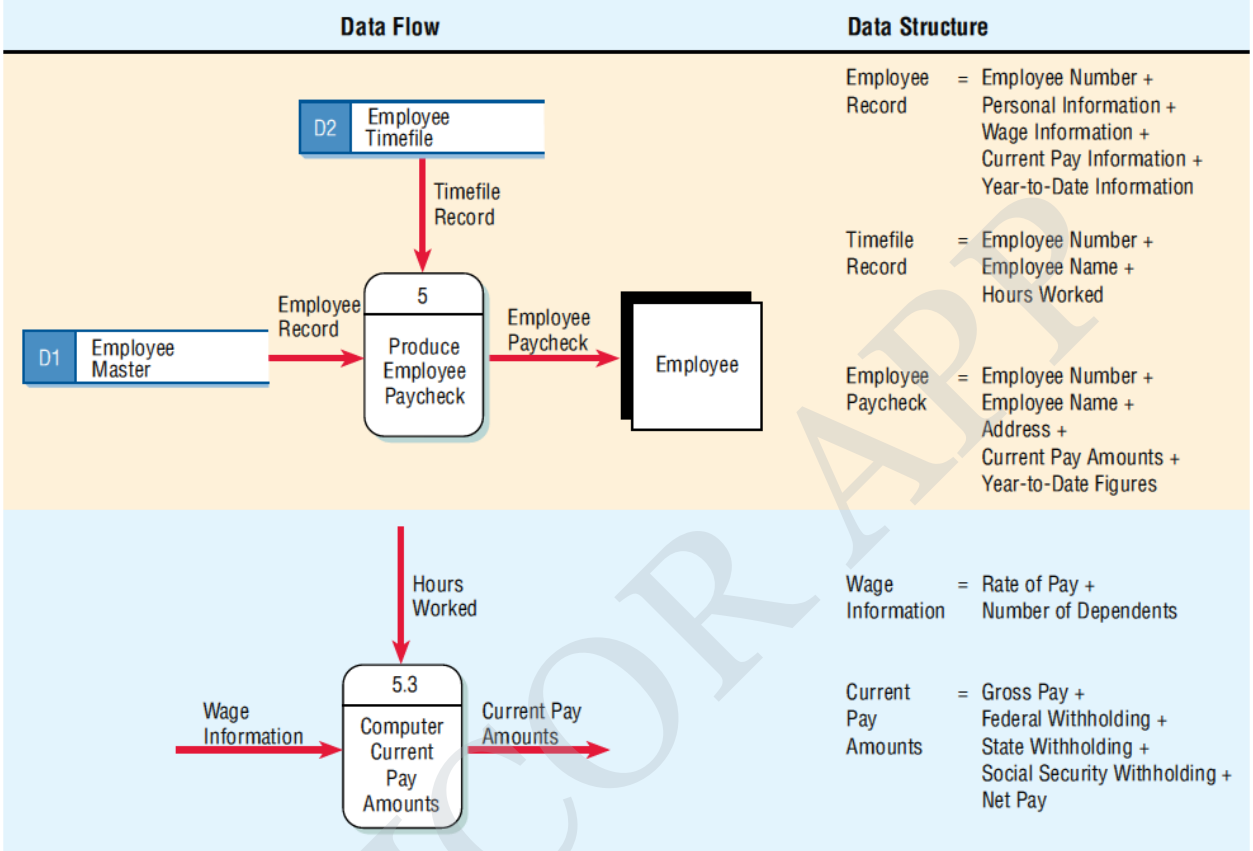



Fig: Sample data dictionary for employee database

# UNIT-III

# SOFTWARE DESIGN

## 3.1 DESIGN PROCESS

- Software design consists of the set of principles, concepts, and practices which is used for developing the system or product
- Software design is an iterative process through which requirements are translated into a blueprint for constructing the software

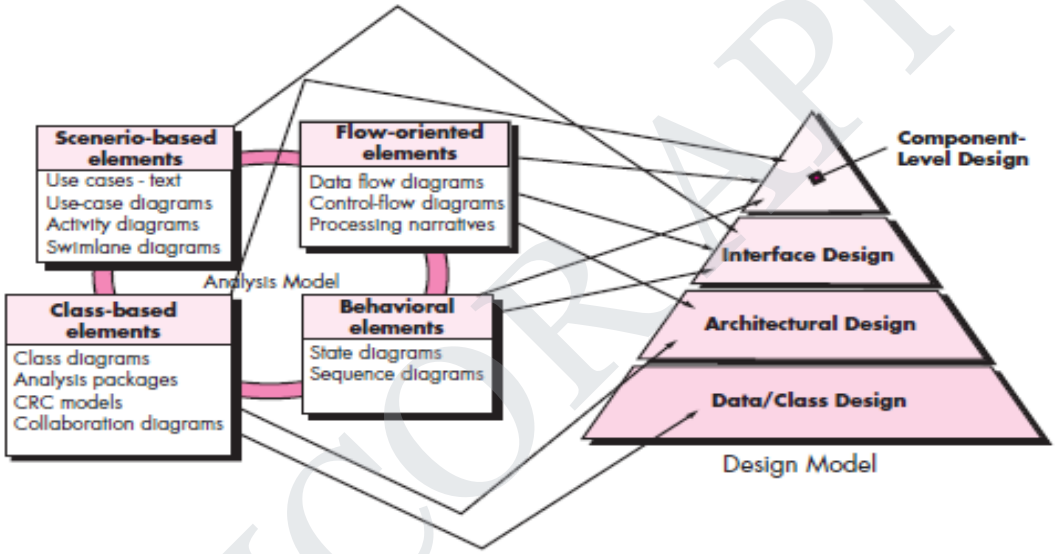



Fig: Translating the Requirements model into Design model

- Initially the design is represented at a high level of abstraction and as iteration occurs, the design shows lower levels of abstraction.
- The design model provides detail about the software data structures, architecture, interfaces, and components

**Characteristics of a good design**

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

**Quality Guidelines**

- A design should exhibit an architecture that is created using recognizable architectural styles or patterns and composed of components that exhibit good design characteristics and can be implemented in an evolutionary fashion
- A design should be modular
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

**Design Principles**

- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize semantic errors.

**Quality Attributes. [FURPS]**

- **Functionality** is assessed by evaluating the feature set and capabilities of the program and the security of the overall system etc.
- **Usability** is assessed by considering human factors, consistency and documentation.

- **Reliability** is evaluated by measuring the frequency and severity of failure, the mean-time-to-failure (MTTF), the ability to recover from failure etc.
- **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability

**The Evolution of Software Design**

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner.
- Procedural aspects of design definition evolved into structured programming
- Later work proposed methods for the translation of data flow into a design definition.
- Newer design approaches proposed an object-oriented approach to design derivation.
- The design patterns can be used to implement software architectures and lower levels of design abstractions.
- Aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs.

**Characteristics of design methods:**

(1)A mechanism for the translation of the requirements model into a design representation,

(2) A notation for representing functional components and their interfaces,

(3) Heuristics for refinement and partitioning, and

(4) Guidelines for quality assessment.

## 3.2 DESIGN CONCEPTS

Concepts that has to be considered while designing the software are:

**Abstraction, Modularity, Architecture, pattern, Functional independence, refinement, information hiding, refactoring and design classes**

### 3.2.1 Abstraction

- There are many levels of abstraction
- At the highest level of abstraction, a solution is stated in broad terms and at lower levels of abstraction, a more detailed description of the solution is provided.

Types:

- **Procedural abstraction** refers to a sequence of instructions that have a specific and limited function
  - Example: open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).
- **Data abstraction** is a named collection of data that describes a data object
  - Eg: Data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

### 3.2.2 Architecture

- Software architecture is the overall structure of the software and the ways in which that structure provides conceptual integrity for a system

**Properties:**

- **Structural properties.** This representation defines the components of a system and how they interact with one another.
- **Extra-functional properties.** The architectural design should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems

**Models to represent the Architectural design:**

- Structural models represent architecture as an organized collection of program components.
- Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- Dynamic models address the behavioural aspects of the program architecture, indicating how the structure changes when an external event occurs.
- Process models focus on the design of the business or technical process that the system must accommodate.
- Functional models can be used to represent the functional hierarchy of a system.

- A number of different architectural description languages (ADLs) have been developed to represent these models

### 3.2.3 Patterns

- Design pattern is a design structure that solves a particular design problem within a specific context
- It provides a description to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

### 3.2.4 Separation of Concerns

- Any complex problem can be more easily handled by subdividing it into pieces which can be solved independently
- A concern is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller manageable pieces, a problem takes less effort and time to solve.

### 3.2.5 Modularity

- Modularity is the single attribute of software that allows a program to be intellectually manageable
- Module is a separately named and addressable components that are integrated to satisfy requirements (divide and conquer principle)

### 3.2.6 Information Hiding

- Information hiding is that the algorithms and local data contained within a module are inaccessible to other modules

### 3.2.7 Functional Independence

- Functional independence is achieved by developing modules that have a "single-minded" function having less interaction with other modules
- Independent modules are easier to maintain and test and error propagation is also reduced.
- Independence is assessed using two qualitative criteria: **cohesion and coupling**.
- *Cohesion* is an indication of the relative functional strength of a module.
- *Coupling* is an indication of the relative interdependence among modules.

- A module should have **high cohesion and low coupling**
- High cohesion – a module performs only a single task
- Low coupling – a module has the lowest amount of connection needed with other modules

### 3.2.8 Refinement

- Refinement is a process of *elaboration*
- It is the development of a program by successively refining levels of procedure detail
- This complements abstraction, which enables a designer to specify procedure and data and suppress low-level details
- Refinement helps to reveal low-level details as design progresses.

### 3.2.9 Aspects

- An *aspect* is a representation of a cross-cutting concern
- These concerns include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts

### 3.2.10 Refactoring

- Refactoring is a reorganization technique that simplifies the design of a component without changing its function or external behaviour
- It removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures

### 3.2.11 Object-Oriented Design Concepts

- The object-oriented (OO) paradigm is widely used in modern software engineering.
- Some of the OO design concepts are
  - Design classes
  - Inheritance
  - Message passing
  - Polymorphism etc

### 3.2.12 Design Classes

- Design classes refines the analysis classes by providing design detail that will enable the classes to be implemented

- Creates a new set of design classes that implement a software infrastructure to support the business solution

**Types of Design Classes**

- **User interface classes** – define all abstractions necessary for human-computer interaction
- **Business domain classes** – refined from analysis classes; identify attributes and methods that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

**Characteristics of a Well-Formed Design Class**

- Complete and sufficient
- Primitiveness - Each method of a class focuses on accomplishing one service for the class
- High cohesion
- Low coupling

## 3.3 DESIGN MODEL

- The design model can be viewed in two different dimensions
  - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed
  - (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Elements of the design model use many of the same UML diagrams used in the analysis model
  - The diagrams are refined and elaborated as part of the design
  - More implementation-specific detail is provided

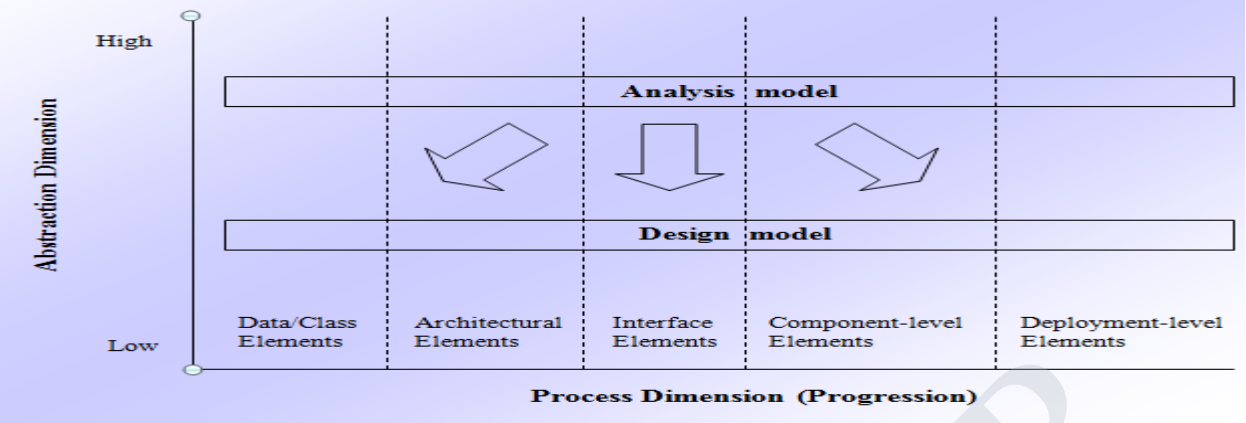
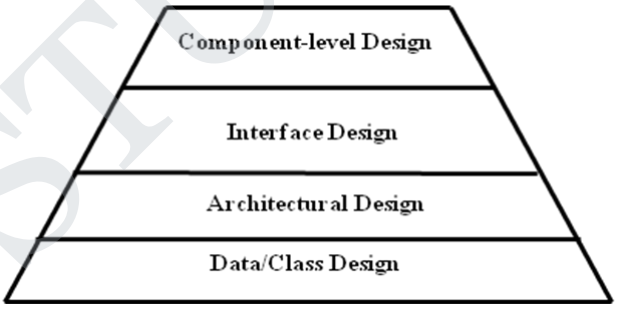



Fig: Dimensions of the Design Model

- In Design model the emphasis is placed on
  - Architectural structure and style
  - Interfaces between components and the outside world
  - Components that reside within the architecture
- The design model has the following layered elements
    - Data/class design
    - Architectural design
    - Interface design
    - Component-level design
- A fifth element that follows all of the others is deployment-level design




### 3.3.1 DESIGN ELEMENTS

**Data/class design elements**

- Data design creates a model of data and objects that is represented at a high level of abstraction

- This data model is then refined into progressively more implementation-specific representations
- At the program component level, the design of data structures and the associated algorithms are used for the creating the high-quality applications.
- At the application level, the translation of a data model into a database is essential to achieve the business objectives of a system.
- At the business level, the collection of information stored in different databases and reorganized into a data warehouse enables data mining or knowledge discovery.

**Architectural design elements**

- Architectural design depicts the overall layout of the software
- The architectural model is derived from three sources:

  (1) Information about the application domain

  (2) Specific requirements model elements such as data flow diagrams and their relationships

  (3) The availability of architectural styles and patterns.
- The architectural design element is usually represented as a set of interconnected subsystems, each have its own architecture

**Interface design elements**

- Interface design tells how information flows into and out of the system and how it is communicated among the various components of the architecture
- Elements of interface design:
  - user interface,
  - external interfaces, and
  - internal interfaces.
- *UI design* (called as usability design) incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).
- *External interface design* requires information about the entity to which the information is communicated. The design should incorporate error checking and security features.

- *Internal interface design* is related with component-level design. Design realizations represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes.

**Component-level design elements**

- Component level design describes the internal detail of each software component like data structure definitions, algorithms, and interface specifications.
- A UML activity diagram can be used to represent processing logic.
- Pseudocode or some other diagrammatic form (e.g., flowchart or box diagram) is used to represent the detailed procedural flow between the component.
- Algorithmic structure follows the rules established for structured programming
- E.g.

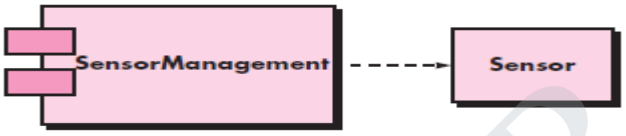



Fig: Component diagram

**Deployment-level design elements**

- This design indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software

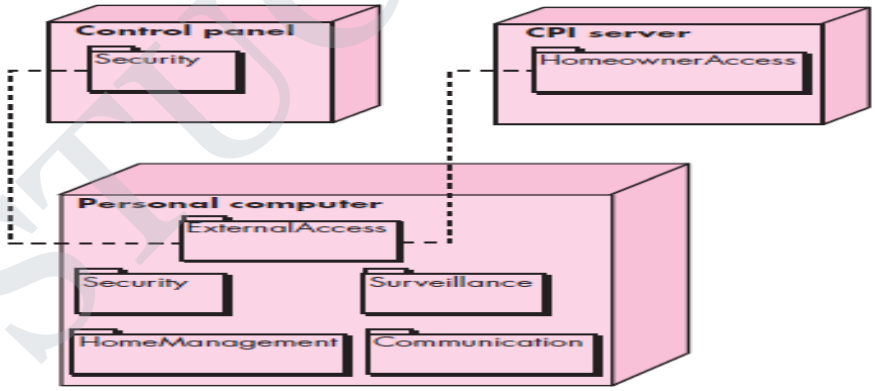



Fig: Deployment Diagram

### 3.4 DESIGN HUERISTIC

1. **Evaluate the first iteration of the program structure to reduce the coupling and improve cohesion**
   - The module independency can be achieved by exploding or imploding the modules
   - Exploding means obtaining more than one modules in the final stage
   - Imploding means combining the results of different modules
2. **Attempt to minimize the structures with high fan-out and strive for fan-in as depth increases**
   - At the higher level of program structure the distribution of control should be made.
   - Fan in means number of immediate ancestors the modules have
   - Fan –out means number of immediate subordinates to the module.
3. **Keep scope of the effect of a module within the scope of control of that module**
   - The decisions made in one module should not affect the other module which is not inside its scope
4. **Evaluate the module interfaces to reduce complexity and redundancy and improve consistency**
   - The major cause of errors is module interfaces. They should simply pass the information and should be consistent with the module
5. **Define module whose function is predictable but avoid modules that are too restrictive**
   - The modules should be designed with simple processing logics and should not restrict the size of the local data structure, control flow or modes of external interfaces
6. **Strive for controlled entry modules by avoiding pathological connections**
   - Interfaces should be constrained and controlled. Pathological connection means many references or branches into the middle of the module

## 3.5 ARCHITECTURAL DESIGN

- The software architecture is the overall structure of the system which is includes the software components , their properties and the relationships among the components
- Software architectural design represents the structure of the data and program components that are required to build a computer-based system
- An architectural design model is transferable
  - It can be applied to the design of other systems
  - It represents a set of abstractions that enable software engineers to describe architecture in predictable ways
- For deriving the architecture, horizontal and vertical partitioning are required

**Horizontal Partitioning**

- Horizontal Partitioning define separate branches of the module hierarchy for each major function
- This use control modules to coordinate communication between functions
- Eg:

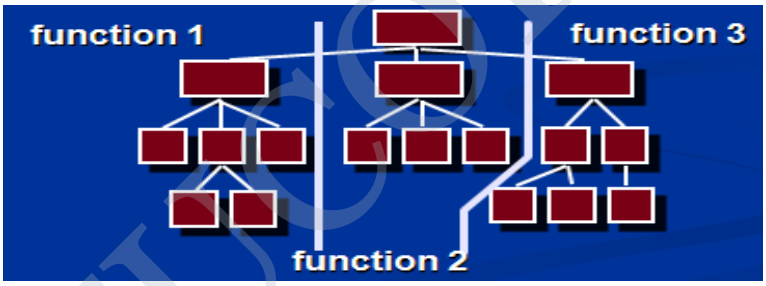



**Fig: Horizontal Partitioning**

**Vertical Partitioning: Factoring**

- Vertical partitioning suggests the control and work should be distributed top-down in the program structure.
- The decision making modules should reside at the top of the architecture

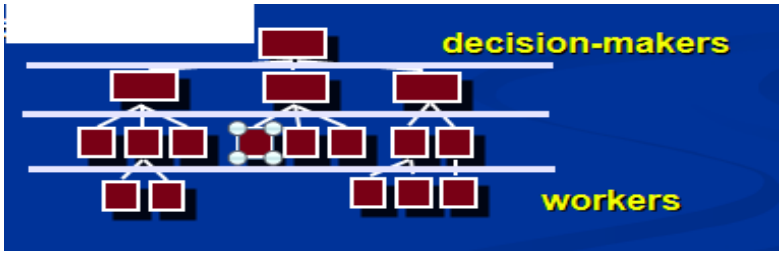



**Fig: Vertical Partitioning**

### 3.5.1 ARCHITECTURAL STYLES

- The architectural styles is used to describes a system which has :
  - A set of components that perform a function required by the system
  - A set of connectors enabling communication among components
  - Semantic constraints that define how components can be integrated to form the system
  - A topological layout of the components indicating their runtime interrelationships

**Taxonomy of Architectural styles**

**Data-Centered Architecture**

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or modify data within the store.
- The main goal is to integrate the data
- The shared data may be a passive repository or an active blackboard
- Passive repository means the client software accesses the data independent of any changes to the data or the actions of other client software.
- A variation on this approach transforms the repository into a "blackboard"
  - A blackboard notifies subscriber clients when changes occur in data of interest
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients

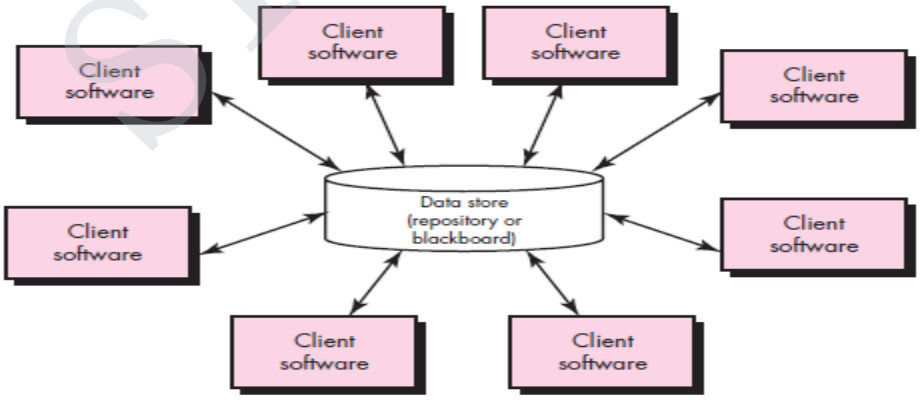



Fig: Data centered architectural style

**Advantages:**

- Data-centered architectures promote *integrability*
- Data can be passed among clients using the blackboard mechanism.
- Client components can execute the processes independently.

**Data Flow Architectures**

- This architecture is used when input has to be transformed into output through a series of computational components.
- The main goal is modifiability
- **Pipe-and-filter style**
  - A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next.
  - The filters incrementally transform the data
  - The filter does not require knowledge of the workings of its neighboring filters.

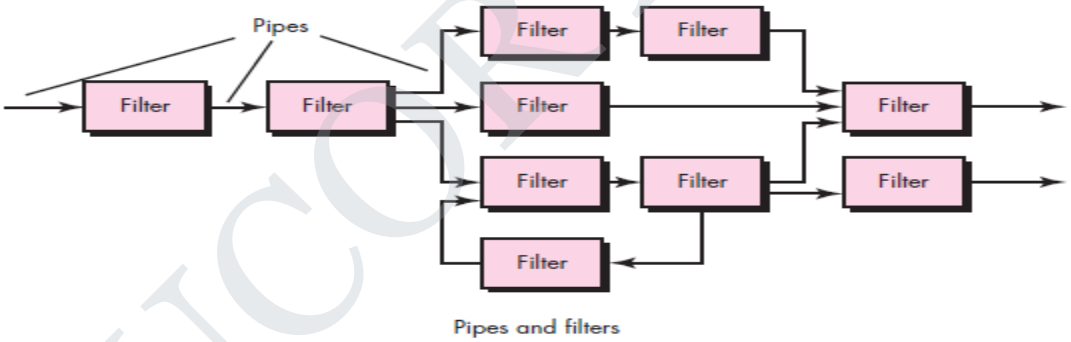



Fig: Pipe and filter style

- **Batch sequential style**
  - If the data flow degenerates into a single line of transforms, it is termed batch sequential.
  - This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.


**Call-and-Return Architectural Style**

- This architectural style enables to achieve a program structure that is relatively easy to modify and scale.

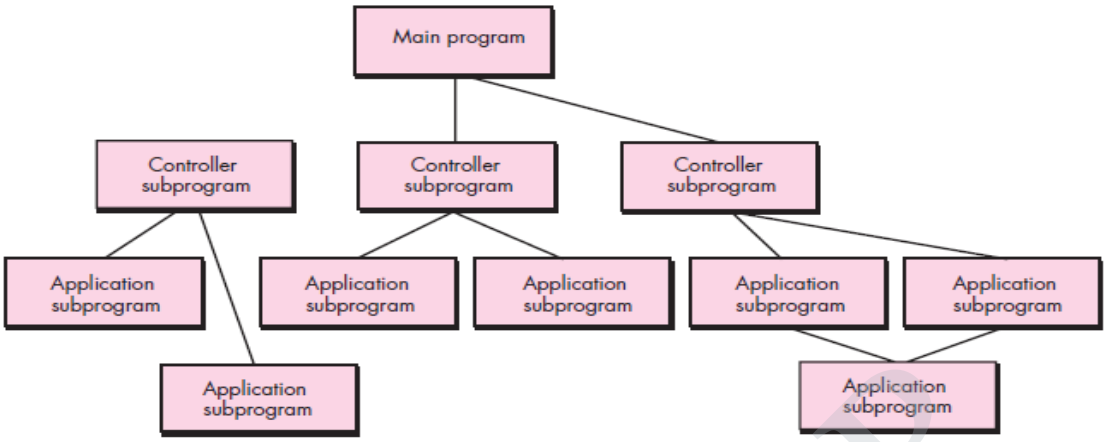



Fig: Call and return architectural style

- Substyles of call and return architecture:
  - *Main program/subprogram architectures.*

    This style decomposes the function into a control hierarchy where a main program invokes a number of program components which in turn may invoke other components.
  - *Remote procedure call architectures.*

    The components of a main program/subprogram architecture are distributed across multiple computers on a network.

**Object-oriented architectures.**

- The components of a system encapsulate data and the operations that are applied to manipulate the data.
- Communication between components are accomplished through message passing.

**Layered architectures.**

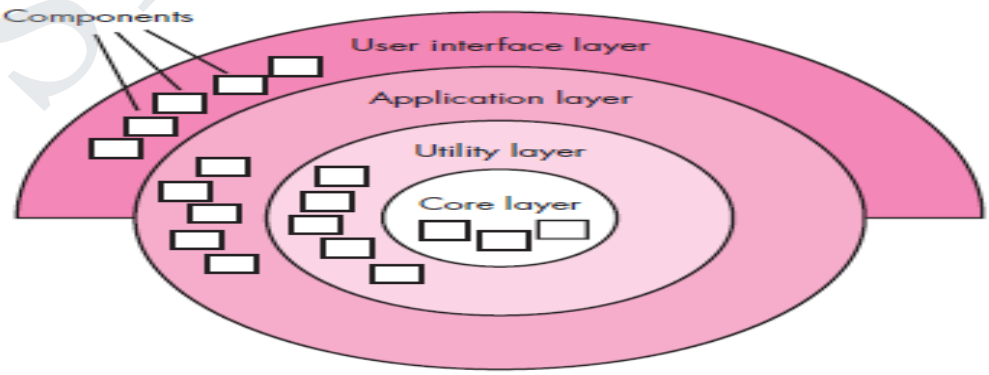



**Fig: Layered architecture**

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

Note:

- Once requirements engineering uncovers the characteristics and constraints of the system, then the appropriate architectural style can be chosen.
- For example, a layered style can be combined with a data-centered architecture in many database applications.

### 3.5.2 ARCHITECTURAL DESIGN PROCESS

**Architectural Design Steps**

1) Represent the system in context
2) Define archetypes
3) Refine the architecture into components
4) Describe instantiations of the system

**1. Represent the System in Context**

- To represent the system in context an architectural context diagram (ACD) is used that shows
  - The identification and flow of all information into and out of a system
  - The specification of all interfaces
  - Any relevant support processing from other systems
- An ACD models in such a way that the software interacts with entities external to its boundaries
- An ACD identifies systems that interoperate with the target system
  - Super-ordinate systems are systems that use the target system as part of some higher level processing scheme
  - Sub-ordinate systems are systems that are used by target system and provide necessary data or processing

- Peer-level systems are those systems that interact on a peer-to-peer basis with target system to produce or consume data
- Actors are the people or devices that interact with target system to produce or consume data

Eg: Safe Home project

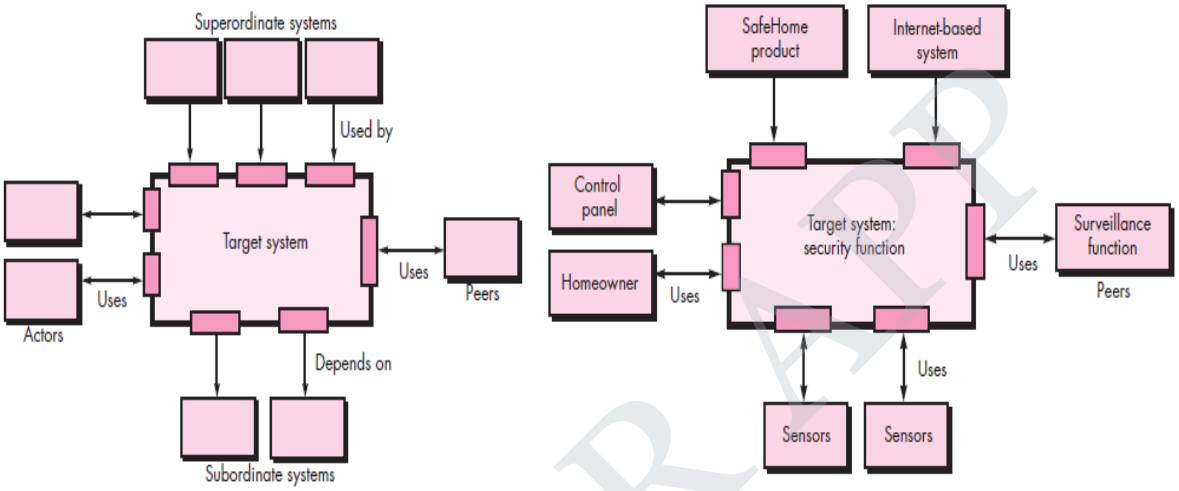



Fig: Architectural context diagram

**2. Define Archetypes**

- Archetypes indicate the abstractions within the problem domain.
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- It includes a common structure and class-specific design strategies and a collection of example program designs and implementations
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes
  - They represent stable elements of the architecture
  - They may be instantiated in different ways based on the behavior of the system
  - They can be derived from the analysis class model
- The archetypes and their relationships can be **illustrated in the fig below**

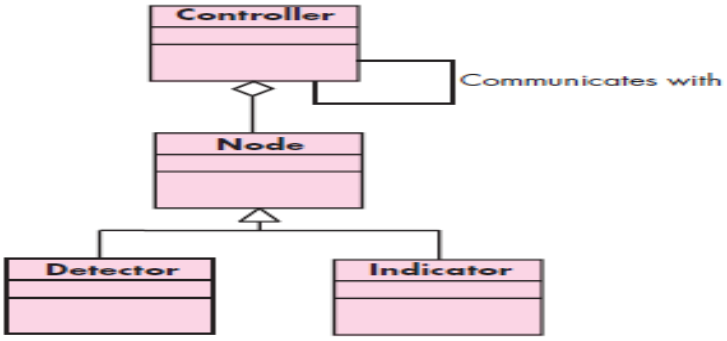
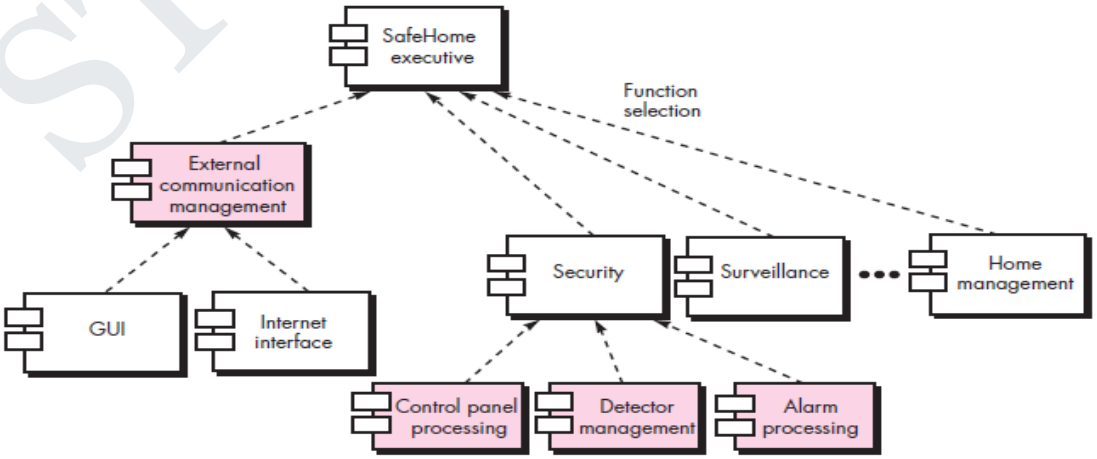



**Fig: UML Class diagram**

**3. Refine the Architecture into Components**

- The architectural designer refines the software architecture into components to describe the overall structure and architectural style of the system
- These components are derived from various sources
  - The application domain provides components, which are the domain classes in the analysis model that represent entities in the real world
  - The infrastructure domain provides design components that enable application components but have no business connection
    - Examples: memory management, communication, database, and task management
  - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface
- A UML class diagram can represent the classes of the refined architecture and their relationships



**4. Describe Instantiations of the System**

- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent the instantiation of the system.

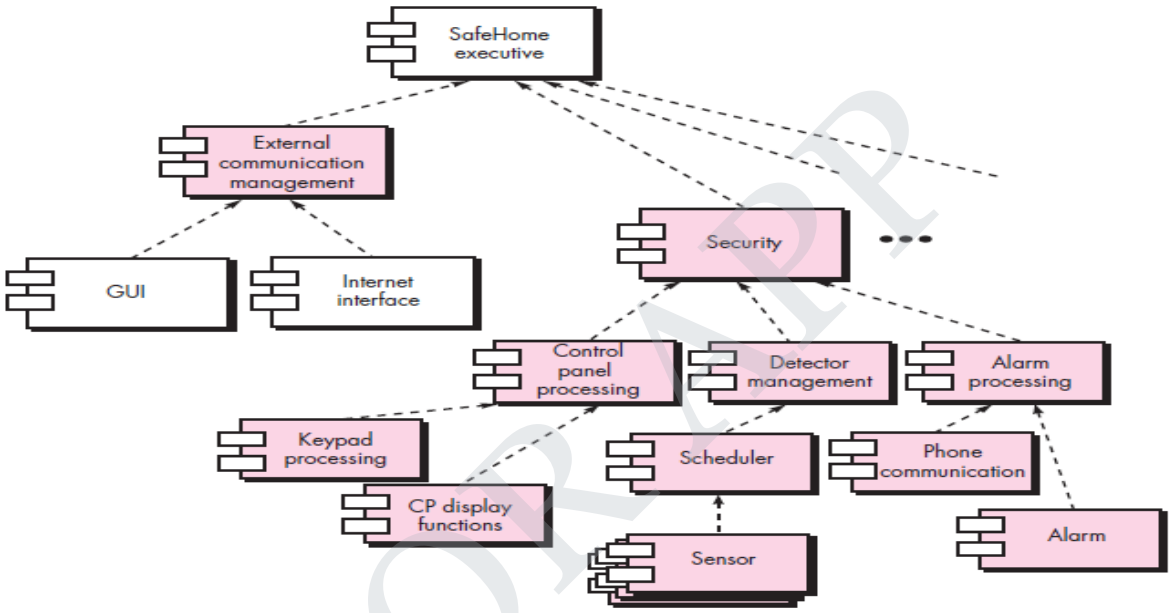



**Fig: Refined Component diagram**

## 3.6 ARCHITECTURAL MAPPING

Structured design provides a convenient transition from a data flow diagram to software Architecture

**Types of information flow**

The 2 different types of information flows:

1. **transaction flow -** a single data item triggers information flow along one of many paths
2. **transform flow**
   - overall data flow is sequential and flows along a small number of straight line paths
   - ***Incoming Flow:*** The paths that transform the external data into an internal form
   - ***Transform Center:*** The incoming data are passed through a transform center and begin to move along paths that lead it out of the software
   - ***Outgoing Flow:*** The paths that move the data out of the software

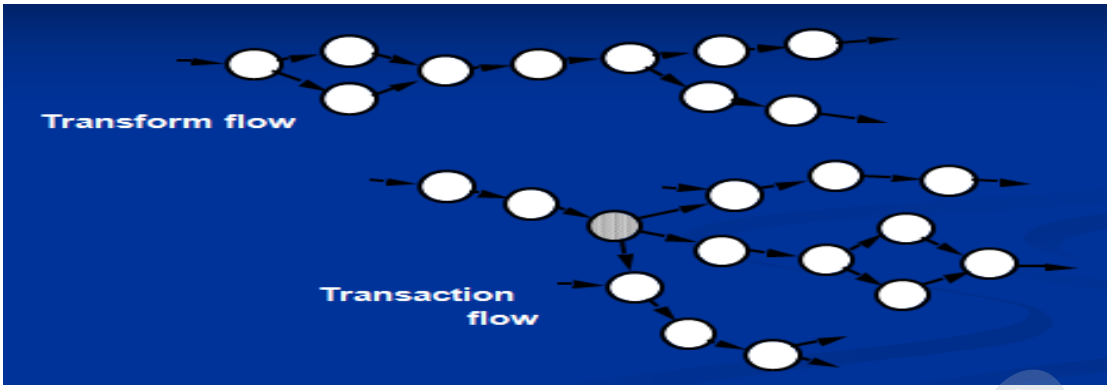



*Fig: Types of Information flow*

### *3.6.1* Transform mapping

- Mapping the DFD with transform flow into an Architectural design is referred as Transform mapping

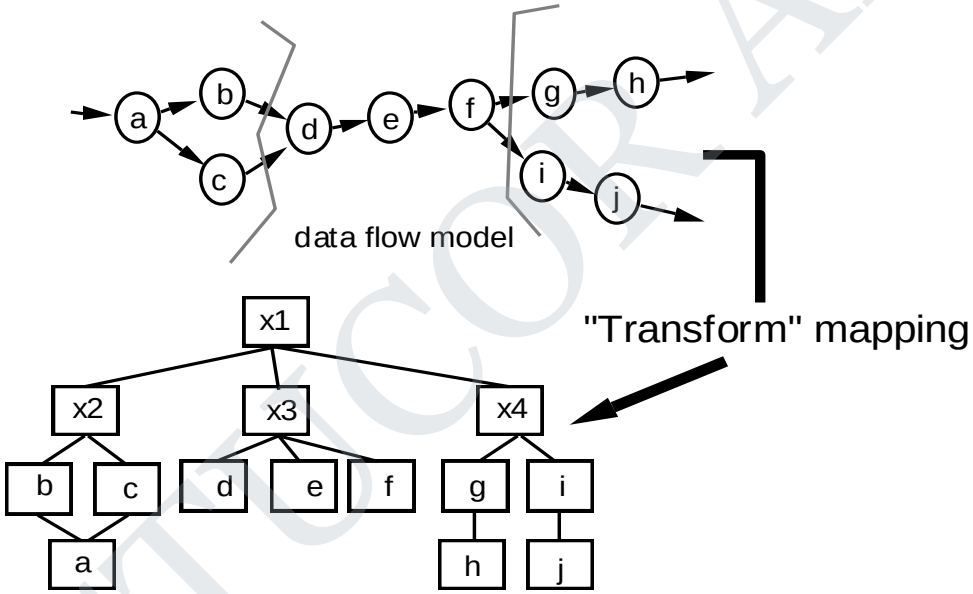



**Fig: Transform mapping**

**STEPS INVOLVED IN ARCHTECTURAL MAPPING:**

- **Step 1. Review the fundamental system model.**
  - Eg: SAFE HOME PROJECT

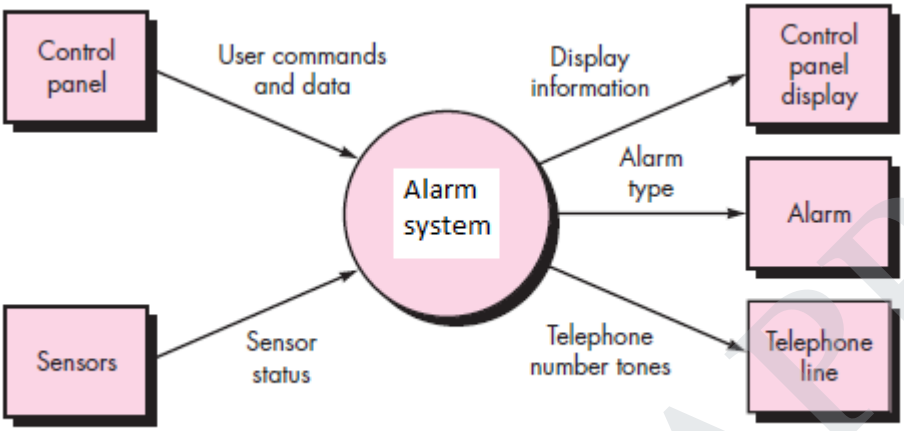



Fig: Context level diagram for Alarm system

- **Step 2. Review and refine data flow diagrams for the software.**

  Eg:

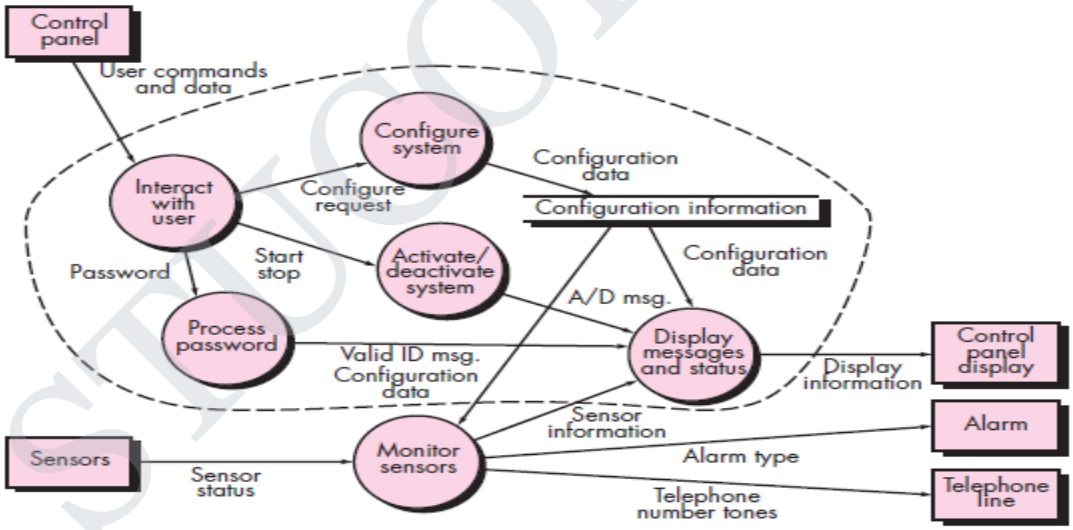
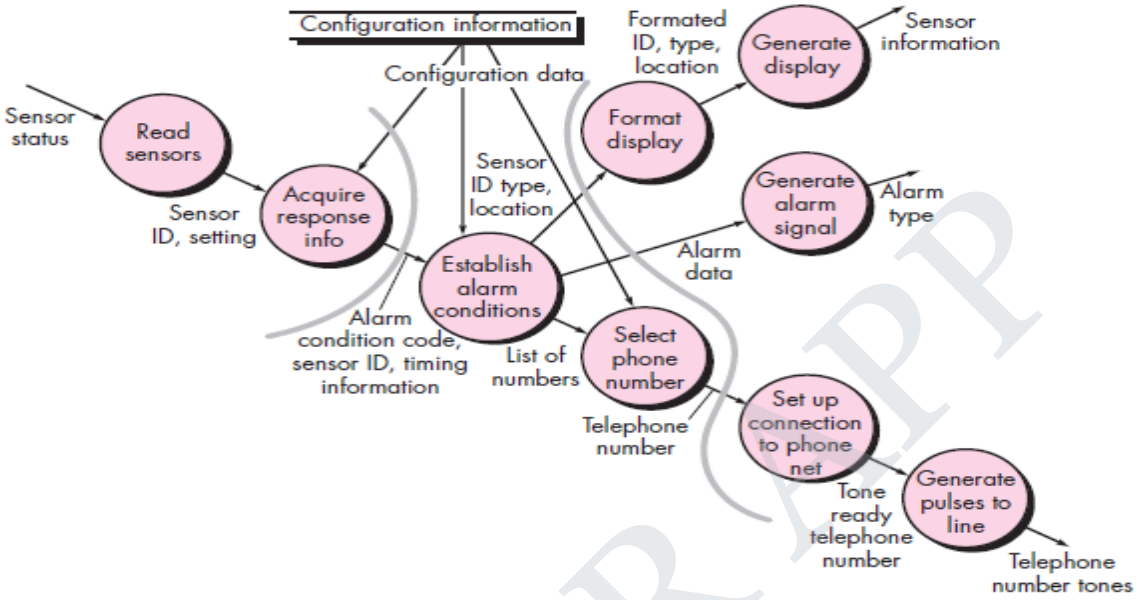



Fig: Level-n DFD

- **Step 3. Determine whether DFD has transform or transaction flow characteristics.**
  - **in general---transform flow**
  - **special case---transaction flow**
- **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries**

- different designers may select slightly differently
- transform center can contain more than one bubble.

Eg:




- **Step 5. Perform "first-level factoring"**
  - Program structure represents a top-down distribution control.
  - factoring results in a program structure(top-level, middle-level, low-level)
  - Number of modules limited to minimum.

Eg:

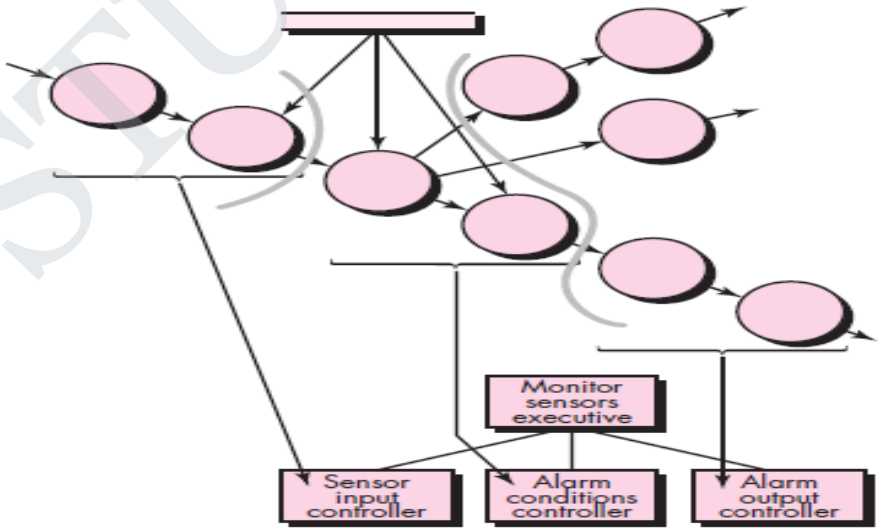



Fig: First level factoring

- **Step 6. Perform "second-level factoring"**
  - Mapping individual transforms (bubbles) to appropriate modules.
  - Factoring accomplished by moving outwards from transform center boundary.

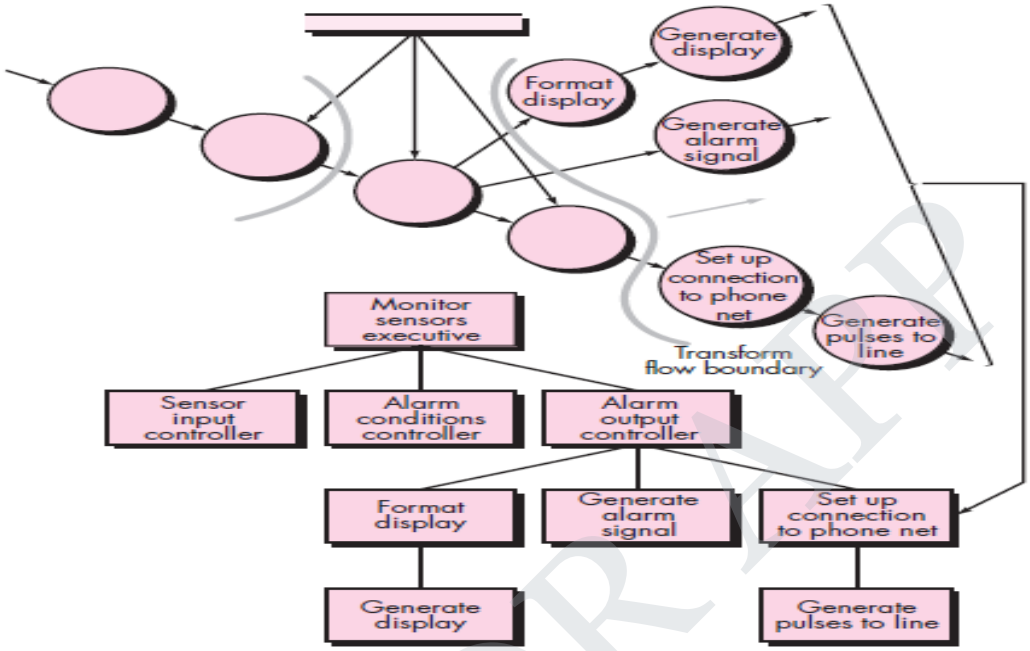
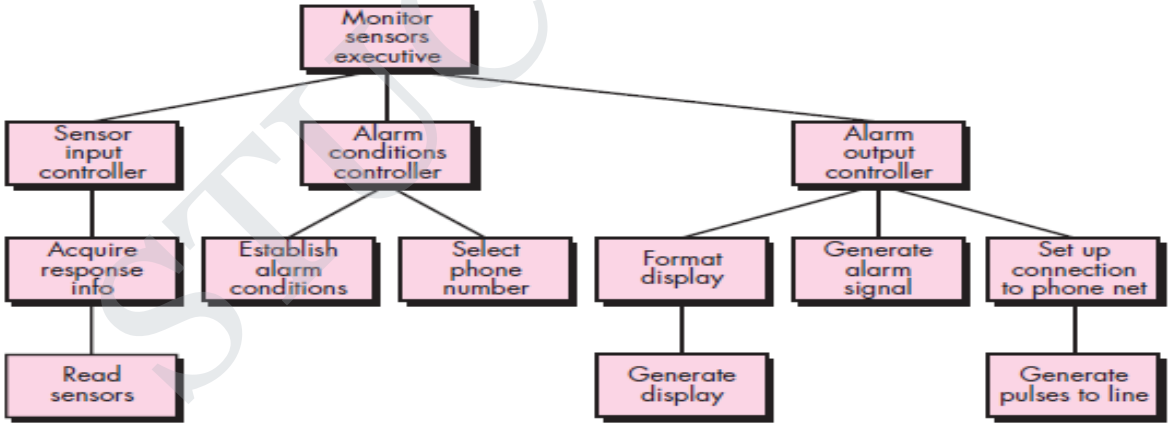



Fig: Second level factoring

Eg: First iteration architecture




- **Step 7. Refine the first iteration program structure using design heuristics** for improved software quality.

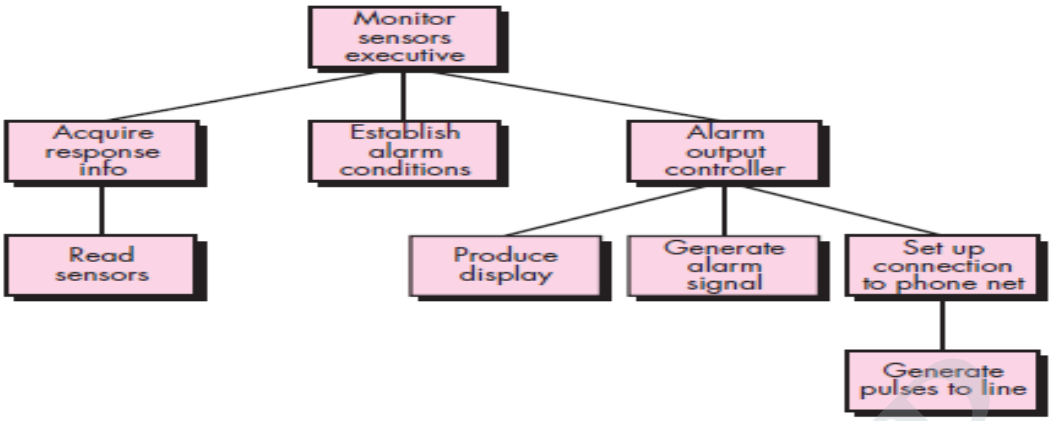



**Fig: final Architecture**

## 3.7 USER INTERFACE ANALYSIS AND DESIGN

### 3.7.1 Introduction

- **Golden Rules**

1. Place the ***user*** in control
   - Define interaction modes in such a way that does not force a user into ***unnecessary or undesired actions***.
   - Allow user interaction to be ***interruptible and undoable***.
   - Streamline interaction as *skill levels* advance and allow the interaction to be ***customized***.
   - Design for ***direct*** interaction with objects that appear on the screen.
2. Reduce the ***user***'s memory load
   - Reduce demand on ***short-term memory***.
   - Establish meaningful ***defaults***.
   - Define ***shortcuts*** that are intuitive.
   - The visual layout of the interface should be based on a ***real world metaphor.***
   - Disclose information in a ***progressive*** fashion
3. Make the interface consistent for the ***user***
   - Allow the user to put the current ***task*** into a meaningful ***context***.
   - Maintain consistency ***across*** a family of applications.
   - If ***past*** interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

### 3.7.2 Interface Design models

- There are Four different models for interface design:
  - **User profile model**
    - Establish the profile of end users of the system
    - considers the syntactic and semantic knowledge of the user
  - **Design model**
    - This model is derived from the analysis model of the requirements
    - This incorporates data, architectural, interface, and procedural representations of the software
  - **Implementation model**
    - This model consists of the look and feel of the interface combined with all supporting information that describe system syntax and semantics
  - **User's mental model**
    - This model is also called the user's system perception
    - This model consists of the image of the system that users carry in their heads

### 3.7.3 INTERFACE DESIGN PROCESS

- User interface development follows a **spiral process**

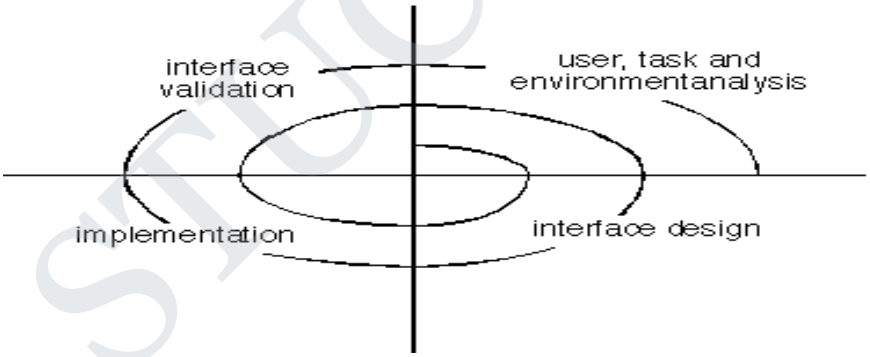



Fig: Interface Design steps

- Steps involved in the interface design process:
  1. Interface analysis
     - This phase focuses on the profile of the users who will interact with the system
     - Skill level, business understanding, and general ideas to the new system are recorded; and different user categories are defined

- Concentrates on users, tasks, content and work environment

2. Interface design
   - The set of interface objects and actions that is needed to perform all defined tasks in a manner that meets every usability goal defined for the system are defined in this phase.
3. Interface construction
   - construction begins with a prototype that enables usage scenarios to be evaluated
   - Continues with development tools to complete the construction
4. Interface validation
   - This focus on the ability of the interface to implement every user task, to accommodate all task variations, and to achieve all user requirements
   - The degree to which the interface is easy to use and easy to learn

### 3.7.4 INTERFACE ANALYSIS

- Interface analysis means understanding

  (1) The *users* who will interact with the system through the interface;

  (2) The *tasks* that end-users must perform to do their work,

  (3) The *content* that is presented as part of the interface

  (4) The *environment* in which these tasks will be conducted.

#### 3.7.4.1 User Analysis

- The analyst uses both the user's mental model and the design model to understand the users and how they use the system
- Information are collected from
  - User interviews with the end users
  - Sales input from the sales people who interact with customers and users on a regular basis
  - Marketing input based on a market analysis to understand how different population segments might use the software
  - Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features are easy to use etc.
- sample questionnaire used for user analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- Are users experts in the subject matter that is addressed by the system?

### 3.7.4.2 Task Analysis and Modeling

- Task analysis helps to know and understand
  - The work the user performs in specific cases
  - The tasks and subtasks that are performed by the user
  - The specific problem domain objects that the user manipulates as work is performed
  - The sequence of work tasks
  - The hierarchy of tasks

**Use cases**

- Use cases are used to define the interaction between the components.
- Use case show how an end user performs some specific work-related task
- This helps to extract tasks, objects, and overall workflow of the interaction
- This helps the software engineer to identify additional helpful features

**Task elaboration**

- Task elaboration refines interactive tasks
- This gives the step wise elaboration of the function
- Eg: withdraw from an ATM
  - Insert the card
  - enter pin
  - select withdraw option
  - enter amount
  - dispense cash etc

**Object elaboration**

- Object elaboration identifies interface objects
- Attributes of each class are defined, and an evaluation of the actions applied to each object provides a list of operations.

- Eg: class Withdraw can include the attributes like account no, name, balance and functions like withdraw() and update() etc

**Workflow analysis**

- Workflow analysis defines how a work process is completed when several people are involved
- **For work flow analysis swimlane diagram is used**
- Eg:

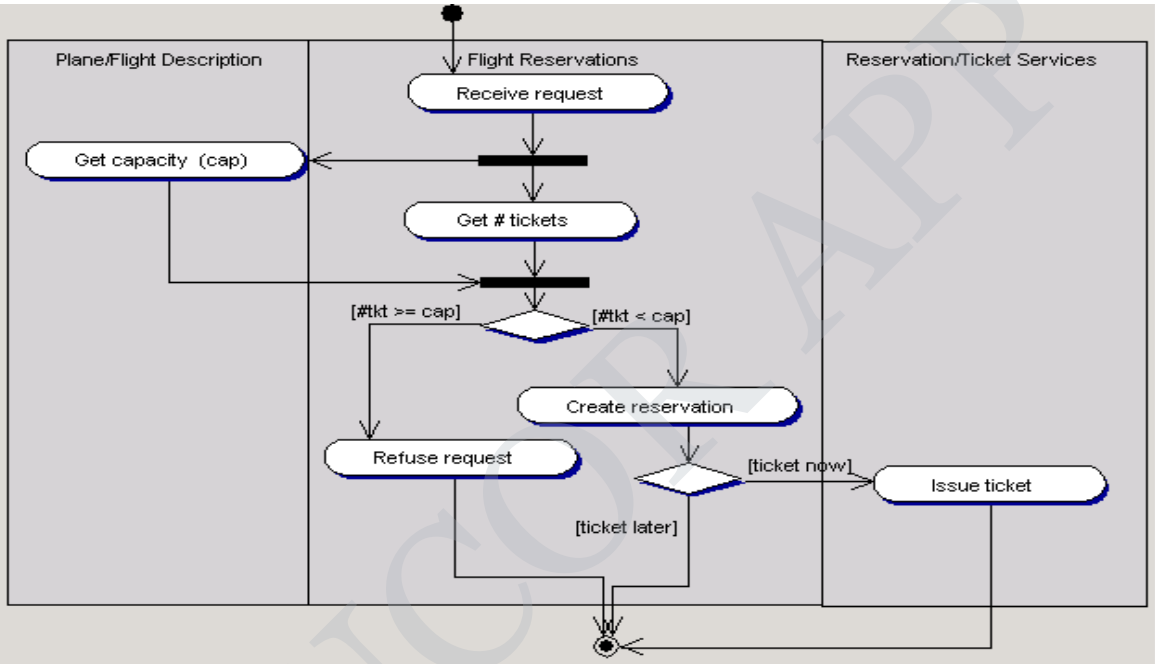



Fig: Swimlane diagram for ticket reservation

**Hierarchical representation**

- Once workflow has been established, a task hierarchy can be defined for each user type.
- The hierarchy is derived by a stepwise elaboration of each task identified for the user

**3.7.4.3 Analysis of Display Content**

- This phase is about analyzing how to present the variety of contents
- The display content may range from character-based reports (e.g Spreadsheet), to graphical displays(e.g : histogram,3D model etc), to multimedia information(e.g Audio or video)
- Display content may be
    - Generated by components in other parts of the application
    - Acquired from data stored in a database that is accessible from the application

- Transmitted from systems external to the application in question

- Sample questionnaire for content analysis
  - Are different types of data assigned to consistent geographic locations on the screen?
  - Can the user customize the screen location for content?
  - Will graphical output be scaled to fit within the bounds of the display device that is used?
  - How will color to be used to enhance understanding?
  - How will error messages and warning be presented to the user?

**3.7.4.4 Work Environment Analysis**

- Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use
- Factors to consider include
  - Type of lighting
  - Display size and height
  - Keyboard size, height and ease of use
  - Mouse type and ease of use
  - Surrounding noise
  - Space limitations for computer and/or user
  - Weather or other atmospheric conditions
  - Temperature or pressure restrictions
  - Time restrictions (when, how fast, and for how long)

**3.7.5 INTERFACE DESIGN**

- User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding step
- Steps involved in user interface design
  1. Define user interface objects and actions from the information collected in the analysis phase
  2. Define events that will cause the state of the user interface to change and model the behavior
  3. Depict each interface state as it will actually look to the end user

4. Indicate how the user interprets the state of the system from information provided through the interface

**Applying Interface Design steps**

- Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement
- Interface objects are categorized into 3 types: source, target, and application
  - A **source object** is dragged and dropped into a **target** object such as to create a hardcopy of a report
  - An **application object** represents application-specific data that are not directly manipulated as part of screen interaction such as a list
- After identifying objects and their actions, an interface designer performs screen layout which involves
  - Graphical design and placement of icons
  - Definition of descriptive screen text
  - Specification and titling for windows
  - Definition of major and minor menu items
  - Specification of a real-world metaphor to follow

**User Interface Design Patterns**

- A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.
- Eg:
  - **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar.

**Design Issues**

- Four common design issues in any user interface
  - **System response time** (both length and variability)
    - **Length** is the amount of time taken by the system to respond.
    - **Variability** is the deviation from average response time
  - **User help facilities**
    - Help facilities gives information about when is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited

- Eg: user manuals or online help facilities

- **Error information handling**
  - Error messages and warnings should describe the problem
  - It should provide constructive advice for recovering from the error
  - It should indicate negative consequences of the error etc
  - This messages helps to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur
- **Menu and command labeling**
  - Menu and command labeling should be consistent, easy to learn
  - Questions for menu labeling
    1. Will every menu option have a corresponding command?
    2. What can be done if a command is forgotten?
    3. Can commands be customized or abbreviated by the user?
- **Application accessibility**
  - Software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs.
- **Internationalization**
  - The challenge for interface designers is to create "globalized" software.
  - That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software.
  - Localization features enable the interface to be customized for a specific market.

## 3.8 COMPONENT LEVEL DESIGN

- A component is a modular building block for computer software
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code

### 3.8.1 DESIGNING CLASS-BASED COMPONENTS

**Component-level Design Principles**

- **Open-closed principle**
  - A module or component should be open for extension but closed for modification

  - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component

- **Liskov substitution principle**
  - The Subclasses should be substitutable for their base classes
  - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- **Dependency inversion principle**
  - This principle depend on abstractions , do not depend on concretions
  - The more a component depends on other concrete components, the more difficult it will be to extend
- **Interface segregation principle**
  - Many client-specific interfaces are better than one general purpose interface
  - For a server class, specialized interfaces should be created to serve major categories of clients
  - Only those operations that are relevant to a particular category of clients should be specified in the interface

**Component Packaging Principles**

- **Release reuse equivalency principle**
  - The granularity of reuse is the granularity of release
  - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- **Common closure principle**
  - Classes that change together belong together
  - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- **Common reuse principle**
  - Classes that aren't reused together should not be grouped together

- Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

### 3.8.2 Component-Level Design Guidelines

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - architectural component names must be obtained from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
  - infrastructure component names must that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
  - Dependencies should be modelled from left to right and inheritance from top (base class) to bottom (derived classes)
- Interfaces
  - Interfaces provide important information about communication and collaboration
  - lollipop representation of an interface should be used in UML approach
  - For consistency, interfaces should flow from the left-hand side of the component box;
  - only those interfaces that are relevant to the component under consideration should be shown

### 3.8.3 Cohesion

- Cohesion is the "single-mindedness' of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - **Functional**
    - Exhibited primarily by operations

- This level of cohesion occurs when a component performs a targeted computation and then returns a result.

- **Layer**
  - Exhibited by packages, components, and classes,

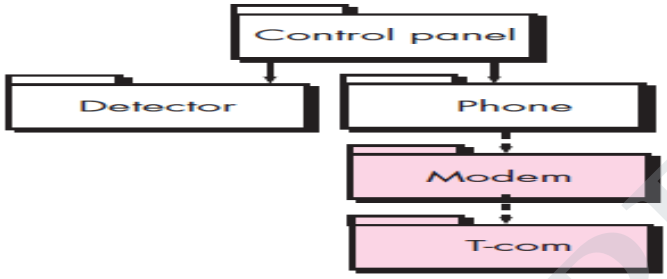



Fig: Layer Cohesion

  - This type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
- **Communicational**
  - All operations that access the same data are defined within one class.

### 3.8.4 Coupling

- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- Different types of coupling are as follows:
  - **Content coupling**
    - Occurs when one component surreptitiously modifies data that is internal to another component .
    - This violates information hiding.
  - **Data coupling**
    - Data coupling occurs when operations pass long strings of data arguments.
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
    - Testing and maintenance are more difficult
  - **Common coupling**
    - This coupling occurs when a number of components make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

- **Stamp coupling**
  - A whole data structure or class instantiation is passed as a parameter to an operation
- **Control coupling**
  - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
  - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
- **Routine call coupling**
  - Occurs when one operation invokes another.
- **Type use coupling**
  - Occurs when a Component A uses a data type defined in component B
  - If the type definition changes, every component that declares a variable of that data type must also change
- **Inclusion or import coupling**
  - Occurs when component **A** imports or includes a package or the content of component **B**.
- **External coupling**
  - Occurs when a component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

## 3.9 DESIGNING CONVENTIONAL COMPONENTS

- Conventional design constructs emphasize the maintainability of a functional domain
- The constructs include Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations used for designing these constructs
  1. Graphical design notation
     - Sequence, if-then-else, selection, repetition
  2. Tabular design notation
  3. Program design language

- This is similar to a programming language but it uses narrative text embedded within the program statements

**Graphical design notation**

- Graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that shows the procedural detail.
- The **activity diagram** allows to represent sequence, condition, and repetition
- A **flowchart** like an activity diagram which is quite simple pictorially.
- Notations used in flowchart:
  - A box is used to indicate a processing step.
  - A diamond represents a logical condition
  - arrows show the flow of control.

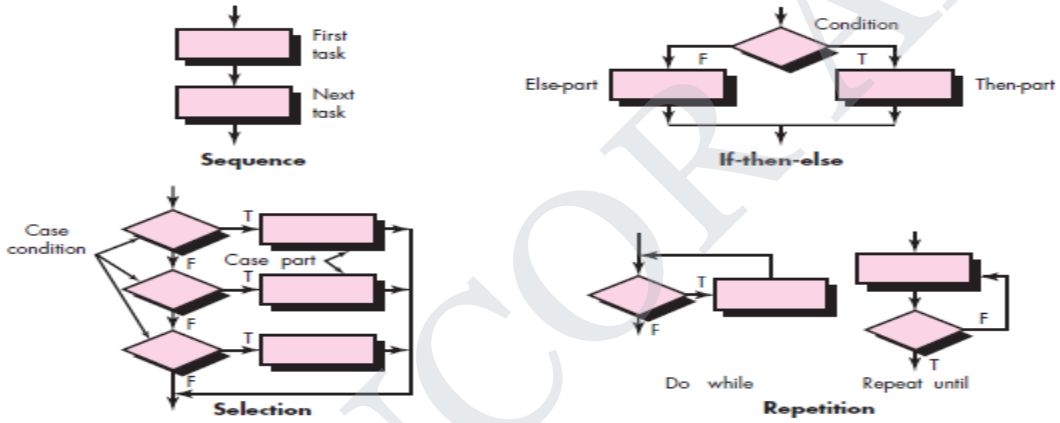



Fig: Flowchart constructs

**Tabular Design Notation**

- Decision tables provide a notation that translates actions and conditions into a tabular form.
- Tabular notation is difficult to misinterpret.
- The table is divided into four sections.
  - The upper left-hand quadrant contains a **list of all conditions**.
  - The lower left-hand quadrant contains a **list of all actions** that are possible based on combinations of conditions.
  - The right-hand quadrants form a matrix that indicates **condition combinations and the corresponding actions** that will occur for a specific combination.
  - Therefore, each column of the matrix may be interpreted as a **processing rule**.

Eg:

| | | Rules | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Conditions | Credit limit exceeded | Y | Y | Y | Y | N | N | N | N |
| | Customer with good payment history | Y | Y | N | N | Y | Y | N | N |
| | Purchase above $200 | Y | N | Y | N | Y | N | Y | N |
| Action | Allow credit | | | | | X | X | X | X |
| | Refuse credit | X | | X | X | | | | |
| | Refer to manager | | X | | | | | | |

Key:
Y = Yes, condition true
N = No, condition not true

- **Steps involved in developing a decision table:**

1) List all actions associated with each module
2) List all conditions during execution of the procedure
3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions
4) Define rules by indicating what actions occurs for a set of conditions

**Program Design Language**

- Program design language (PDL), also called structured English or pseudocode,
- PDL combines the logical structure of a programming language with the free-form natural language.
- Automated tools can be used to enhance the application.
- A PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and I/O constructs
- PDL can be extended to include keywords for multitasking and concurrent processing, interrupt handling, interprocess synchronization etc.
- Sample PDL:

```
IF credit level exceeded
        THEN (credit level exceeded)
                IF customer has bad payment history
                        THEN refuse credit
                ELSE ( customer has good payment history)
                IF purchase is above $200
                        THEN refuse credit
                ELSE (purchase is below $200)
                Refer to manager
        ELSE (credit level not exceeded) allow credit
```

# UNIT-IV

# TESTING AND IMPLEMENTATION

## 4.1 INTRODUCTION TO TESTING

- Testing is the process of finding the errors in the software before delivering to the end user.
- The main objective of testing is to uncover errors with a minimum of effort and time

**Characteristics of a Good Test:**

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test is neither too simple nor too complex
- A good test should be "best of breed"

**Testability:**

- Software testability is simply how easily a computer program can be tested.

**Characteristics of Testable Software**

- Operable
  - The better it works, the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently
- Simple
  - The less there is to test, the more quickly we can test it.
  - The program should exhibit
    - Functional simplicity
    - Structural simplicity
    - Code simplicity
- Stable
  - If the changes are less, then the disruptions to testing are also less.

- Changes to the software during testing are infrequent and do not invalidate existing tests

- Understandable
  - The more information exists, the smarter to test.
  - The architectural design is well understood; documentation is available and organized

## 4.2 INTERNAL AND EXTERNAL VIEWS OF TESTING

**Black-box testing (External view of testing)**

- Knowing the specified function that has to be performed, if we test to check whether the function is fully operational and error free then it is termed as Black box testing.
- Black box testing will not test the internal logical structure of the software

**White-box testing (Internal view of testing)**

- White box testing checks whether the internal operations perform according to the specification
- This is also referred as glass box testing
- This involves tests that concentrate on close examination of procedural detail
- Logical paths are also tested.
- This uses the control structure of component design to derive the test cases
- The test cases
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

## 4.3 WHITE-BOX TESTING

### 4.3.1 BASIS PATH TESTING

- Basis path testing is a one of the White-box testing technique.
- This helps to derive a logical complexity measure of a procedural design
- Test cases derived to exercise the basis set should execute every statement in the program at least one time during testing

**Flow Graph Notation**

- Flow graph *is* a pictorial representation of control flow.
- This is also referred as a *program graph*
- A circle in a graph represents a node, which stands for procedural statements
- A node containing the condition is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)

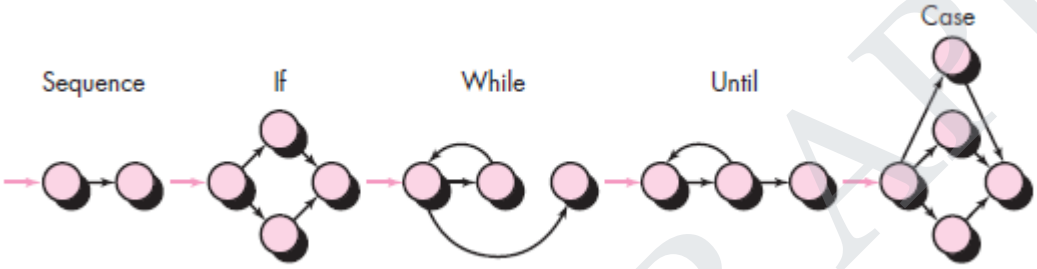
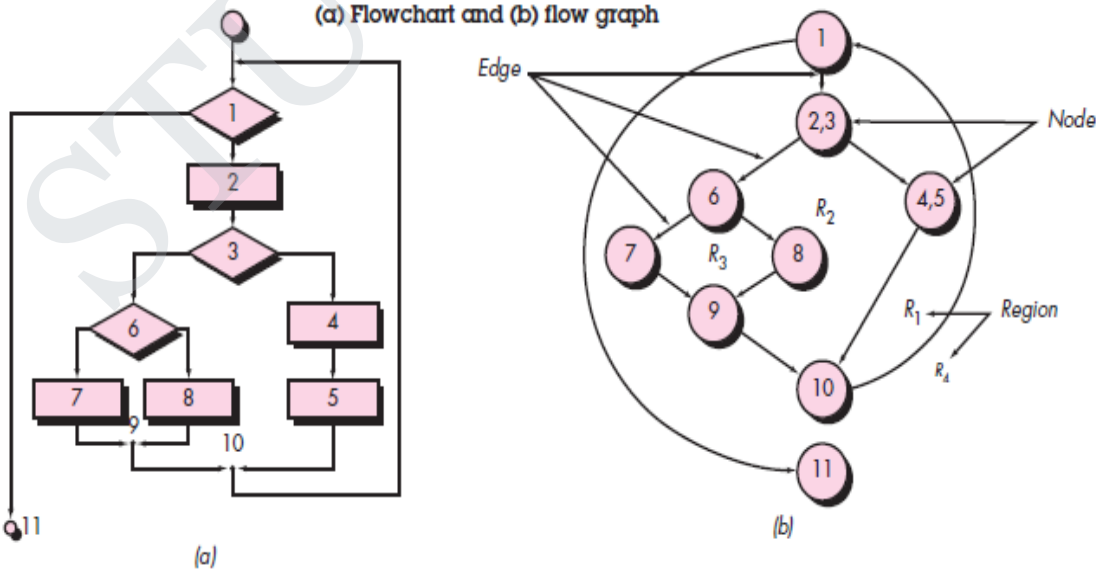



Fig: Flow Graph Notation

- An edge, or a link, is a an arrow representing the control flow in a specific direction
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too
- Eg:



**Independent Program Paths**

- An independent path is any path from the start node until the end node that introduces at least one new set of processing statements or a new condition.
- An independent path must move along at least one edge that has not been traversed before by a previous path
- Basis set for the above flow graph :
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

**Cyclomatic Complexity**

- Cyclomatic complexity provides a quantitative measure of the logical complexity of a program
- It defines the number of independent paths in the basis set
- It provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- This can be computed three ways
  - The number of regions
  - V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G
- Cyclomatic complexity for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

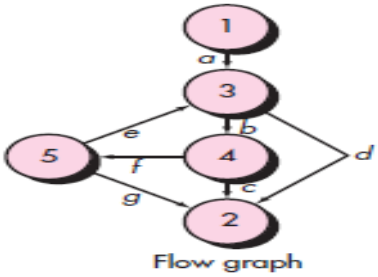
**Deriving the Basis Set and Test Cases**

- The following steps can be applied to derive the basis set:
  1) Using the design or code as a foundation, draw a corresponding flow graph
  2) Determine the cyclomatic complexity of the resultant flow graph
  3) Determine a basis set of linearly independent paths

4) Prepare test cases that will force execution of each path in the basis set

**Graph Matrices**

- Graph matrix is a data structure used for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections between nodes.
- Eg:



Flow graph

| Node \ Connected to node | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | | a | | |
| 2 | | | | | |
| 3 | | d | | b | |
| 4 | | c | | | f |
| 5 | | g | e | | |

Graph matrix

- A *link weight* can be added to each matrix entry to provide additional information about control flow.
- If the link weight is 1 (a connection exists) or 0 (a connection does not exist).

Eg:

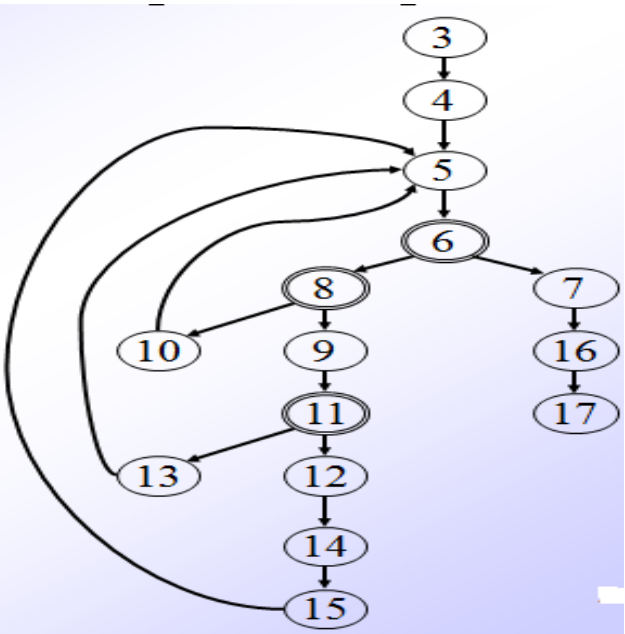
```
 1  int functionY(void)
 2  {
 3      int x = 0;
 4      int y = 19;

 5  A:  x++;
 6      if (x > 999)
 7          goto D;
 8      if (x % 11 == 0)
 9          goto B;
10      else goto A;

11  B:  if (x % y == 0)
12          goto C;
13      else goto A;

14  C:  printf("%d\n", x);
15      goto A;

16  D:  printf("End of list\n");
17      return 0;
18  }
```



## 4.4 CONTROL STRUCTURE TESTING

### Condition Testing

- *Condition testing* is a test-case design method used to check the logical conditions in the module.
- This tests each condition in the program to ensure that its error- free.
- A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.
- A relational expression takes the form

  *E*1 <relational-operator> *E*2

  where *E*1 and *E*2 are arithmetic expressions and

  <relational-operator> is one of the following: <,>,= , < = etc
- A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. Boolean operators allowed in a compound condition include OR (||), AND (&), and NOT (¬).
- A condition without relational expressions is referred to as a Boolean expression.
- If a condition is incorrect, then at least one component of the condition is incorrect.
- Types of errors in a condition include Boolean operator errors, Boolean variable errors, Boolean Parenthesis errors, relational operator errors, and arithmetic expression errors.

### Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.
- Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.
- For a statement with *S* as its statement number,

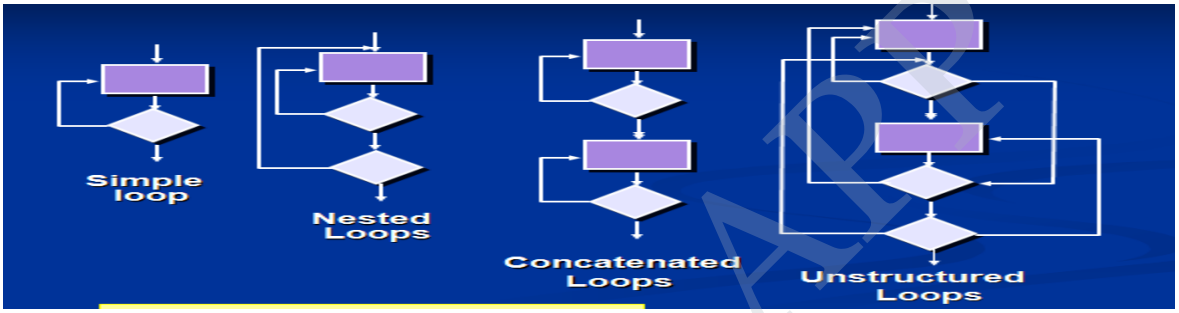
  DEF(*S*) = {*X* | statement *S* contains a definition of *X*}

  USE(*S*) = {*X* | statement *S* contains a use of *X*}
- If statement *S* is an *if* or *loop statement*, its DEF set is empty and its USE set is based on the condition of statement *S*.
- A *definition-use (DU) chain* of variable *X* is of the form [*X, S, S'*], where *S* and *S'* are statement numbers, *X* is in DEF(*S*) and USE(*S'*), and the definition of *X* in statement *S* is live at statement *S'*.

- Every DU chain must be covered at least once. This strategy as the DU testing strategy.

**Loop Testing**

- Loop testing is a white-box testing technique that is used to test the validity of loop constructs.
- Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops
- Testing occurs by varying the loop boundary values




**Testing of Simple Loops**

- Testing of Simple Loops where *n* is the maximum number of allowable passes through the loop.
    1. Skip the loop entirely
    2. Only one pass through the loop
    3. Two passes through the loop
    4. m passes through the loop, where m < n
    5. n –1, n, n + 1 passes through the loop

**Testing of Nested Loops**

1) Start at the innermost loop; set all other loops to minimum values
2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values
3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values
4) Continue until all loops have been tested

**Testing of Concatenated Loops**

- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

**Testing of Unstructured Loops**

- Redesign the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

## 4.5 BLACK-BOX TESTING

- Black-box testing is also called as behavioral testing
- This testing focuses on the functional requirements of the software and the information domain of the software
- Black-box testing attempts to find errors in the following categories:

  (1) Incorrect or missing functions,

  (2) Interface errors,

  (3) Errors in data structures or external database access,

  (4) Behavior or performance errors, and

  (5) Initialization and termination errors.
- A set of input conditions are identified to check all the functional requirements for a program

### 4.5.1 Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects and their relationships.
- Then a series of tests are defined to verify whether all the objects have the expected relationship with each other.
- Eg:

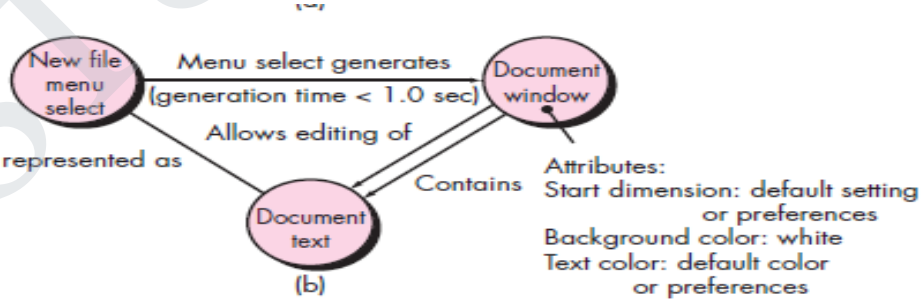



Fig: Graph notation

- Nodes represent the objects and links represent the relationships between objects,
- Node weights describe the properties of the node and link weights describes the characteristic of the link.

- A directed link (represented by an arrow) indicates that a relationship moves in only one direction. A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.
- Behavioral testing methods that can make use of graphs:

1. **Transaction flow modeling.**

   The nodes represent steps in some transaction and the links represent the logical connection between steps. The data flow diagram can be used to assist in creating graphs of this type.

2. **Finite state modeling.**

   The nodes represent different user-observable states of the software and the links represent the transitions that occur to move from state to state. The state diagram can be used to assist in creating graphs of this type.

3. **Data flow modeling.**

   The nodes are data objects, and the links are the transformations that occur to translate one data object into another.

4. **Timing modeling.**

   The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times.

### 4.5.2 Equivalence Partitioning

- Equivalence partitioning is a black-box testing method which divides the input domain into classes of data and then derives the test cases from it.
- An ideal test case single-handedly uncovers a complete class of errors and reduce the total number of test cases that has to be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An **equivalence class** represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

**Guidelines for Defining Equivalence Classes**

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined

- Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}

- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

### 4.5.3 Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than at the center so boundary value analysis is developed
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

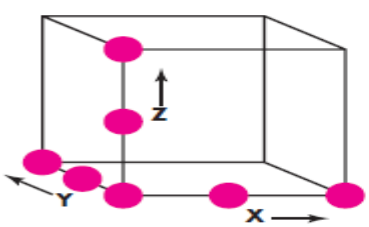
**Guidelines for Boundary Value Analysis**

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*
2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
3. Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
4. If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

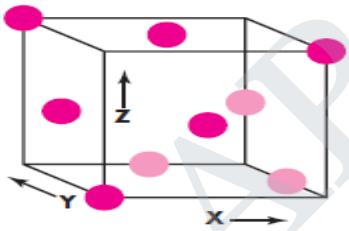
### 4.5.4 Orthogonal Array Testing

- Orthogonal array testing can be applied for the problems having a small input domain but too large to accommodate exhaustive testing.
- The orthogonal array testing method is useful in finding **region faults.**

- Region fault is an error category associated with faulty logic within a software component.
- Eg:
- To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X*, *Y*, and *Z*.
- Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases



One input item at a time



L9 orthogonal array

- To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function.
  Each takes on three discrete values.
  For example, P1 takes on values:
  P1= 1, send it now
  P1=2, send it one hour later
  P1= 3, send it after midnight
- If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).
- Eg:

| Test case | Test parameters | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 |
| 3 | 1 | 3 | 3 | 3 |
| 4 | 2 | 1 | 2 | 3 |
| 5 | 2 | 2 | 3 | 1 |
| 6 | 2 | 3 | 1 | 2 |
| 7 | 3 | 1 | 3 | 2 |
| 8 | 3 | 2 | 1 | 3 |
| 9 | 3 | 3 | 2 | 1 |

## 4.6 SOFTWARE TESTING STRATEGIES

- Software testing strategies integrates the design of test cases into the different phases of software development
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation

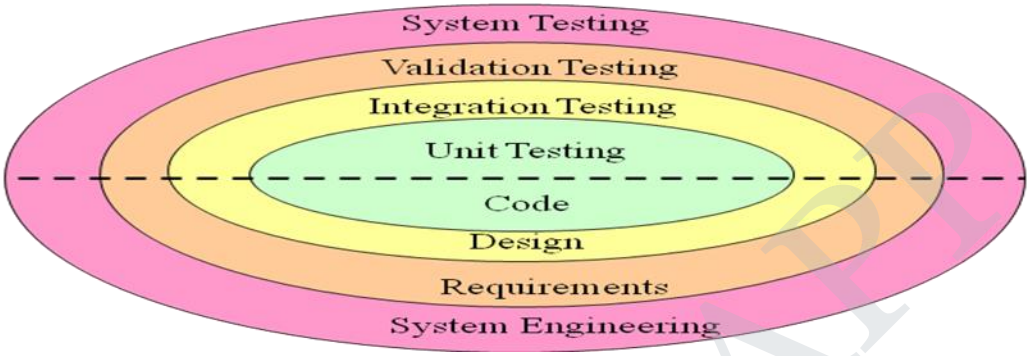



**Fig: Testing Strategy**

### Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements

### Levels of Testing

- **Unit testing**
  - Unit testing concentrates on each component or module of the software
  - This exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- **Integration testing**
  - This focuses on inputs and outputs, and how well the components fit together and work together

- **Validation testing**
  - Requirements are validated against the constructed software
  - Validation testing provides final assurance that the software meets all functional, behavioral, and performance requirements
- **System testing**
  - The software and other system elements are tested as a whole
  - This verifies that all system elements communicate properly and that overall system function and performance is achieved

### 4.6.1 UNIT TESTING

- Unit testing focuses testing each individual module separately.
- This concentrates on the internal processing logic and data structures
- Unit testing is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Unit testing concentrates on critical modules and those with high cyclomatic complexity when testing resources are limited

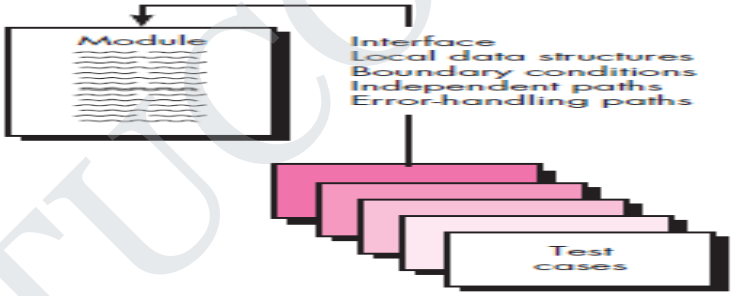



Fig: Unit test

**Unit test considerations**

- Module interface
  - This is tested to ensure that information flows properly into and out of the module
- Local data structures
  - This ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - This ensures that the module operates properly at boundary values

- Independent paths
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific error conditions

**Common Errors in Execution Paths**

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)

**Unit Test Procedures**

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Stubs serve to replace modules that are subordinate to the component to be tested
  - A stub is a "dummy subprogram" that uses the subordinate module's interface, does minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs both represent overhead
  - Both must be written but don't constitute part of the installed software product

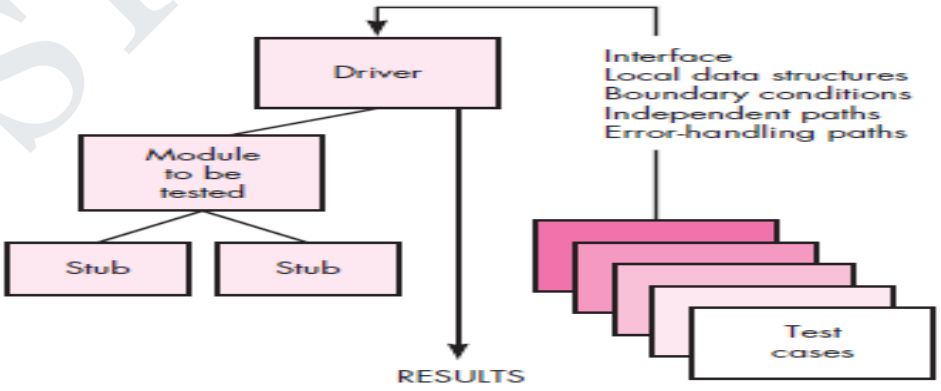



Fig: Unit Test Environment

### 4.6.2 INTEGRATION TESTING

- Integration testing is defined as a systematic technique for constructing the software architecture and also conduct tests to uncover errors associated with interfaces
- The Objective of this testing is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches used
    1. Non-incremental Integration Testing or Big Bang Approach
    2. Incremental Integration Testing

**1. Big Bang Approach**

- In this approach all the components are combined in advance and the entire program is tested as a whole
- This is not appropriate and results in Chaos.
- Many seemingly-unrelated errors are encountered in this testing
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

**2. Incremental Integration Testing**

- Three kinds
    - Top-down integration
    - Bottom-up integration
    - Sandwich integration
- In this approach the program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

**Top-down Integration**

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
- Depth First: All modules on a major control path are integrated
- Breadth First: All modules directly subordinate at each level are integrated

Advantages

- This approach verifies major control or decision points early in the test process

Disadvantages

- Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
- Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Steps followed in top-down integration:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

- Eg:

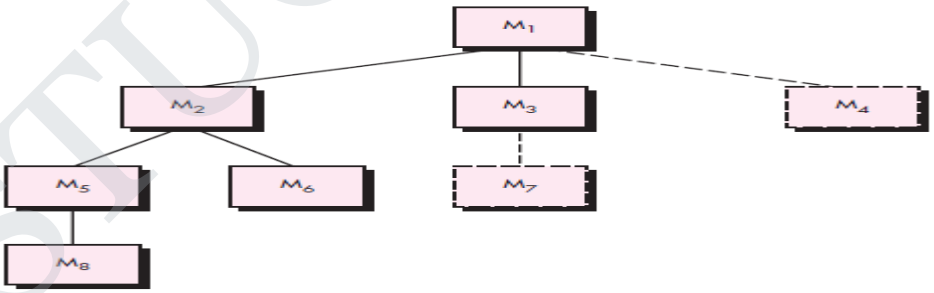



Fig: Top-down integration

- *Depth-first integration* integrates the left side components M1, M2 , M5 first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built.
- *In Breadth-first integration the c*omponents M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on.

**Bottom-up Integration**

- Integration and testing starts with the most atomic modules in the control hierarchy

Advantages

- This approach verifies low-level data processing early in the testing process
- Need for stubs is eliminated

Disadvantages

- Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
- Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Steps for bottom up integration:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Eg:

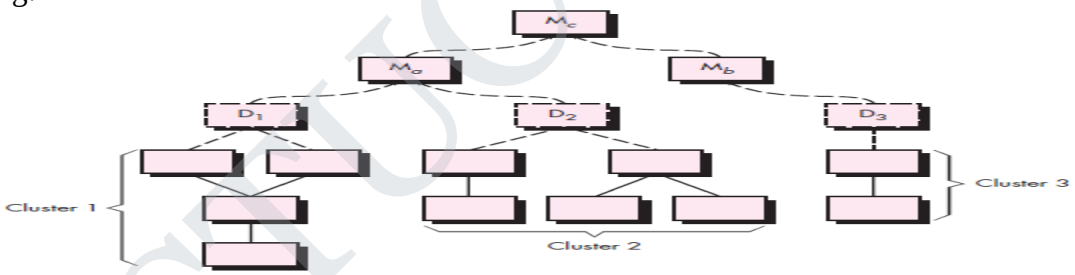



Fig: Bottom-up integration

**Sandwich Integration:**

- This consists of a combination of both top-down and bottom-up integration
- This occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group

- When integration for a certain functional group is complete, integration and testing moves onto the next group

**Regression Testing**

- Any addition or change to the software may cause problems with the functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
- This helps to ensure that changes have not propagated unexpected errors in the program
- This testing can be done manually or through automated tools
- Regression test suite contains three different classes of test cases
  - A sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

**Smoke Testing**

- Smoke testing is designed as a pacing mechanism for time-critical projects
- This testing allows the software team to assess its project on a frequent basis
- This includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

**Benefits of Smoke Testing**

- Integration risk is minimized
- The quality of the end-product is improved
- Error diagnosis and correction are simplified
- Progress is easier to assess

### 4.6.3 VALIDATION TESTING

- Validation testing follows integration testing
- This testing focuses on user-visible actions and user-recognizable output from the system
- Validation testing is designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- After each validation test
  - The function conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created

**Configuration review**

- A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

**Alpha testing**

- Alpha testing is conducted at the developer's site by end users
- Software is used in a natural setting with developers watching intently
- Testing is conducted in a controlled environment

**Beta testing**

- This testing is conducted at end-user sites
- Developer is generally not present
- It serves as a live application of the software in an environment that cannot be controlled by the developer
- The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, engineers make modifications in the software and prepare for release of the software product to the entire customer base

### 4.6.4 SYSTEM TESTING

**Different Types of system testing**

**Recovery testing**

- This is to tests for recovery from system faults
- Recover testing forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness

**Security testing**

- This verifies whether the protection mechanisms built in the system will protect it from improper access
- During security testing, the tester plays the role of the hacker, attempts to acquire passwords, may purposely cause system errors, may browse through insecure data, hoping to find the key to system entry.

**Stress testing**

- Stress tests are designed to confront programs with abnormal situations.
- This executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

**Performance testing**

- Tests the run-time performance of software within the context of an integrated system
- Performance testing is often combined with stress testing and usually requires both hardware and software instrumentation
- This testing can uncover situations that lead to degradation and possible system failure

**Deployment Testing**

- Deployment testing is also called as configuration testing
- Software should be able to execute on a variety of platforms and under more than one operating system environment. Deployment testing exercises the software in each environment in which it is to operate.
- Deployment testing examines all installation procedures and specialized installation software that will be used by customers, and all documentation that will be used to introduce the software to end users.

## 4.7 DEBUGGING

### Debugging Process

- Debugging is the process of removing the defects.
- It occurs as a consequence of successful testing
- The debugging process begins with the execution of a test case, assess the result and then the difference between expected and actual performance is encountered
- The debugging process matches the symptom with the cause and leads to error correction
- The debugging process will usually have one of two outcomes:

  (1) The cause will be found and corrected

  (2) The cause will not be found. In this case, the debugger may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

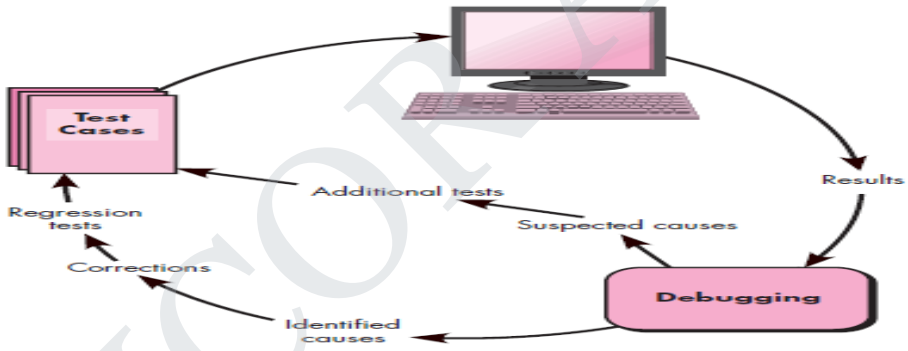



Fig: Debugging Process

### Difficulties in Debugging

- The symptom may disappear when another error is corrected
- The symptom may be caused by human error that cannot be easily traced
- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

### Debugging Strategies

- The main objective of debugging is to find the cause of the error and correct it.
- Bugs can be found by systematic evaluation and intuition
- There are three main debugging strategies

- Brute force
- Backtracking
- Cause elimination

**Brute Force**

- Brute force is the most commonly used and least efficient method. This is used when all else fails
- This approach involves the use of memory dumps, run-time traces, and output statements
- This leads to wasted effort and time

**Backtracking**

- This can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

**Cause Elimination**

- This approach involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is defined, and the hypothesis is proved or disproved using the data collected.
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

**Correcting the Error**

- Three Questions to ask Before Correcting the Error
  - Is the cause of the bug reproduced in another part of the program?

- Similar errors may be occurring in other parts of the program
- What next bug might be introduced by the fix that I'm about to make?
- What could we have done to prevent this bug in the first place?

## 4.8 SOFTWARE MAINTANENCE

**Software maintenance:**

- Software maintenance is the process of modifying a software system after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.
- Maintenance is concerned with correcting errors or design, adapting the software to changing requirements, changing environments etc.
- Types of Software Maintenance:
  1. Corrective maintenance: - correcting the software faults.
  2. Adaptive maintenance: - Maintenance for adapting to the change in environment
  3. Perfective maintenance: - modifying or enhancing the system to meet the new requirements
  4. Preventive maintenance: - changes made to improve future maintainability.

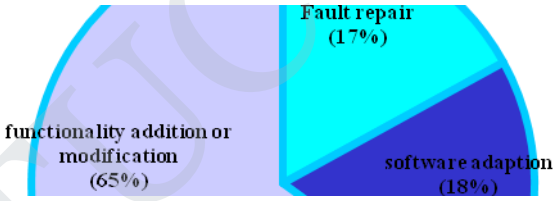



Fig: Maintenance Effort distribution

**Software Maintenance Process:**

Step 1: **Change request**

These are requests for system changes from users, customers or management. All change requests should be carefully analysed as part of the maintenance process and then implemented.

Step2: **Impact Analysis**

This involves identifying the system products affected by a change request, make an estimate of resources needed to effect the change and analyze the benefits of change.

Step 3: **System release planning**

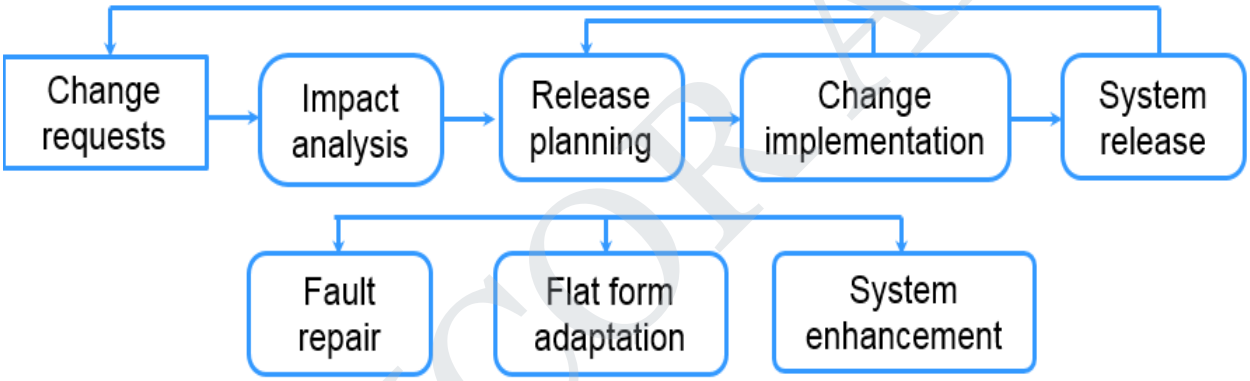
Here the schedule and the contents of software release is planned.

Step 4: **Change Implementation**

Implementation is done by designing the change and the coding and testing.

Step 5: **System Release**

During release documentation, software, hardware changes and data conversion should be described.

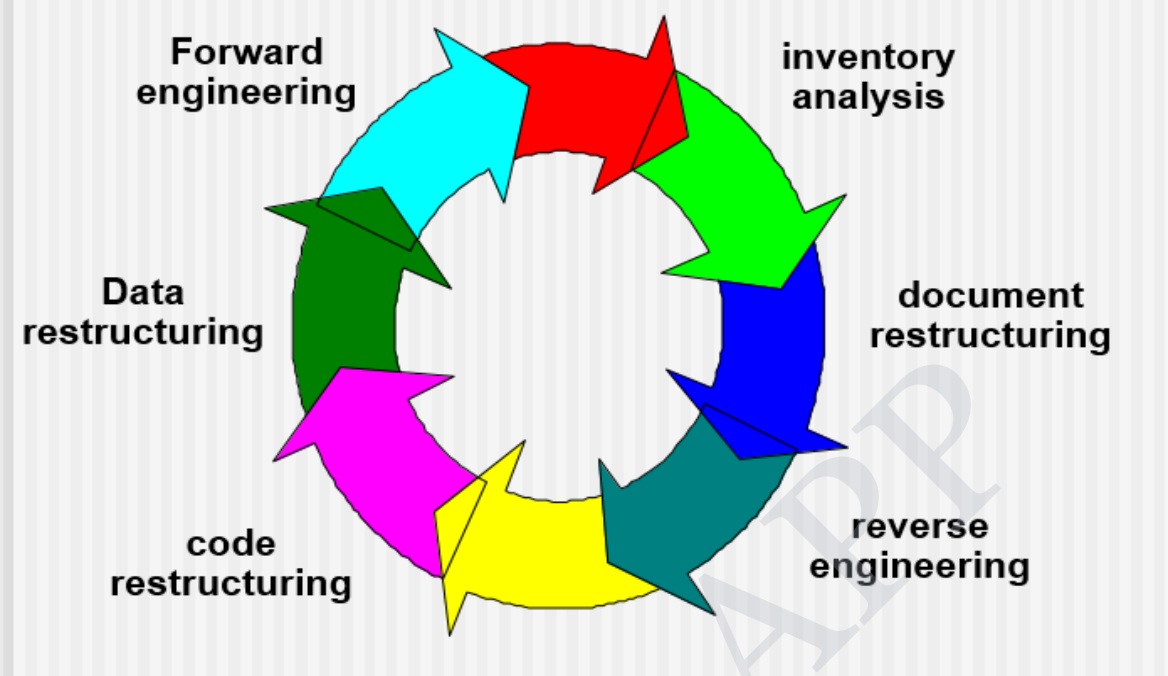



## 4.9 REENGINEERING

- Reengineering as the process of rebuilding the software products with increased functionality, better performance, greater reliability, and are easier to maintain.
- Software reengineering involves inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering.
- The final product for any reengineering process is a reengineered business process and/or the reengineered software to support it.

### 4.9.1 REENGINEERING PROCESS MODEL

- Software reengineering process model defines six activities that occur in a linear sequence, but this is not always the case.




**Software Reengineering activities**

**Inventory analysis**

- The inventory is like a spreadsheet model containing information that provides a detailed description about the project.
- By sorting this information according to business criticality, longevity, current maintainability and supportability, and other locally important criteria, candidates for reengineering can be identified.
- The inventory should be revisited on a regular cycle.

**Document restructuring**

- Documents are the most essential part of SDLC. Inadequate or inappropriate approach of documentation leads to document restructuring activity
- Three alternatives for document restructuring:
  - Instead of having time consuming documentation, remain with weak documentation
  - Update poor documents if they are used
  - Fully rewrite the documentation for critical systems focusing on the "essential minimum"

**Reverse engineering**

- Reverse engineering is the process of analyzing a program to create a representation of the program at a higher level of abstraction than source code.

- Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

**Code restructuring**

- Here the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured or rewritten in a more modern programming language.
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced.
- Internal code documentation is updated.

**Data restructuring**

- Data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.
- When data structure is weak, the data are reengineered because data architecture has a strong influence on program architecture and the algorithms that populate it.

**Forward engineering**

- This is also called reclamation or renovation, recovers design information from existing source code and uses this information to reconstitute the existing system to improve its overall quality and/or performance.

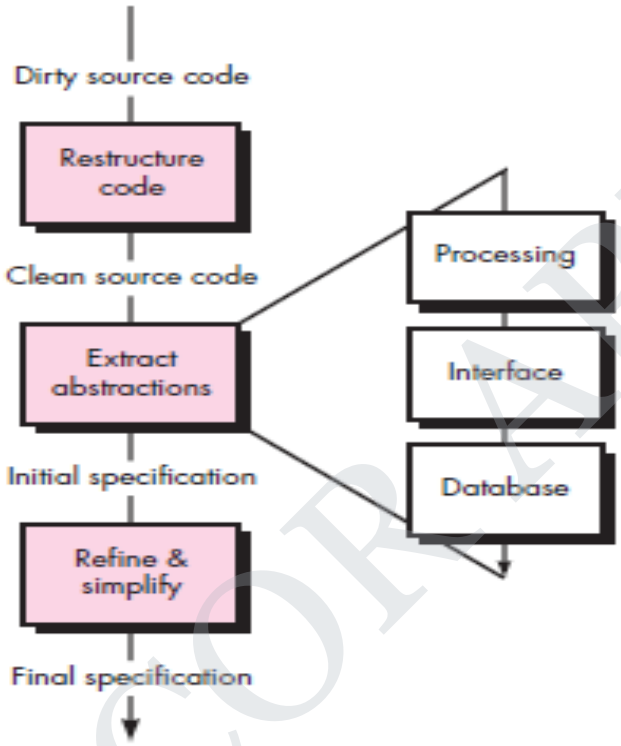
### 4.9.2 REVERSE ENGINEERING

- Reverse engineering is the process of analyzing a program to create a representation of the program at a higher level of abstraction than source code.
- Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

**Issues to be considered in Reverse Engineering:**

- The **completeness** of a reverse engineering process refers to the level of detail that is provided at an abstraction level. Completeness improves in direct proportion to the amount of analysis performed.
- **Interactivity** refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process.

- If the **directionality** of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can use it during maintenance. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.




- First the unstructured ("dirty") source code is restructured so that it contains only the structured programming constructs which makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.
- The core of reverse engineering is an activity called extract abstractions. In this stage, the old program is evaluated and from the source code develop a meaningful specification of the processing, user interface and the database that is used.
- Output of reverse engineering is a clear, unambiguous final specification that helps in easy understanding of the code.

**Reverse Engineering to Understand Data**

- Internal data structures - Reverse engineering techniques for internal program data focus on the definition of classes of objects. Program code is examined with the intention of grouping related program variables.

- Database structure – A database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

**Reverse Engineering to Understand Processing**

- Here the source code is analyzed at varying levels of detail (system, program, component, pattern, statement) to understand procedural abstractions and overall functionality

**Reverse Engineering User Interfaces**

- To fully understand an existing user interface, the structure and behavior of the interface must be specified.
- three basic questions that must be answered are:

1. What are the basic actions (e.g., key strokes or mouse operations) processed by the interface?
2. What is a compact description of the system's behavioral response to these actions?
3. What concept of equivalence of interfaces is relevant here?

### 4.9.3 FORWARD ENGINEERING

- The forward engineering process applies software engineering principles, concepts, and methods to re-create an existing application. The new user and technology requirements are integrated into the older application.

**Forward Engineering for Client-Server Architectures**

- A mainframe application that is reengineered into a client-server architecture has the following features:
  - application functionality migrates to each client computer
  - new GUI interfaces implemented at client sites
  - database functions allocated to servers
  - specialized functionality may remain at server site
  - new communications, security, archiving, and control requirements must be established at both client and server sites

Steps involved in Reengineering for client-server applications:

1. Reengineering process begins with a thorough analysis of the business environment that encompasses the existing mainframe.
2. Three layers of abstraction can be identified.
3. The database sits at the foundation of a client-server architecture and manages transactions and queries from server applications.
4. The functions of the existing database management system and the data architecture of the existing database must be reverse engineered as a precursor to the redesign of the database foundation layer.
5. The client applications layer implements business functions that are required by specific groups of end users.
6. In many instances, a mainframe application is segmented into a number of smaller, reengineered desktop applications. Communication among the desktop applications is controlled by the business rules layer.

**Forward Engineering for Object-Oriented Architectures**

- Here the existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created
- The use-cases are created if reengineered system extends functionality of application
- The data models created during reverse engineering are used with CRC modeling as a basis to define classes
- Class hierarchies, object-relationship models, object-behavior models are defined and then object-oriented design is started.
- A component-based process model may be used if a robust component library already exists
- For those components that must be built from scratch, it is possible to reuse algorithms and data structures from the original application

### 4.9.4 BUSINESS PROCESS REENGINEERING(BPR)

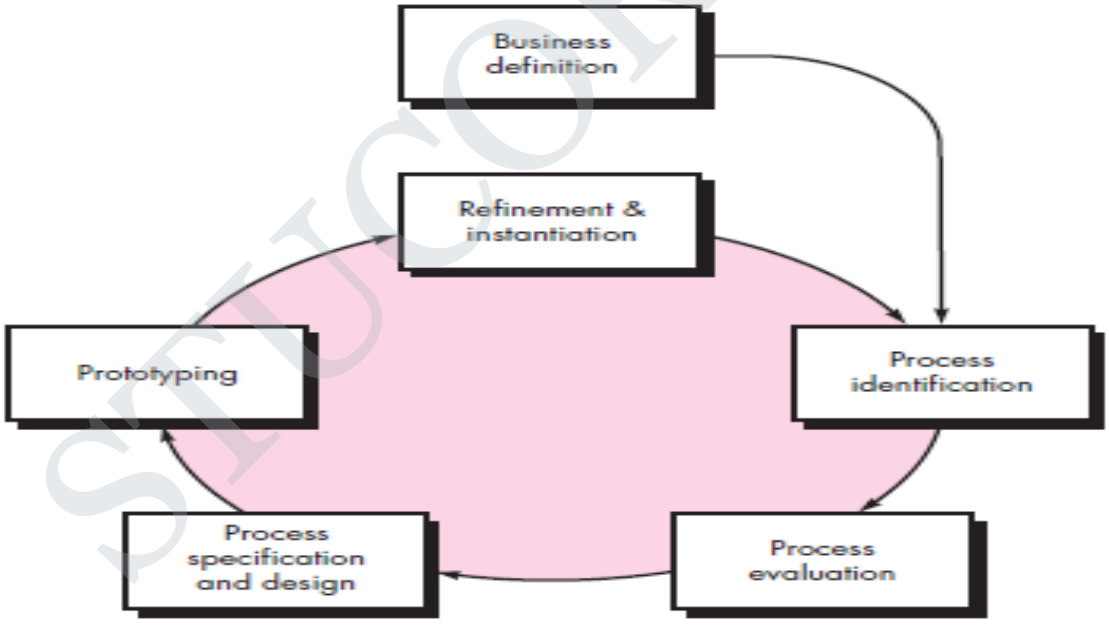
- Business process reengineering (BPR) defines business goals, evaluates existing business processes, and creates revised business processes that better meet current goals.
- A Business process is "a set of logically related tasks performed to achieve a defined business outcome".
- Every system is actually a hierarchy of subsystems. The overall business is segmented in the following manner:

**The business → business systems → business processes → business subprocesses**

- Each business system (also called business function) is composed of one or more business processes, and each business process is defined by a set of sub processes.
- BPR can be applied at any level of the hierarchy.

**BPR MODEL**

- BPR is an iterative model and has six activities




- **Business definition:** Business goals are identified within the context of four key drivers: cost reduction, time reduction, quality improvement, and personnel development and empowerment.
- **Process identification:** Processes that are critical to achieving the goals defined in the business definition are identified.

- **Process evaluation:** The existing process is thoroughly analyzed and measured.
- **Process specification and design:** Based on information obtained during the first three BPR activities, use-cases are prepared for each process that is to be redesigned.
- **Prototyping**: A redesigned business process must be prototyped before it is fully integrated into the business.
- **Refinement and instantiation:** Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

**BPR Principles**

- Organize around outcomes, not tasks.
- Have those who use the output of the process perform the process.
- Incorporate information processing work into the real work that produces the raw information.
- Treat geographically dispersed resources as though they were centralized.
- Link parallel activities instead of integrated their results. When different
- Put the decision point where the work is performed, and build control into the process.
- Capture data once, at its source.

# UNIT-V

# SOFTWARE PROJECT MANAGEMENT

**Estimation – LOC, FP Based Estimation, Make/Buy Decision COCOMO I & II Model – Project Scheduling – Scheduling, Earned Value Analysis Planning – Project Plan, Planning Process, RFP Risk Management – Identification, Projection – Risk Management-Risk Identification-RMMM Plan-CASE TOOLS.**

==========

## INTRODUCTION

- Software project management is managing people, process and problems during a software project
- It is the discipline of planning, organizing, and managing resources for the successful completion of specific project goals and objectives
- Software project management activities includes:
  - Project Planning
  - Estimation of the work
  - Estimation of resources required
  - Project scheduling
  - Risk management

## 5.1 ESTIMATION

- Estimation serves as a foundation for all other project planning actions.
- Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information, and the courage to commit to quantitative predictions when qualitative information is all that exists.
- Estimation carries inherent risk and this risk leads to uncertainty.

### 5.1.1 Factors influencing estimation risk

- Project complexity:
- Project size
- Problem decomposition
- Degree of structural uncertainty

### 5.1.2 Ways to achieve reliable estimation

- Software cost and effort estimation is not an exact estimate as the variables like human, technical, environmental, political can affect the ultimate cost of software and effort applied to develop it.
- To achieve reliable cost and effort estimates, a number of options arise:

**1. Delay estimation until late in the project**

**2. Base estimates on similar projects that have already been completed.**

**3. Use relatively simple decomposition techniques** to generate project cost and effort estimates.

**4. Use one or more empirical models** for software cost and effort estimation.

- Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion.
- A model is based on experience (historical data) and takes the form

$$d = f(vi)$$

where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).

### 5.1.3 Decomposition

- Software project estimation is a form of problem solving, so to solve it decompose the problem into a set of smaller manageable problems.
- The decomposition approach can be applied from two different points of view:
  - Decomposition of the problem
  - Decomposition of the process.
- Estimation uses one or both forms of partitioning.
- Before an estimate the scope of the software to be built must be understood and then generate an estimate of its "size."

**Software Sizing**

- Size refers to a quantifiable outcome of the software project.
- If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP).
- Four different approaches to the sizing problem:

1. **Fuzzy logic sizing**

This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this first identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

2. **Function point sizing**

The planner develops estimates of the information domain characteristics

3. **Standard component sizing**

- The standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions.
- Estimates the number of occurrences of each standard component and then use the historical data to estimate the delivered size per standard component.

4. **Change sizing**

- This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project.

### 5.1.4 PROBLEM-BASED ESTIMATION

- The lines of code and function points were the measures from which productivity metrics can be computed.
- LOC and FP data are used in two ways during software project estimation:
  (1)As estimation variables to "size" each element of the software and
  (2)As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.
- LOC and FP estimation are distinct estimation techniques.

### 5.1.4.1 LOC BASED ESTIMATION

- Steps involved in LOC based estimation:
  1. Find the bounded statement of software scope and
  2. Decompose the statement of scope into problem functions that can each be estimated individually.
  3. Estimate LOC (the estimation variable) for each function.
  4. A three-point or expected value can then be computed.

5. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates.

$$S = ( S_{opt} + 4 S_m + S_{pess} ) / 6$$

6. Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied.

- Local domain averages should then be computed by the project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters.
- The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning.
- When LOC is used as the estimation variable, the greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

**An Example of LOC-Based Estimation**

- Problem:

  Develop a software package for a computer-aided design application for mechanical components. The software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer.

- Preliminary statement of software scope can be developed:

  The mechanical CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which willbe displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

  This statement of scope is preliminary—it is not bounded.

| Function | Estimated LOC |
|---|---|
| user interface and control facilities (UICF) | 2,300 |
| two-dimensional geometric analysis (2DGA) | 5,300 |
| three-dimensional geometric analysis (3DGA) | 6,800 |
| database management (DBM) | 3,350 |
| computer graphics display facilities (CGDF) | 4,950 |
| peripheral control (PC) | 2,100 |
| design analysis modules (DAM) | 8,400 |
| *estimated lines of code* | 33,200 |

- Average productivity for systems of this type = 620 LOC/pm.
- Burdened labor rate =$8000 per month, the cost per line of code is approximately $13.
- Based on the LOC estimate and the historical productivity data,
  - **total estimated project cost is $431,000 ( 33,200*13) and**
  - **estimated effort is 54 person-months.(33,200/620)**

### 5.1.4.2 FP BASED ESTIMATION

- The function point (FP) metric can be used effectively as a means for measuring the functionality delivered by a system.
- Using historical data, the FP metric can then be used to

  (1) Estimate the cost or effort required to design, code, and test the software;

  (2) Predict the number of errors that will be encountered during testing; and

  (3) Forecast the number of components and/or the number of projected source lines in the implemented system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity.
- Information domain values are defined in the following manner:
  - **Number of external inputs (EIs).**
  - **Number of external outputs (EOs).**
  - **Number of external inquiries (EQs).**
  - **Number of internal logical files (ILFs).**
  - **Number of external interface files (EIFs).**
- Steps involved in FP based estimation:
  1. Find the bounded statement of software scope and

2. Decompose the statement of scope into problem functions that can each be estimated individually.
3. Estimate the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values
4. Using the estimates derive an FP value that can be tied to past data and used to generate an estimate.
5. Using historical data, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.
6. A three-point or expected value can then be computed.
7. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates.

$$S = ( S_{opt} + 4 S_m + S_{pess} ) / 6$$

8. Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied.

**Example for FP based estimation:**

- Once the information data have been collected, calculate the FP values by associating a complexity value with each count as shown below

| Information Domain Value | Count | | Weighting factor | | | | |
|---|---|---|---|---|---|---|---|
| | | | Simple | Average | Complex | | |
| External Inputs (EIS) | 3 | X | 3 | 4 | 6 | = | 9 |
| External Outputs (EOs) | 2 | X | 4 | 5 | 7 | = | 8 |
| External Inquiries (EQs) | 2 | X | 3 | 4 | 6 | = | 6 |
| Internal Logical Files (ILFs) | 1 | X | 7 | 10 | 15 | = | 7 |
| External Interface Files (EIFs) | 4 | X | 5 | 7 | 10 | = | 20 |
| Count Total | | | | | | | 50 |

- To compute function points (FP), the following relationship is used:
  - $FP_{estimated}$ = count-total × [0.65 + 0.01 × Σ ($F_i$)]

where $F_i$ (i = 1 to 14 are value adjustment factors) and count total is the sum of all FP entries obtained

- the various adjustment factors to be considered are:

| Factor | | Value |
|---|---|---|
| 1. | Backup and recovery | 4 |
| 2. | Data communications | 2 |
| 3. | Distributed processing | 0 |
| 4. | Performance critical | 4 |
| 5. | Existing operating environment | 3 |
| 6. | On-line data entry | 4 |
| 7. | Input transaction over multiple screens | 5 |
| 8. | ILFs updated online | 3 |
| 9. | Information domain values complex | 5 |
| 10. | Internal processing complex | 5 |
| 11. | Code designed for reuse | 4 |
| 12. | Conversion/installation in design | 3 |
| 13. | Multiple installations | 5 |
| 14. | Application designed for change | 5 |

For the given CAD software problem,

- $FP_{estimated} = 320 \times [0.65 + 0.01 \times \Sigma (F_i)] = 375$

- The organizational average productivity for systems of this type is 6.5 FP/pm.
- Based on a burdened labor rate of $8000 per month, the cost per FP is approximately $1230.
- Based on the FP estimate and the historical productivity data, **the total estimated project cost is $461,000 (375 * 1230) and the estimated effort is 58 person-months (375/6.5)**

**5.2 MAKE/BUY DECISION**

- Sometimes it is more cost effective to acquire the software rather than developing.
- Managers have many acquisition options.
  - Software may be purchased off the shelf
  - Full-experience or partial-experience software components may be acquired and integrated to meet specific needs
  - Software may be custom built by an outside contractor to meet the purchaser's specifications
- The make/buy decision can be made based on the following conditions
  - Will the software product be available sooner than internally developed software?
  - Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
  - Will the cost of outside support be less than the cost of internal support?

**CREATING A DECISION TREE**

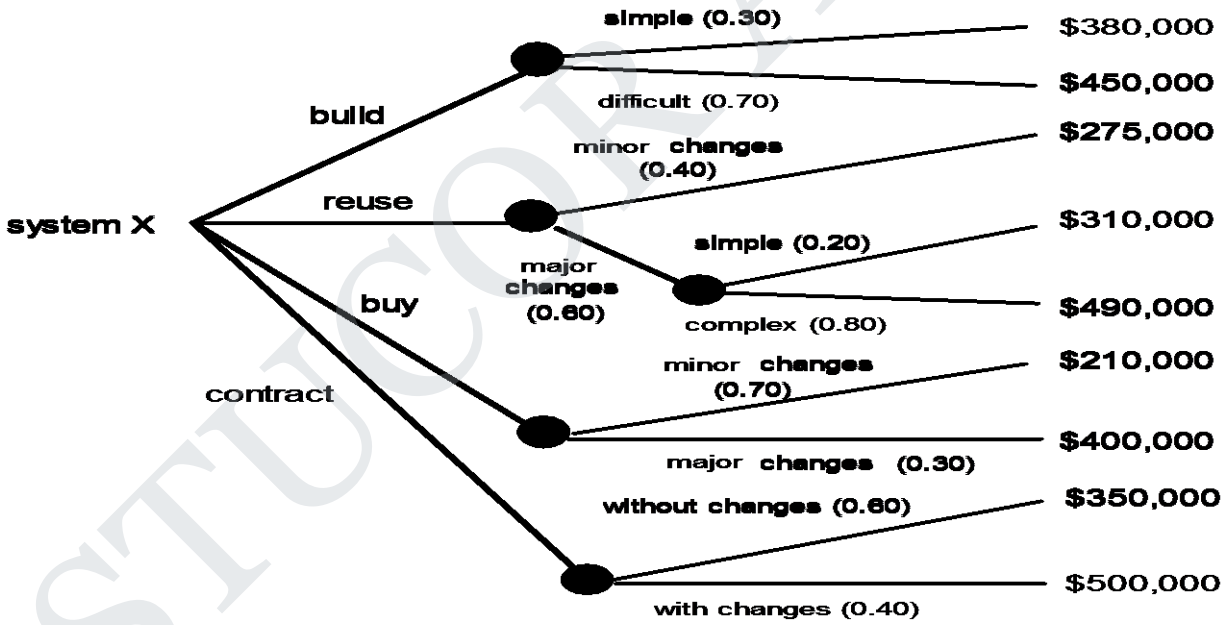
- Make/Buy decision can be made using statistical techniques such as decision tree analysis.

- Eg:

  Figure depicts a decision tree for a software- based system X. In this case, the software engineering organization can (1) build system X from scratch, (2) reuse existing partial-experience components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.

- If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. The project planner estimates that a difficult development effort will cost $450,000. A "simple" development effort is estimated to cost $380,000.

**Expected cost = ∑ (path probability) x (estimated path cost)**




- For example, the expected cost to build is:

**Expected cost = 0.30($380K)+0.70($450K)**

**= $429 K**

**Expected cost (reuse) = $382K**

**Expected cost (buy) = $267K**

**Expected cost (contract) = $410K**

## 5.3 COCOMO MODEL

Software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded.

Boehm's definition of organic, semidetached, and embedded systems is:

**Organic:**

A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

**Semidetached:**

A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

**Embedded:**

A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

**COCOMO MODEL**

- COCOMO (Constructive Cost Estimation Model) was proposed by Boehm
- According to Boehm, software cost estimation should be done through three stages:
  - Basic COCOMO
  - Intermediate COCOMO and
  - Complete COCOMO.

**Basic COCOMO Model**

- The basic COCOMO model gives an approximate estimate of the project parameters.
- The basic COCOMO estimation model is given by the following expressions:

$$\textbf{Effort} = a_1 \times (\text{KLOC})^{a2}\ \textbf{PM}$$

$$T_{dev} = b_1 \times (\text{Effort})^{b2}\ \textbf{Months}$$

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 products,

• $T_{dev}$ 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).

- The effort estimation is expressed in units of person-months (PM).

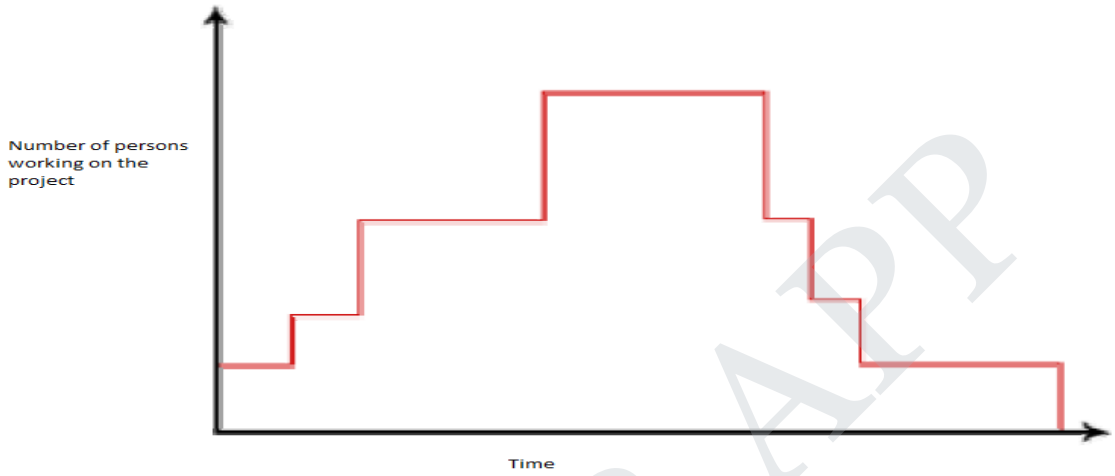



Fig: Person-Month Curve

- According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC.
- The values of $a_1$, $a_2$, $b_1$, $b_2$ for different categories of products (i.e. organic, semidetached, and embedded)are summarized below.

**Estimation of development effort**

- For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

  Organic : **Effort** = $2.4(KLOC)^{1.05}$ **PM**

  Semi-detached : **Effort** = $3.0(KLOC)^{1.12}$ **PM**

  Embedded : **Effort** = $3.6(KLOC)^{1.20}$ **PM**

**Estimation of development time**

- For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

  Organic : $T_{dev} = 2.5(Effort)^{0.38}$ **Months**

  Semi-detached : $T_{dev} = 2.5(Effort)^{0.35}$ **Months**

  Embedded : $T_{dev} = 2.5(Effort)^{0.32}$ **Months**

- Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes.
- Fig. shows a plot of estimated effort versus product size.

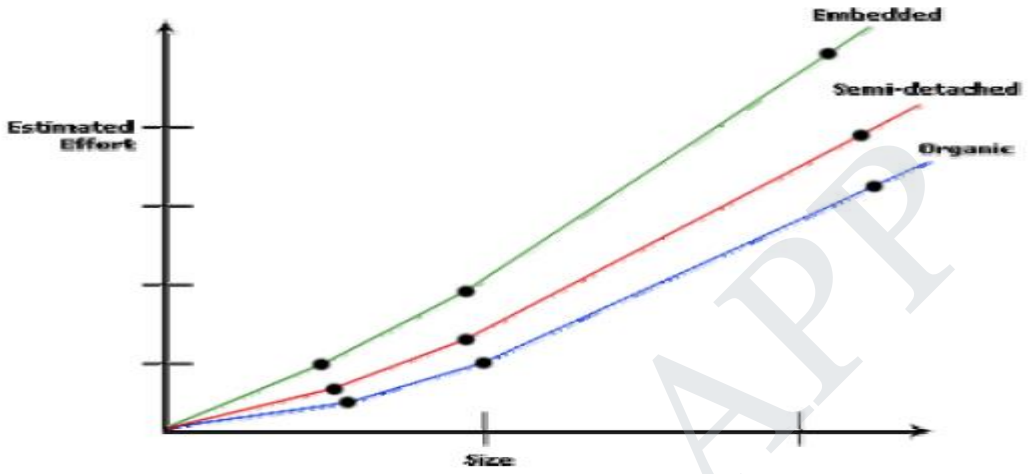



Fig: Effort Vs Product Size

- The effort is super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.
- The development time versus the product size in KLOC is plotted in fig.

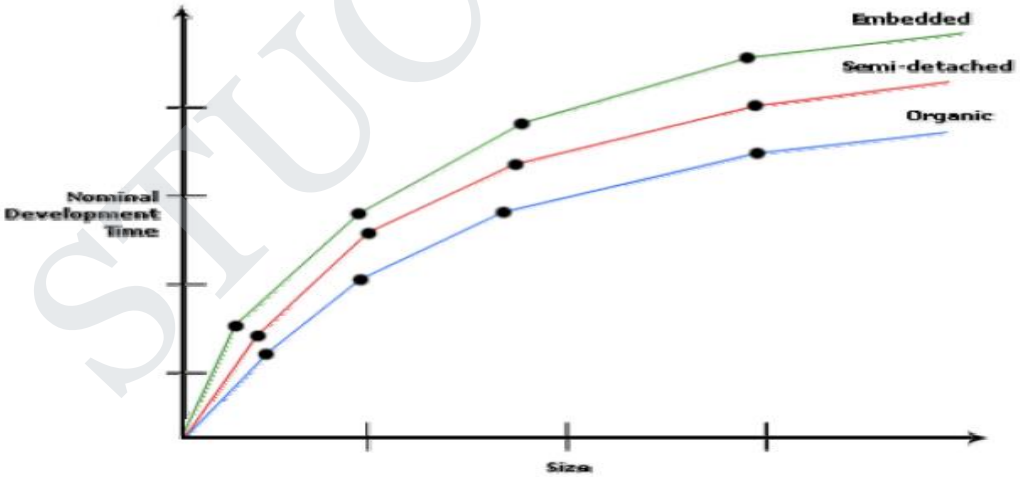



Fig: Development Time Vs Size

- The development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately.
- The development time is roughly the same for all the three categories of products.

- It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate.

**Intermediate COCOMO model**

- The basic COCOMO model assumes that effort and development time are functions of the product size alone.
- But other project parameters also affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account.
- The intermediate COCOMO model recognizes this and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
- In general, the cost drivers 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, 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.

**Complete COCOMO model**

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics.
- For example, some sub-systems may be considered as organic type, some semidetached, and some embedded, also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on.

- The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems.
- The cost of each subsystem is estimated separately.
- This approach reduces the margin of error in the final estimate.

**Example:**

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05} = 91$ PM**

Nominal development time = **$2.5 \times (91)^{0.38} = 14$ months**

Cost required to develop the product = **14 x 15,000**

= **Rs. 210,000/-**

## 5.4 COCOMO II MODEL

- COCOMO stands for COnstructive COst MOdel
- COCOMO II is actually a hierarchy of three estimation models

**COCOMO Models**

- **Application composition model -** Used during the early stages of software engineering when the following are important
  - Prototyping of user interfaces
  - Consideration of software and system interaction
  - Assessment of performance
  - Evaluation of technology maturity
- **Early design stage model** – This is used once requirements have been stabilized and basic software architecture has been established
- **Post-architecture stage model** – This model is used during the construction of the software

- This requires sizing information and accepts it in three forms:
  - object points,

- function points
- lines of source code

- Object point is an indirect software measure that is computed using counts of the number of

  (1) Screens

  (2) Reports, and

  (3) Components likely to be required to build the application.

- Each object instance is classified into one of three complexity levels as given below

| Object type | Complexity weight | | |
|---|---|---|---|
| | Simple | Medium | Difficult |
| Screen | 1 | 2 | 3 |
| Report | 2 | 5 | 8 |
| 3GL component | | | 10 |

- Complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.
- The object point count is then determined by multiplying the number of object instances by the weighting factor and summing to obtain a **total object point count**.
- The percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$NOP = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

Where NOP is defined as new object points.

- To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived.

$$PROD = \frac{NOP}{\text{person-month}}$$

- Productivity rate is based on different levels of developer's experience and environment maturity

| Developer's experience/capability | Very low | Low | Nominal | High | Very high |
|---|---|---|---|---|---|
| Environment maturity/capability | Very low | Low | Nominal | High | Very high |
| PROD | 4 | 7 | 13 | 25 | 50 |

- Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$

- In more advanced COCOMO II models, 12 a variety of scale factors, cost drivers, and adjustment procedures are required.

**COCOMO Cost Drivers**

- **Personnel Factors**
  - Programming language experience
  - Personnel capability and experience
  - Language and tool experience etc
- **Product Factors**
  - Database size
  - Required reusability
  - Product reliability and complexity etc
- **Project Factors**
  - Use of software tools
  - Required development schedule
  - Multi-site development etc

## 5.5 PROJECT SCHEDULING

- Software project scheduling is nothing but allocating the estimated effort to specific software engineering tasks.
- The schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This schedule identifies all major process framework activities and the product functions to which they are applied.
- As the project gets developed, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks required to accomplish an activity are identified and scheduled.

**Basic Principles**

- **Compartmentalization**

  Decompose the project into a number of manageable activities and tasks.

- **Interdependency**

The interdependency between each module must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some may be independent.

- **Time allocation**

  Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). Each task must be assigned a start date and a completion date .

- **Effort validation**

  Every project has a defined number of people on the software team. So ensure that no more than the allocated number of people has been scheduled at any given time.

- **Defined responsibilities**

  Every task that is scheduled should be assigned to a specific team member.

- **Defined outcomes**

  Every task that is scheduled should have a defined outcome.Work products are often combined in deliverables.

- **Defined milestones**

  Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

- Each of these principles is applied as the project schedule evolves.

## STEP INVOLVED IN PROJECT SCHEDULING:

1. Establish a meaningful task set.
2. Define a task network.
3. Use scheduling tools to develop a time-line chart.
4. Define schedule tracking mechanisms

### STEP 1: DEFINING A TASK SET FOR THE SOFTWARE PROJECT

- A task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project.
- To develop a project schedule, a task set must be distributed on the project time line.
- The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work.

Different types of projects:

1. Concept development projects that are initiated to explore some new business concept or application of some new technology.
2. New application development projects that are undertaken as a consequence of a specific customer request.
3. Application enhancement projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
4. Application maintenance projects that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
5. Reengineering projects that are undertaken with the intent of rebuilding an existing system in whole or in part.

Factors influencing the task set to be chosen are:

- Size of the project, number of potential users, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, project staff, and reengineering factors etc.

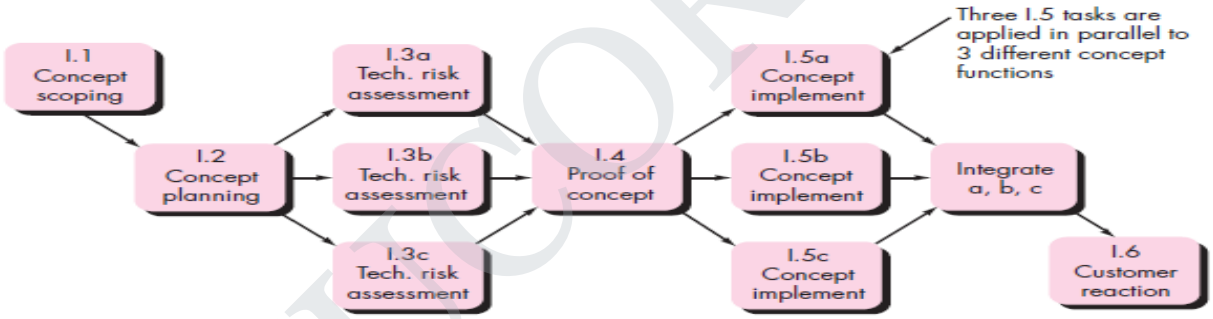
**A Task Set Example:**

- Concept development projects are initiated when the potential for some new technology must be explored.
- Concept development projects are approached by applying the following actions:
  1. Concept scoping determines the overall scope of the project.
  2. Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.
  3. Technology risk assessment evaluates the risk associated with the technology to be implemented as part of the project scope.
  4. Proof of concept demonstrates the viability of a new technology in the software context.
  5. Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

6. Customer reaction to the concept solicits feedback on a new technology concept and targets specific customer applications.

**STEP 2: DEFINING A TASK NETWORK**

- A *task network,* also called an *activity network*
- It is a graphic representation of the task flow for a project.
- A task network depicts the task length, sequence, concurrency, and dependency
- Task network points out inter-task dependencies in the project
- The critical path
  - A single path leading from start to finish in a task network
  - It contains the sequence of tasks that must be completed on schedule to complete the entire project in time
  - It also determines the minimum duration of the project
- Eg: Task network for a concept development project.




**STEP 3: USE SCHEDULING TOOLS TO DEVELOP A TIME LINE CHART**

- The two project scheduling methods that can be applied to software development are
  - *Program evaluation and review technique* (PERT)
  - *Critical path method* (CPM)
- Interdependencies among tasks are defined using a task network.
- Tasks, sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions.
- Both PERT and CPM provide quantitative tools that allows to

(1) Determine the critical path—the chain of tasks that determines the duration of the project,

(2) Establish "most likely" time estimates for individual tasks by applying statistical models,

(3) Calculate "boundary times" that define a time "window" for a particular task.

**Time-Line Charts**

- A *time-line chart*, also called a *Gantt chart*.
- A time-line chart can be developed for the entire project or a separate chart can be developed for each project function.
- All project tasks are listed in the left hand column
- The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)
- To the right are columns representing dates on a calendar
- The length of a horizontal bar on the calendar indicates the duration of the task
- Eg:

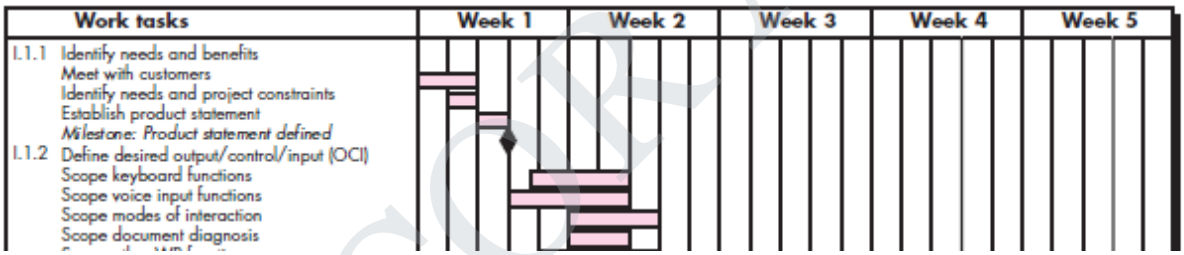



Fig: Time-line chart

- When multiple bars occur at the same time interval on the calendar, this implies task concurrency
- A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero
- Software project scheduling tools produce *project tables* from the information available in the time – line chart
- Project table is a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information.
- Project tables helps to track progress.
- Eg: Project table

| Work tasks | Planned start | Actual start | Planned complete | Actual complete | Assigned person | Effort allocated | Notes |
|---|---|---|---|---|---|---|---|
| I.1.1 Identify needs and benefits | | | | | | | Scoping will require more effort/time |
| Meet with customers | wk1, d1 | wk1, d1 | wk1, d2 | wk1, d2 | BLS | 2 p-d | |
| Identify needs and project constraints | wk1, d2 | wk1, d2 | wk1, d2 | wk1, d2 | JPP | 1 p-d | |
| Establish product statement | wk1, d3 | wk1, d3 | wk1, d3 | wk1, d3 | BLS/JPP | 1 p-d | |
| *Milestone: Product statement defined* | wk1, d3 | wk1, d3 | wk1, d3 | wk1, d3 | | | |
| I.1.2 Define desired output/control/input (OCI) | | | | | | | |
| Scope keyboard functions | wk1, d4 | wk1, d4 | wk2, d2 | | BLS | 1.5 p-d | |
| Scope voice input functions | wk1, d3 | wk1, d3 | wk2, d2 | | JPP | 2 p-d | |
| Scope modes of interaction | wk2, d1 | | wk2, d3 | | MLL | 1 p-d | |
| Scope document diagnostics | wk2, d1 | | wk2, d2 | | BLS | 1.5 p-d | |
| Scope other WP functions | wk1, d4 | wk1, d4 | wk2, d3 | | JPP | 2 p-d | |
| Document OCI | wk2, d1 | | wk2, d3 | | MLL | 3 p-d | |
| FTR: Review OCI with customer | wk2, d3 | | wk2, d3 | | all | 3 p-d | |
| Revise OCI as required | wk2, d4 | | wk2, d4 | | all | 3 p-d | |
| *Milestone: OCI defined* | wk2, d5 | | wk2, d5 | | | | |
| I.1.3 Define the function/behavior | | | | | | | |

## STEP 4: DEFINE SCHEDULE TRACKING MECHANISMS

### Tracking the Schedule

- Qualitative approaches for tracking:
    1. Conducting periodic project status meetings in which each team member reports progress and problems
    2. Evaluating the results of all reviews conducted throughout the software engineering process
    3. Determining whether formal project milestones have been accomplished by the scheduled date
    4. Comparing the actual start date to the planned start date for each project task listed in the resource table
    5. Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon
- Quantitative approach:
    6. Using earned value analysis to assess progress quantitatively
- Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff.
- If things are going well, control is light. But when problems occur, the project manager must apply tight control to reconcile the problems as quickly as possible. For example:
    - Staff may be redeployed
    - The project schedule may be redefined

- After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.
- When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called **time-boxing.**

**Time-boxing strategy:**

- The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline.
- An incremental software paradigm is applied to the project
- The tasks associated with each increment are "time-boxed" (i.e., given a specific start and stop time) by working backward from the delivery date
- A "box" is put around each task. When a task hits the boundary of its time box, work stops and the next task begins.
- This approach succeeds based on the premise that when the time-box boundary is encountered, it is likely that 90% of the work is complete
- The remaining 10% of the work can be
  - Delayed until the next increment
  - Completed later if required
- Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

## 5.6 EARNED VALUE ANALYSIS

- Earned value analysis is a measure of progress by assessing the percent of completeness for a project
- It gives accurate and reliable readings of performance very early into a project
- It provides a common value scale (i.e., time) for every project task, regardless of the type of work being performed
- The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total
- To determine the earned value, the following steps are performed:
  1. The **budgeted cost of work scheduled (BCWS)** is determined for each work task represented in the schedule. The value of BCWS is the sum of the

BCWS*i* values for all work tasks that should have been completed by that point in time on the project schedule.

2. The BCWS values for all work tasks are summed to derive the **budget at completion (BAC).**

   BAC $= \sum (BCWS_k)$ for all tasks k

3. Next, the value for **budgeted cost of work performed (BCWP)** is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

- **BCWS** represents the budget of the activities that were planned to be completed and **BCWP** represents the budget of the activities that actually were completed.
- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

  **Schedule performance index, SPI = BCWP / BCWS**

  **Schedule variance, SV = BCWP - BCWS**

  where,

  → SPI is an indication of the efficiency with which the project is utilizing scheduled resources.

  →SPI value close to 1.0 indicates efficient execution of the project schedule.

  →SV is simply an absolute indication of variance from the planned schedule.
- Percentage scheduled for completion provides an indication of the percentage of work that should have been completed by time *t*.

  **Percent scheduled for completion = BCWS / BAC**
- Percent complete provides a quantitative indication of the percent of completeness of the project at a given point in time *t*.

  **Percent complete = BCWP / BAC**
- The actual cost of work performed (ACWP) is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule.
- It is then possible to compute

  **Cost performance index, CPI= BCWP /ACWP**

  **Cost variance, CV =BCWP - ACWP**

→ A CPI value close to 1.0 provides a strong indication that the project is within its defined budget.

→CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

- Earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables to take corrective action before a project crisis develops.

## 5.7 PROJECT PLANNING

- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- Estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.

**Task Set for Project Planning**

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks.
4. Define required resources.
    a. Determine required human resources.
    b. Define reusable software resources.
    c. Identify environmental resources.
5. Estimate cost and effort.
    a. Decompose the problem.
    b. Develop two or more estimates using size, function points, process tasks, or use cases.
    c. Reconcile the estimates.
6. Develop a project schedule.
    a. Establish a meaningful task set.
    b. Define a task network.
    c. Use scheduling tools to develop a time-line chart.
    d. Define schedule tracking mechanisms

### 5.7.1 SOFTWARE SCOPE AND FEASIBILITY

- Software scope describes the functions and features that are to be delivered to end users.

- Functions described in the statement of scope are evaluated and refined to provide more detail at the beginning of estimation.

**Software feasibility** has four solid dimensions:

- **Technology—** Is a project technically feasible? Determine whether the technology needed to develop the project is available.
- **Finance—**Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford?
- **Time—**Will the project's time-to-market beat the competition?
- **Resources**— Does the organization have the resources needed to succeed?

## 5.7.2 RESOURCES

- The three major categories of software engineering resources are:
  - People
  - Reusable software components
  - Development environment (hardware and software tools).
- Each resource is specified with four characteristics:
  - description of the resource
  - a statement of availability
  - time when the resource will be required
  - Duration of time that the resource will be applied.

**Human Resources**

- The planner begins by evaluating software scope and selecting the skills required to complete development.
- Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., tele communications, database, client-server) are specified.
- For relatively small projects, a single individual may perform all software engineering tasks, consulting with specialists as required.
- For larger projects, the software team may be geographically dispersed across a number of different locations.
- The number of people required for a software project can be determined only after the estimation of development effort

**Reusable Software Resources**

- Component-based software engineering (CBSE) emphasizes reusability—the creation and reuse of software building blocks. Such building blocks, often called components.
- Four software resource categories that should be considered as planning proceeds:
  - **Off-the-shelf components.**

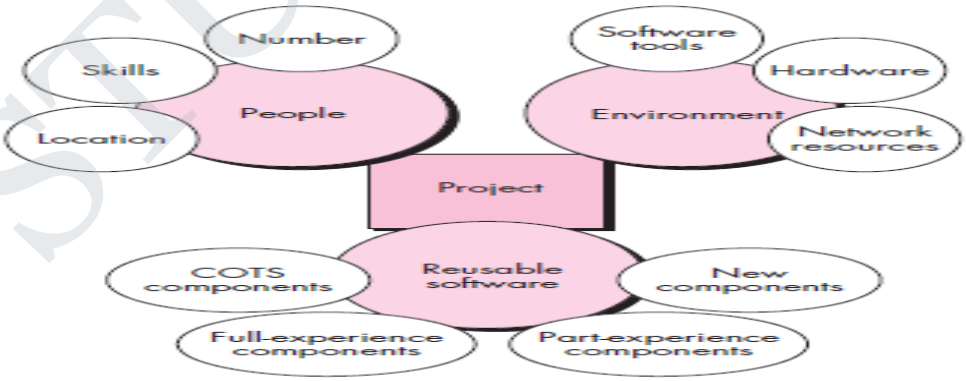
    Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.
  - **Full-experience components.**

    Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project.

    Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low risk.
  - **Partial-experience components.**

    Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification.

    Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.




  - **New components.**

    Software components must be built by the software team specifically for the needs of the current project.

Reusable software components are often neglected during planning. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

**Environmental Resources**

- The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software.
- Hardware provides a platform that supports the tools required to produce the work products that are an outcome of good software engineering practice.
- The time window required for hardware and software should also be prescribed and verify that these resources will be available.
- Each hardware element must be specified as part of planning.

## 5.8 RISK MANAGEMENT

- Risk analysis and management are the actions which help to understand and manage uncertainty.
- A risk is a potential problem which may or may not happen.
- To manage risk, first identify the risk, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur

**RISK CATEGORIZATION**

- The two broad categories of risks are:
  - **Product specific risk**

    - Risks that can be identified only with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - **Generic risk** - Risks that are a potential threat to every software project

**Categories of Product specific risk:**

- **Project risks**
  - Project risks threaten the project plan
  - If they occur, then the project schedule will slip and the costs will increase
- **Technical risks**
  - This threaten the quality and timeliness of the software to be produced
  - If they occur, implementation becomes difficult or impossible

- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product
  - Sub-categories of Business risks
    - Market risk – building an excellent product or system that no one really wants
    - Strategic risk – building a product that no longer fits into the overall business strategy for the company
    - Sales risk – building a product that the sales force doesn't understand how to sell
    - Management risk – losing the support of senior management due to a change in focus or a change in people
    - Budget risk – losing budgetary or personnel commitment

**Categories of Generic risks:**

- **Known risks**
  - Known risks are those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- **Predictable risks**
  - Predictable risks are extrapolated from past project experience (e.g., past turnover)
- **Unpredictable risks**
  - Those risks that are extremely difficult to identify in advance are referred as unpredictable risks.

**Reactive vs. Proactive Risk Strategies**

- Reactive risk strategies
  - Nothing is done about risks until something goes wrong
    - The team then flies into action in an attempt to correct the problem rapidly (also referred as firefighting mode)
  - Crisis management is the choice of management techniques
- Proactive risk strategies

- Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner.

**Seven Principles of Risk Management**

- **Maintain a global perspective**
  - View software risks within the context of a system and the business problem that is intended to solve
- **Take a forward-looking view**
  - Think about risks that may arise in the future; establish contingency plans
- **Encourage open communication**
  - Encourage all stakeholders and users to point out risks at any time
- **Integrate risk management**
  - Integrate the consideration of risk into the software process
- **Emphasize a continuous process of risk management**
  - Modify identified risks as more becomes known and add new risks as better insight is achieved
- **Develop a shared product vision**
  - A shared vision by all stakeholders facilitates better risk identification and assessment
- **Encourage teamwork when managing risk**
  - Pool the skills and experience of all stakeholders when conducting risk management activities

**STEPS INVOLVED IN RISK MANAGEMENT**

1. Identify possible risks; recognize what can go wrong
2. Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3. Rank the risks by probability and impact
4. Develop a contingency plan to manage those risks having high probability and high impact

**STEP 1: RISK IDENTIFICATION**

- Risk identification is a systematic attempt to specify threats to the project plan

- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- To identify the risk create, Risk item checklist.

**Risk Item Checklist**

- This focuses on known and predictable risks in specific subcategories
- The check list can be organized in several ways
    1. A list of characteristics relevant to each risk subcategory
    2. Questionnaire that leads to an estimate on the impact of each risk
    3. A list containing a set of risk component and drivers and their probability of occurrence

**1. Known and Predictable Risk Categories**

- Product size – risks associated with overall size of the software to be built
- Business impact – risks associated with constraints imposed by management or the marketplace
- Customer characteristics – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- Process definition – risks associated with the degree to which the software process has been defined and is followed
- Development environment – risks associated with availability and quality of the tools to be used to build the project
- Technology to be built – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- Staff size and experience – risks associated with overall technical and project experience of the software engineers who will do the work

**2. Sample Questionnaire on Project Risk**

1) Are requirements fully understood by the software engineering team and its customers?
2) Have customers been involved fully in the definition of requirements?
3) Is the project scope stable?
4) Does the software engineering team have the right mix of skills?
5) Are project requirements stable?
6) Does the project team have experience with the technology to be implemented?

7) Is the number of people on the project team adequate to do the job?

**3. Risk Components and Drivers**

- The project manager identifies the risk drivers that affect the following risk components
  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of **four impact levels**
  - Negligible
  - Marginal
  - Critical
  - Catastrophic
- Risk drivers can be assessed as **impossible, improbable, probable, and frequent**

**STEP 2 & 3: RISK ANALYSIS & RANKING:**

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real
  - The consequence of the problems associated with the risk

Steps involved in Risk Estimation:

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Assess the overall accuracy of the risk projection so that there will be no misunderstandings

Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk Summary – short description of the risk

- Risk Category – one of seven risk categories (Problem size, business impact etc )
- Probability – estimation of risk occurrence based on group input
- Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
- RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

| Risks | Category | Probability | Impact | RMMM |
|---|---|---|---|---|
| Size estimate may be significantly low | PS | 60% | 2 | |
| Larger number of users than planned | PS | 30% | 3 | |
| Less reuse than planned | PS | 70% | 2 | |
| End-users resist system | BU | 40% | 3 | |
| Delivery deadline will be tightened | BU | 50% | 2 | |
| Funding will be lost | CU | 40% | 1 | |
| Customer will change requirements | PS | 80% | 2 | |
| Technology will not meet expectations | TE | 30% | 1 | |
| Lack of training on tools | DE | 80% | 3 | |
| Staff inexperienced | ST | 30% | 2 | |
| Staff turnover will be high | ST | 60% | 2 | |
| Σ | | | | |
| Σ | | | | |
| Σ | | | | |

Fig: Sample Risk Table

- After identifying the category, probability of occurrence and impacts sort the rows by probability and impact in descending order
- Draw a **horizontal cutoff line** in the table that indicates the risks that will be given further attention
- Risks that fall below the line are reevaluated to accomplish second-order prioritization.
- Probability has a distinct influence on management concern.
- A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

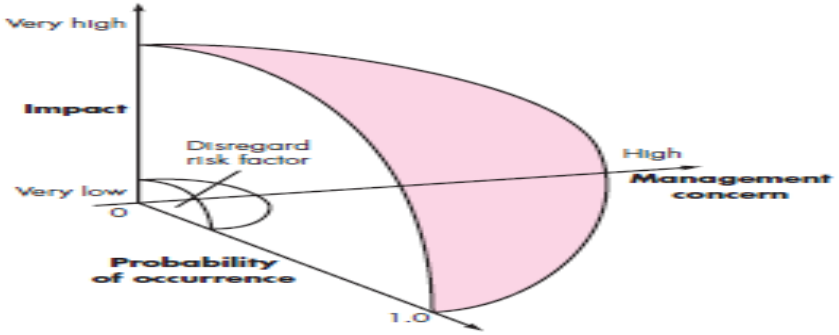


Fig: Risk and Management concern

**Assessing Risk Impact**

- Three factors affect the consequences that are likely if a risk does occur
  - **Its nature** – This indicates the problems that are likely if the risk occurs
  - **Its scope** – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - **Its timing** – This considers when and for how long the impact will be felt
- The overall risk exposure formula is **RE = P x C**
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

**STEP 4: RISK MITIGATION, MONITORING, AND MANAGEMENT**

- An effective strategy for dealing with risk must consider three issues
  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning
- **Mitigation** : How can we avoid the risk?
- **Monitoring** : What factors can we track that will enable us to determine if the risk is becoming more or less likely?
- **Management** : what contingency plans do we have if the risk becomes a reality?

**Example:**

**Risk: Technology Does Not Meet Specifications**

- **Mitigation**

  In order to prevent this from happening, meetings (formal and informal) will be held with the customer on a routine business. This insures that the product we are producing, and the specifications of the customer are equivalent.

- **Monitoring**

The meetings with the customer should ensure that the customer and our organization understand each other and the requirements for the product.

- **Management**

  Should the development team come to the realization that their idea of the product specifications differs from those of the customer, the customer should be immediately notified and whatever steps necessary to rectify this problem should be done. Preferably a meeting should be held between the development team and the customer to discuss at length this issue.

**The RMMM Plan**

- The RMMM plan may be a part of the software development plan or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
  - Risk mitigation is a problem avoidance activity
  - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
  - To assess whether predicted risks do, in fact, occur
  - To ensure that risk aversion steps defined for the risk are being properly applied
  - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project.

**UNIT V**

**PROJECT MANAGEMENT**

**Estimation – FP Based, LOC Based, Make/Buy Decision, COCOMO II - Planning – Project Plan, Planning Process, RFP Risk Management – Identification, Projection, RMMM - Scheduling and Tracking – Relationship between people and effort, Task Set & Network, Scheduling, EVA – Process and Project Metrics**

## 5.1 INTRODUCTION

- Software project management is managing people, process and problems during a software project
- It is the discipline of planning, organizing, and managing resources for the successful completion of specific project goals and objectives
- Software project management activities includes:
  - Project Planning
  - Estimation of the work
  - Estimation of resources required
  - Project scheduling
  - Risk management

## 5.2 PROJECT PLANNING

- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- Estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.

**Task Set for Project Planning**

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks.
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.

5. Estimate cost and effort.
    a. Decompose the problem.
    b. Develop two or more estimates using size, function points, process tasks, or use cases.
    c. Reconcile the estimates.
6. Develop a project schedule.
    a. Establish a meaningful task set.
    b. Define a task network.
    c. Use scheduling tools to develop a time-line chart.
    d. Define schedule tracking mechanisms

## 5.3 ESTIMATION

- Estimation serves as a foundation for all other project planning actions.
- Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information, and the courage to commit to quantitative predictions when qualitative information is all that exists.
- Estimation carries inherent risk and this risk leads to uncertainty.

**Factors influencing estimation risk**

- Project complexity:
- Project size
- Problem decomposition
- Degree of structural uncertainty

**Ways to achieve reliable estimation**

- Software cost and effort estimation is not an exact estimate as the variables like human, technical, environmental, political can affect the ultimate cost of software and effort applied to develop it.
- To achieve reliable cost and effort estimates, a number of options arise:

1. **Delay estimation until late in the project**

    (It's possible to achieve 100 percent accurate estimates after the project is complete! So not practical).

**2. Base estimates on similar projects that have already been completed.**

This can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

**3. Use relatively simple decomposition techniques** to generate project cost and effort estimates.

**4. Use one or more empirical models** for software cost and effort estimation.

The remaining options (3 & 4) are viable approaches to software project estimation.

- Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion.
- Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right.
- A model is based on experience (historical data) and takes the form

  d = f (vi)

  where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).
- Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating.
- Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation.

**Decomposition**

- Software project estimation is a form of problem solving, so to solve it decompose the problem into a set of smaller manageable problems.
- The decomposition approach can be applied from two different points of view:
  - Decomposition of the problem
  - Decomposition of the process.

- Estimation uses one or both forms of partitioning.
- Before an estimate the scope of the software to be built must be understood and then generate an estimate of its "size."

**Software Sizing**

- The accuracy of a software project estimate is predicated on a number of things:
  - the degree to which you have properly estimated the size of the product to be built;
  - the ability to translate the size estimate into human effort, calendar time, and dollars
  - the degree to which the project plan reflects the abilities of the software team; and
  - the stability of product requirements and the environment that supports the software engineering effort.
- Here size refers to a quantifiable outcome of the software project.
- If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP).
- Four different approaches to the sizing problem:

1. **Fuzzy logic sizing**

   This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this first identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

2. **Function point sizing**

   The planner develops estimates of the information domain characteristics

3. **Standard component sizing**
   - Software is composed of a number of different "standard components" that are generic to a particular application area.
   - For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions.
   - Estimates the number of occurrences of each standard component and then use the historical data to estimate the delivered size per standard component.

4. **Change sizing**
   - This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project.
   - The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

## 5.4 PROBLEM-BASED ESTIMATION

- The lines of code and function points were the measures from which productivity metrics can be computed.
- LOC and FP data are used in two ways during software project estimation:
  (1)As estimation variables to "size" each element of the software and
  (2)As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.
- LOC and FP estimation are distinct estimation techniques.

## LOC BASED ESTIMATION

- Steps involved in LOC based estimation:
  1. Find the bounded statement of software scope and
  2. Decompose the statement of scope into problem functions that can each be estimated individually.
  3. Estimate LOC (the estimation variable) for each function.
  4. A three-point or expected value can then be computed.
  5. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates.

     $$S = ( S_{opt} + 4 S_m + S_{pess} ) / 6$$

  6. Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied.

- Local domain averages should then be computed by the project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters.
- The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning.
- When LOC is used as the estimation variable, the greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

**An Example of LOC-Based Estimation**

- Problem:

  Develop a software package for a computer-aided design application for mechanical components. The software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer.

- Preliminary statement of software scope can be developed:

  The mechanical CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which willbe displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

  This statement of scope is preliminary—it is not bounded.

| Function | Estimated LOC |
|---|---|
| user interface and control facilities (UICF) | 2,300 |
| two-dimensional geometric analysis (2DGA) | 5,300 |
| three-dimensional geometric analysis (3DGA) | 6,800 |
| database management (DBM) | 3,350 |
| computer graphics display facilities (CGDF) | 4,950 |
| peripheral control (PC) | 2,100 |
| design analysis modules (DAM) | 8,400 |
| *estimated lines of code* | 33,200 |

- Average productivity for systems of this type = 620 LOC/pm.
- Burdened labor rate =$8000 per month, the cost per line of code is approximately $13.
- Based on the LOC estimate and the historical productivity data,
  - **total estimated project cost is $431,000 ( 33,200*13) and**
  - **estimated effort is 54 person-months.(33,200/620)**

**FP BASED ESTIMATION**

- The function point (FP) metric can be used effectively as a means for measuring the functionality delivered by a system.
- Using historical data, the FP metric can then be used to

  (1) Estimate the cost or effort required to design, code, and test the software;

  (2) Predict the number of errors that will be encountered during testing; and

  (3) Forecast the number of components and/or the number of projected source lines in the implemented system.

- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity.
- Information domain values are defined in the following manner:
  - **Number of external inputs (EIs).**
  - **Number of external outputs (EOs).**
  - **Number of external inquiries (EQs).**
  - **Number of internal logical files (ILFs).**
  - **Number of external interface files (EIFs).**
- Steps involved in FP based estimation:
  1. Find the bounded statement of software scope and
  2. Decompose the statement of scope into problem functions that can each be estimated individually.
  3. Estimate the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values

4. Using the estimates derive an FP value that can be tied to past data and used to generate an estimate.
5. Using historical data, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.
6. A three-point or expected value can then be computed.
7. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates.

$$S = ( S_{opt} + 4 S_m + S_{pess} ) / 6$$

8. Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied.

**Example for FP based estimation:**

- Once the information data have been collected, calculate the FP values by associating a complexity value with each count as shown below

| Information Domain Value | Count | | Weighting factor | | | | |
|---|---|---|---|---|---|---|---|
| | | | Simple | Average | Complex | | |
| External Inputs (EIS) | 3 | X | 3 | 4 | 6 | = | 9 |
| External Outputs (EOs) | 2 | X | 4 | 5 | 7 | = | 8 |
| External Inquiries (EQs) | 2 | X | 3 | 4 | 6 | = | 6 |
| Internal Logical Files (ILFs) | 1 | X | 7 | 10 | 15 | = | 7 |
| External Interface Files (EIFs) | 4 | X | 5 | 7 | 10 | = | 20 |
| Count Total | | | | | | | 50 |

- the various adjustment factors to be considered are:

| Factor | | Value |
|---|---|---|
| 1. | Backup and recovery | 4 |
| 2. | Data communications | 2 |
| 3. | Distributed processing | 0 |
| 4. | Performance critical | 4 |
| 5. | Existing operating environment | 3 |
| 6. | On-line data entry | 4 |
| 7. | Input transaction over multiple screens | 5 |
| 8. | ILFs updated online | 3 |
| 9. | Information domain values complex | 5 |
| 10. | Internal processing complex | 5 |
| 11. | Code designed for reuse | 4 |
| 12. | Conversion/installation in design | 3 |
| 13. | Multiple installations | 5 |
| 14. | Application designed for change | 5 |

- To compute function points (FP), the following relationship is used:
  - $FP_{estimated}$ = count-total × [0.65 + 0.01 × Σ ($F_i$)]

where $F_i$ (i = 1 to 14 are value adjustment factors) and count total is the sum of all FP entries obtained

For the given CAD software problem,

- $FP_{estimated}$ = 320 × [0.65 + 0.01 × Σ ($F_i$)] =375
- The organizational average productivity for systems of this type is 6.5 FP/pm.
- Based on a burdened labor rate of $8000 per month, the cost per FP is approximately $1230.
- Based on the FP estimate and the historical productivity data, **the total estimated project cost is $461,000 (375 * 1230) and the estimated effort is 58 person-months (375/6.5)**

## 5.5 COCOMO MODEL

Software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded.

Boehm's definition of organic, semidetached, and embedded systems is:

**Organic:**

A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

**Semidetached:**

A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited

experience on related systems but may be unfamiliar with some aspects of the system being developed.

**Embedded:**

A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

**COCOMO MODEL**

- COCOMO (Constructive Cost Estimation Model) was proposed by Boehm
- According to Boehm, software cost estimation should be done through three stages:
  - Basic COCOMO
  - Intermediate COCOMO and
  - Complete COCOMO.

**Basic COCOMO Model**

- The basic COCOMO model gives an approximate estimate of the project parameters.
- The basic COCOMO estimation model is given by the following expressions:

$$\textbf{Effort} = a_1 \times (\textbf{KLOC})^{a2}\ \textbf{PM}$$

$$T_{dev} = b_1 \times (\textbf{Effort})^{b2}\ \textbf{Months}$$

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 products,

• $T_{dev}$ 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).

- The effort estimation is expressed in units of person-months (PM).

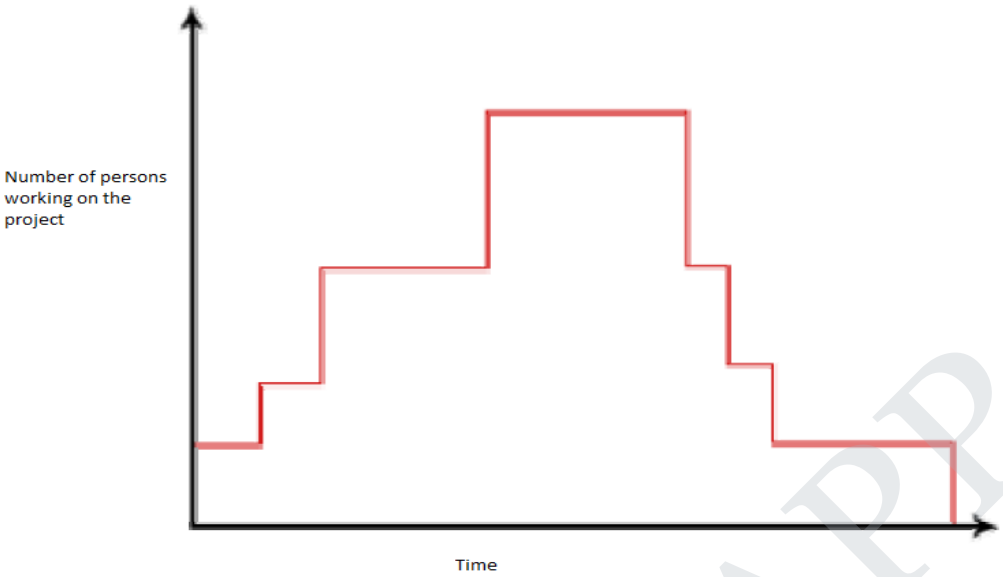



Fig: Person-Month Curve

- According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC.
- The values of $a_1$, $a_2$, $b_1$, $b_2$ for different categories of products (i.e. organic, semidetached, and embedded)are summarized below.

**Estimation of development effort**

- For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

  Organic : **Effort = 2.4(*KLOC*)$^{1.05}$ PM**

  Semi-detached : **Effort = 3.0(*KLOC*)$^{1.12}$ PM**

  Embedded : **Effort = 3.6(*KLOC*)$^{1.20}$ PM**

**Estimation of development time**

- For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

  Organic : **$T_{dev}$ = 2.5(*Effort*)$^{0.38}$ Months**

  Semi-detached : **$T_{dev}$ = 2.5(*Effort*)$^{0.35}$ Months**

  Embedded : **$T_{dev}$ = 2.5(*Effort*)$^{0.32}$ Months**

- Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes.
- Fig. shows a plot of estimated effort versus product size.

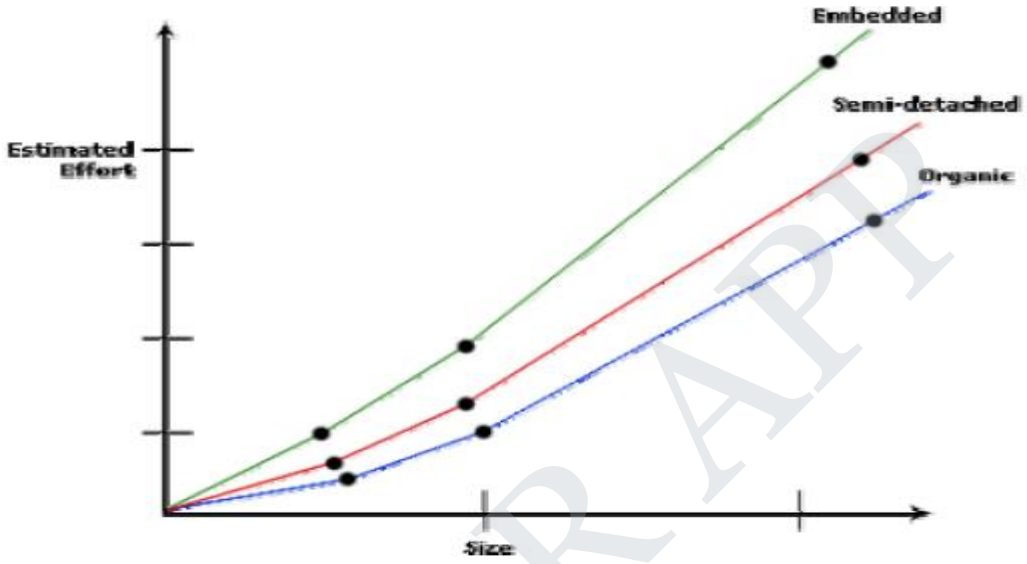



Fig: Effort Vs Product Size

- The effort is super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.
- The development time versus the product size in KLOC is plotted in fig.

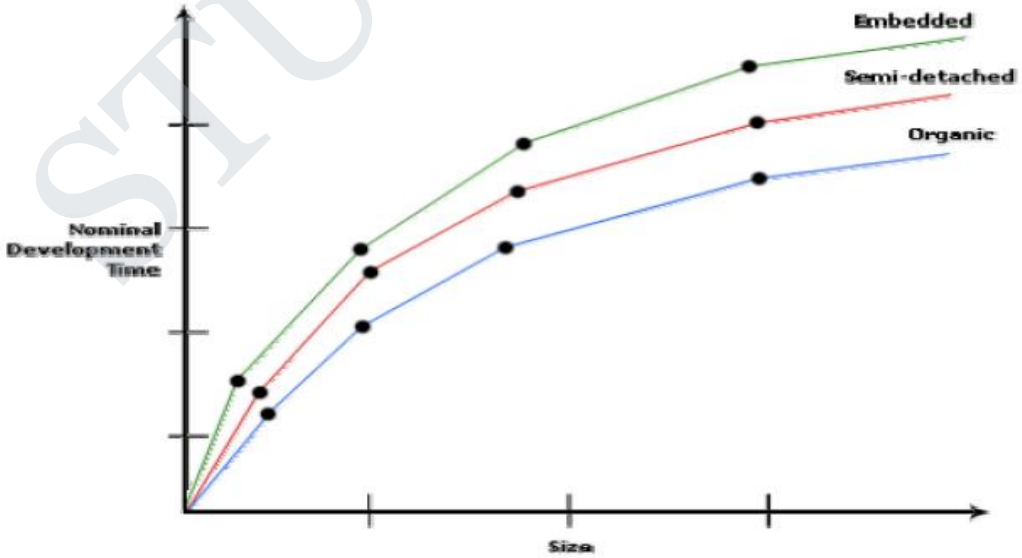


Fig: Development Time Vs Size

- The development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately.
- The development time is roughly the same for all the three categories of products.
- It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate.

**Intermediate COCOMO model**

- The basic COCOMO model assumes that effort and development time are functions of the product size alone.
- But other project parameters also affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account.
- The intermediate COCOMO model recognizes this and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
- In general, the cost drivers 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, 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.

**Complete COCOMO model**

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large

systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics.

- For example, some sub-systems may be considered as organic type, some semidetached, and some embedded, also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on.
- The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems.
- The cost of each subsystem is estimated separately.
- This approach reduces the margin of error in the final estimate.

**Example:**

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05} = 91$ PM**

Nominal development time = **$2.5 \times (91)^{0.38} = 14$ months**

Cost required to develop the product = **14 x 15,000**

= **Rs. 210,000/-**

## 5.6 PROJECT SCHEDULING

- Software project scheduling is nothing but allocating the estimated effort to specific software engineering tasks.
- The schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This schedule identifies all major process framework activities and the product functions to which they are applied.

- As the project gets developed, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks required to accomplish an activity are identified and scheduled.

**Basic Principles**

- **Compartmentalization**

  Decompose the project into a number of manageable activities and tasks.

- **Interdependency**

  The interdependency between each module must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some may be independent.

- **Time allocation**

  Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). Each task must be assigned a start date and a completion date .

- **Effort validation**

  Every project has a defined number of people on the software team. So ensure that no more than the allocated number of people has been scheduled at any given time.

- **Defined responsibilities**

  Every task that is scheduled should be assigned to a specific team member.

- **Defined outcomes**

  Every task that is scheduled should have a defined outcome.Work products are often combined in deliverables.

- **Defined milestones**

  Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

- Each of these principles is applied as the project schedule evolves.

## STEP INVOLVED IN PROJECT SCHEDULING:

1. Establish a meaningful task set.
2. Define a task network.
3. Use scheduling tools to develop a time-line chart.
4. Define schedule tracking mechanisms

**STEP 1: DEFINING A TASK SET FOR THE SOFTWARE PROJECT**

- A task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project.
- To develop a project schedule, a task set must be distributed on the project time line.
- The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work.

Different types of projects:

1. Concept development projects that are initiated to explore some new business concept or application of some new technology.
2. New application development projects that are undertaken as a consequence of a specific customer request.
3. Application enhancement projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
4. Application maintenance projects that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
5. Reengineering projects that are undertaken with the intent of rebuilding an existing system in whole or in part.

Factors influencing the task set to be chosen are:

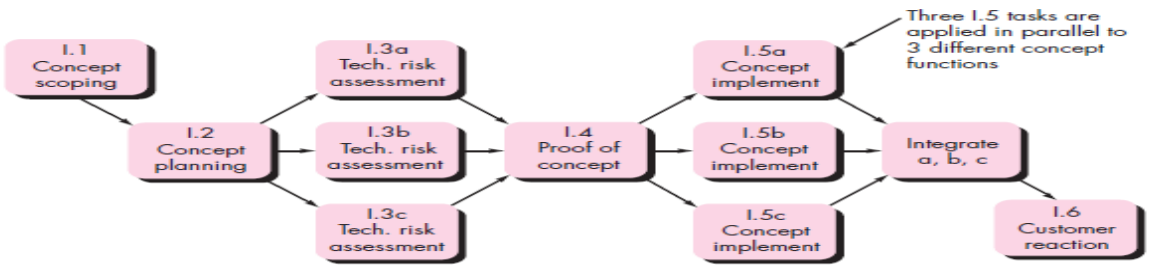
- Size of the project, number of potential users, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, project staff, and reengineering factors etc.
- When taken in combination, these factors provide an indication of the degree of rigor with which the software process should be applied.

**A Task Set Example:**

- Concept development projects are initiated when the potential for some new technology must be explored.
- Concept development projects are approached by applying the following actions:
  1. Concept scoping determines the overall scope of the project.
  2. Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.
  3. Technology risk assessment evaluates the risk associated with the technology to be implemented as part of the project scope.
  4. Proof of concept demonstrates the viability of a new technology in the software context.
  5. Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.
  6. Customer reaction to the concept solicits feedback on a new technology concept and targets specific customer applications.

**STEP 2: DEFINING A TASK NETWORK**

- A *task network*, also called an *activity network*
- It is a graphic representation of the task flow for a project.
- A task network depicts the task length, sequence, concurrency, and dependency
- Task network points out inter-task dependencies in the project
- The critical path
  - A single path leading from start to finish in a task network
  - It contains the sequence of tasks that must be completed on schedule to complete the entire project in time
  - It also determines the minimum duration of the project
- Eg: Task network for a concept development project.




**STEP 3: USE SCHEDULING TOOLS TO DEVELOP A TIME LINE CHART**

- The two project scheduling methods that can be applied to software development are
  - *Program evaluation and review technique* (PERT)
  - C*ritical path method* (CPM)
- Interdependencies among tasks are defined using a task network.
- Tasks, sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions.
- Both PERT and CPM provide quantitative tools that allows to

(1) Determine the critical path—the chain of tasks that determines the duration of the project,

(2) Establish "most likely" time estimates for individual tasks by applying statistical models,

(3) Calculate "boundary times" that define a time "window" for a particular task.

**Time-Line Charts**

- A *time-line chart*, also called a *Gantt chart*.
- A time-line chart can be developed for the entire project or a separate chart can be developed for each project function.
- All project tasks are listed in the left hand column
- The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)
- To the right are columns representing dates on a calendar
- The length of a horizontal bar on the calendar indicates the duration of the task

- When multiple bars occur at the same time interval on the calendar, this implies task concurrency
- A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero
- Eg:

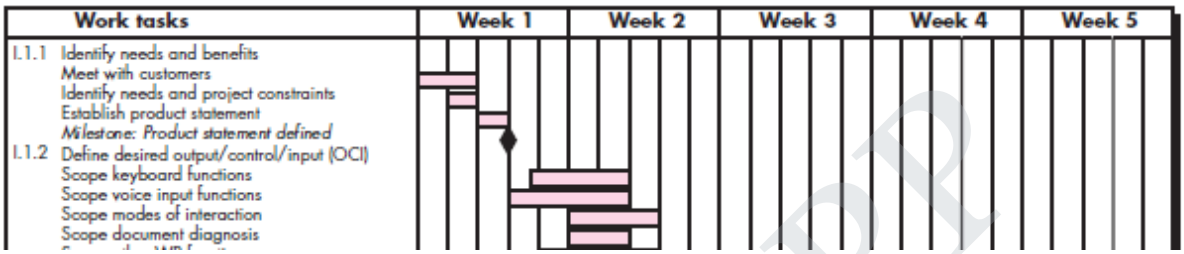



Fig: Time-line chart

- Software project scheduling tools produce *project tables* from the information available in the time – line chart
- Project table is a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information.
- Project tables helps to track progress.
- Eg: Project table

| Work tasks | Planned start | Actual start | Planned complete | Actual complete | Assigned person | Effort allocated | Notes |
|---|---|---|---|---|---|---|---|
| I.1.1 Identify needs and benefits | | | | | | | Scoping will require more effort/time |
| Meet with customers | wk1, d1 | wk1, d1 | wk1, d2 | wk1, d2 | BLS | 2 p-d | |
| Identify needs and project constraints | wk1, d2 | wk1, d2 | wk1, d2 | wk1, d2 | JPP | 1 p-d | |
| Establish product statement | wk1, d3 | wk1, d3 | wk1, d3 | wk1, d3 | BLS/JPP | 1 p-d | |
| *Milestone: Product statement defined* | wk1, d3 | wk1, d3 | wk1, d3 | wk1, d3 | | | |
| I.1.2 Define desired output/control/input (OCI) | | | | | | | |
| Scope keyboard functions | wk1, d4 | wk1, d4 | wk2, d2 | | BLS | 1.5 p-d | |
| Scope voice input functions | wk1, d3 | wk1, d3 | wk2, d2 | | JPP | 2 p-d | |
| Scope modes of interaction | wk2, d1 | | wk2, d3 | | MLL | 1 p-d | |
| Scope document diagnostics | wk2, d1 | | wk2, d2 | | BLS | 1.5 p-d | |
| Scope other WP functions | wk1, d4 | wk1, d4 | wk2, d3 | | JPP | 2 p-d | |
| Document OCI | wk2, d1 | | wk2, d3 | | MLL | 3 p-d | |
| FTR: Review OCI with customer | wk2, d3 | | wk2, d3 | | all | 3 p-d | |
| Revise OCI as required | wk2, d4 | | wk2, d4 | | all | 3 p-d | |
| *Milestone: OCI defined* | wk2, d5 | | wk2, d5 | | | | |
| I.1.3 Define the function/behavior | | | | | | | |

Fig: Project Table

**STEP 4: DEFINE SCHEDULE TRACKING MECHANISMS**

**Tracking the Schedule**

- Qualitative approaches for tracking:

1. Conducting periodic project status meetings in which each team member reports progress and problems
2. Evaluating the results of all reviews conducted throughout the software engineering process
3. Determining whether formal project milestones have been accomplished by the scheduled date
4. Comparing the actual start date to the planned start date for each project task listed in the resource table
5. Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon

- Quantitative approach:
    6. Using earned value analysis to assess progress quantitatively

- Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff.
- If things are going well, control is light. But when problems occur, the project manager must apply tight control to reconcile the problems as quickly as possible. For example:
    - Staff may be redeployed
    - The project schedule may be redefined
- After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.
- When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called **time-boxing.**

**Time-boxing strategy:**

- The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline.
- An incremental software paradigm is applied to the project
- The tasks associated with each increment are "time-boxed" (i.e., given a specific start and stop time) by working backward from the delivery date

- A "box" is put around each task. When a task hits the boundary of its time box, work stops and the next task begins.
- This approach succeeds based on the premise that when the time-box boundary is encountered, it is likely that 90% of the work is complete
- The remaining 10% of the work can be
  - Delayed until the next increment
  - Completed later if required
- Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

## 5.7 EARNED VALUE ANALYSIS

- Earned value analysis is a measure of progress by assessing the percent of completeness for a project
- It gives accurate and reliable readings of performance very early into a project
- It provides a common value scale (i.e., time) for every project task, regardless of the type of work being performed
- The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total
- To determine the earned value, the following steps are performed:
  1. The **budgeted cost of work scheduled (BCWS)** is determined for each work task represented in the schedule. The value of BCWS is the sum of the BCWS*i* values for all work tasks that should have been completed by that point in time on the project schedule.
  2. The BCWS values for all work tasks are summed to derive the **budget at completion (BAC).**

     $BAC = \sum (BCWS_k)$ for all tasks k

  3. Next, the value for **budgeted cost of work performed (BCWP)** is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.
- **BCWS** represents the budget of the activities that were planned to be completed and **BCWP** represents the budget of the activities that actually were completed.

- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

**Schedule performance index, SPI = BCWP / BCWS**

**Schedule variance, SV = BCWP - BCWS**

where,

→ SPI is an indication of the efficiency with which the project is utilizing scheduled resources.

→SPI value close to 1.0 indicates efficient execution of the project schedule.

→SV is simply an absolute indication of variance from the planned schedule.

- Percentage scheduled for completion provides an indication of the percentage of work that should have been completed by time *t*.

**Percent scheduled for completion = BCWS / BAC**

- Percent complete provides a quantitative indication of the percent of completeness of the project at a given point in time *t*.

**Percent complete = BCWP / BAC**

- The actual cost of work performed (ACWP) is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule.
- It is then possible to compute

**Cost performance index, CPI= BCWP /ACWP**

**Cost variance, CV =BCWP - ACWP**

→ A CPI value close to 1.0 provides a strong indication that the project is within its defined budget.

→CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

- Earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables to take corrective action before a project crisis develops.

## 5.8 RISK MANAGEMENT

- Risk analysis and management are the actions which help to understand and manage uncertainty.
- A risk is a potential problem which may or may not happen.
- To manage risk, first identify the risk, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur

### RISK CATEGORIZATION

- The two broad categories of risks are:
  - **Product specific risk**

    - Risks that can be identified only with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - **Generic risk** - Risks that are a potential threat to every software project

### Categories of Product specific risk:

- **Project risks**
  - Project risks threaten the project plan
  - If they occur, then the project schedule will slip and the costs will increase
- **Technical risks**
  - This threaten the quality and timeliness of the software to be produced
  - If they occur, implementation becomes difficult or impossible
- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product
  - Sub-categories of Business risks
    - Market risk – building an excellent product or system that no one really wants

- Strategic risk – building a product that no longer fits into the overall business strategy for the company
- Sales risk – building a product that the sales force doesn't understand how to sell
- Management risk – losing the support of senior management due to a change in focus or a change in people
- Budget risk – losing budgetary or personnel commitment

**Categories of Generic risks:**

- **Known risks**
  - Known risks are those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- **Predictable risks**
  - Predictable risks are extrapolated from past project experience (e.g., past turnover)
- **Unpredictable risks**
  - Those risks that are extremely difficult to identify in advance are referred as unpredictable risks.

**Reactive vs. Proactive Risk Strategies**

- Reactive risk strategies
  - Nothing is done about risks until something goes wrong
    - The team then flies into action in an attempt to correct the problem rapidly (also referred as fire fighting mode)
  - Crisis management is the choice of management techniques
- Proactive risk strategies
  - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner.

**Seven Principles of Risk Management**

- **Maintain a global perspective**
  - View software risks within the context of a system and the business problem that is intended to solve
- **Take a forward-looking view**
  - Think about risks that may arise in the future; establish contingency plans
- **Encourage open communication**
  - Encourage all stakeholders and users to point out risks at any time
- **Integrate risk management**
  - Integrate the consideration of risk into the software process
- **Emphasize a continuous process of risk management**
  - Modify identified risks as more becomes known and add new risks as better insight is achieved
- **Develop a shared product vision**
  - A shared vision by all stakeholders facilitates better risk identification and assessment
- **Encourage teamwork when managing risk**
  - Pool the skills and experience of all stakeholders when conducting risk management activities

## STEPS INVOLVED IN RISK MANAGEMENT

1. Identify possible risks; recognize what can go wrong
2. Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3. Rank the risks by probability and impact
4. Develop a contingency plan to manage those risks having high probability and high impact

## STEP 1: RISK IDENTIFICATION

- Risk identification is a systematic attempt to specify threats to the project plan

- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- To identify the risk create, Risk item checklist.

**Risk Item Checklist**

- This focuses on known and predictable risks in specific subcategories
- The check list can be organized in several ways
    1. A list of characteristics relevant to each risk subcategory
    2. Questionnaire that leads to an estimate on the impact of each risk
    3. A list containing a set of risk component and drivers and their probability of occurrence

**1. Known and Predictable Risk Categories**

- Product size – risks associated with overall size of the software to be built
- Business impact – risks associated with constraints imposed by management or the marketplace
- Customer characteristics – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- Process definition – risks associated with the degree to which the software process has been defined and is followed
- Development environment – risks associated with availability and quality of the tools to be used to build the project
- Technology to be built – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- Staff size and experience – risks associated with overall technical and project experience of the software engineers who will do the work

**2. Sample Questionnaire on Project Risk**

1) Are requirements fully understood by the software engineering team and its customers?
2) Have customers been involved fully in the definition of requirements?
3) Is the project scope stable?
4) Does the software engineering team have the right mix of skills?
5) Are project requirements stable?

6) Does the project team have experience with the technology to be implemented?
7) Is the number of people on the project team adequate to do the job?

**3. Risk Components and Drivers**

- The project manager identifies the risk drivers that affect the following risk components
  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of **four impact levels**
  - Negligible
  - Marginal
  - Critical
  - Catastrophic
- Risk drivers can be assessed as **impossible, improbable, probable, and frequent**

## STEP 2 & 3: RISK ANALYSIS & RANKING:

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real
  - The consequence of the problems associated with the risk

Steps involved in Risk Estimation:

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Assess the overall accuracy of the risk projection so that there will be no misunderstandings

Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk Summary – short description of the risk
  - Risk Category – one of seven risk categories (Problem size, business impact etc )
  - Probability – estimation of risk occurrence based on group input
  - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
  - RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

| Risks | Category | Probability | Impact | RMMM |
|---|---|---|---|---|
| Size estimate may be significantly low | PS | 60% | 2 | |
| Larger number of users than planned | PS | 30% | 3 | |
| Less reuse than planned | PS | 70% | 2 | |
| End-users resist system | BU | 40% | 3 | |
| Delivery deadline will be tightened | BU | 50% | 2 | |
| Funding will be lost | CU | 40% | 1 | |
| Customer will change requirements | PS | 80% | 2 | |
| Technology will not meet expectations | TE | 30% | 1 | |
| Lack of training on tools | DE | 80% | 3 | |
| Staff inexperienced | ST | 30% | 2 | |
| Staff turnover will be high | ST | 60% | 2 | |
| Σ | | | | |
| Σ | | | | |
| Σ | | | | |

Fig: Sample Risk Table

- After identifying the category, probability of occurrence and impacts sort the rows by probability and impact in descending order
- Draw a **horizontal cutoff line** in the table that indicates the risks that will be given further attention
- Risks that fall below the line are reevaluated to accomplish second-order prioritization.
- Probability has a distinct influence on management concern.
- A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

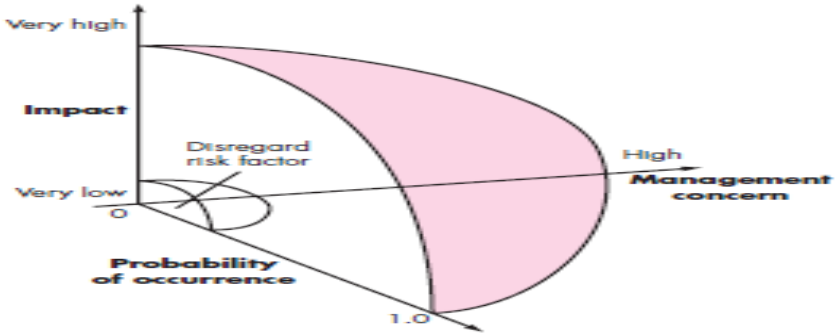



Fig: Risk and Management concern

**Assessing Risk Impact**

- Three factors affect the consequences that are likely if a risk does occur
  - **Its nature** – This indicates the problems that are likely if the risk occurs
  - **Its scope** – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - **Its timing** – This considers when and for how long the impact will be felt
- The overall risk exposure formula is **RE = P x C**
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

**STEP 4: RISK MITIGATION, MONITORING, AND MANAGEMENT**

- An effective strategy for dealing with risk must consider three issues
  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning
- **Mitigation** : How can we avoid the risk?
- **Monitoring** : What factors can we track that will enable us to determine if the risk is becoming more or less likely?

- **Management** : what contingency plans do we have if the risk becomes a reality?

**Example:**

**Risk: Technology Does Not Meet Specifications**

- **Mitigation**

  In order to prevent this from happening, meetings (formal and informal) will be held with the customer on a routine business. This insures that the product we are producing, and the specifications of the customer are equivalent.

- **Monitoring**

  The meetings with the customer should ensure that the customer and our organization understand each other and the requirements for the product.

- **Management**

  Should the development team come to the realization that their idea of the product specifications differs from those of the customer, the customer should be immediately notified and whatever steps necessary to rectify this problem should be done. Preferably a meeting should be held between the development team and the customer to discuss at length this issue.

Example: Risk of high staff turnover

- Strategy for Reducing Staff Turnover
  - Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
  - Mitigate those causes that are under our control before the project starts
  - Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
  - Organize project teams so that information about each development activity is widely dispersed
  - Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
  - Conduct peer reviews of all work (so that more than one person is "up to speed")
  - Assign a backup staff member for every critical technologist

- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality
- RMMM steps incur additional project cost
    - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself

Risks can occur after the software has been delivered to the user

- Software safety and hazard analysis
    - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
    - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

**The RMMM Plan**

- The RMMM plan may be a part of the software development plan or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
    - Risk mitigation is a problem avoidance activity
    - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
    - To assess whether predicted risks do, in fact, occur
    - To ensure that risk aversion steps defined for the risk are being properly applied
    - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

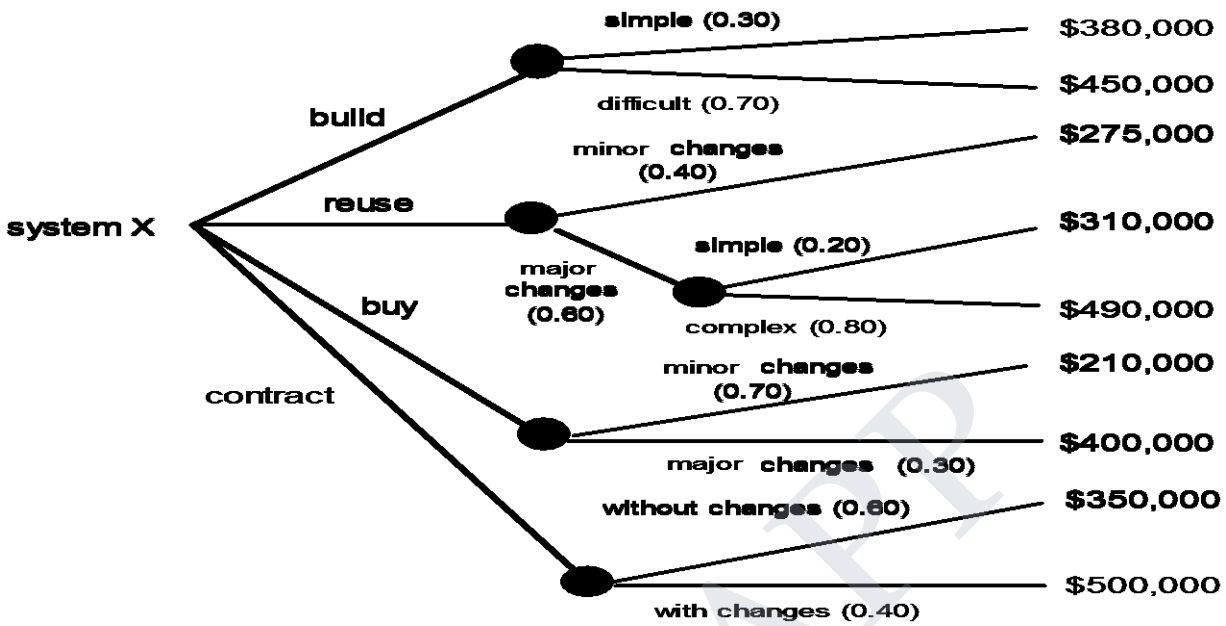
**5.9 MAKE/BUY DECISION**

- It is often more cost effective to acquire rather than develop software.
- Managers have many acquisition options.
  - Software may be purchased off the shelf
  - "Full-experience" or "partial-experience" software components may be acquired and integrated to meet specific needs
  - Software may be custom built by an outside contractor to meet the purchaser's specifications
- The make/buy decision can be made based on the following conditions
  - Will the software product be available sooner than internally developed software?
  - Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
  - Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support?

**CREATING A DECISION TREE**

- Make/Buy decision can be made using statistical techniques such as decision tree analysis.
- For example, Figure 26.8 depicts a decision tree for a software- based system X. In this case, the software engineering organization can (1) build system X from scratch, (2) reuse existing partial-experience components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.
- If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. The project planner estimates that a difficult development effort will cost $450,000. A "simple" development effort is estimated to cost $380,000.

**Expected cost = ∑ (path probability) x (estimated path cost)**




- For example, the expected cost to build is:

**Expected cost = 0.30($380K)+0.70($450K)**

**= $429 K**

**Expected cost (reuse) = $382K**

**Expected cost (buy) = $267K**

**Expected cost (contract) = $410K**

**OUTSOURCING**

- The decision to outsource can be either strategic or tactical.
- At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others.
- At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.
- The outsourcing decision is often a financial one.

**Advantages:**

- Cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them.

**Disadvantage:**

- A company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

## 5.10 COCOMO II MODEL

- COCOMO stands for COnstructive COst MOdel
- COCOMO II is actually a hierarchy of three estimation models

**COCOMO Models**

- **Application composition model** - Used during the early stages of software engineering when the following are important
  - Prototyping of user interfaces
  - Consideration of software and system interaction
  - Assessment of performance
  - Evaluation of technology maturity
- **Early design stage model** – Used once requirements have been stabilized and basic software architecture has been established
- **Post-architecture stage model** – Used during the construction of the software

- This requires sizing information and accepts it in three forms:
  - object points,
  - function points
  - lines of source code
- Object point is an indirect software measure that is computed using counts of the number of
  (1) Screens
  (2) Reports, and
  (3) Components likely to be required to build the application.
- Each object instance is classified into one of three complexity levels as given below

| Object type | Complexity weight | | |
|---|---|---|---|
| | Simple | Medium | Difficult |
| Screen | 1 | 2 | 3 |
| Report | 2 | 5 | 8 |
| 3GL component | | | 10 |

- Complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.
- The object point count is then determined by multiplying the number of object instances by the weighting factor and summing to obtain a **total object point count**.
- The percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$NOP = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

Where NOP is defined as new object points.

- To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived.

$$PROD = \frac{NOP}{\text{person-month}}$$

- Productivity rate is based on different levels of developer's experience and environment maturity

| Developer's experience/capability | Very low | Low | Nominal | High | Very high |
|---|---|---|---|---|---|
| Environment maturity/capability | Very low | Low | Nominal | High | Very high |
| PROD | 4 | 7 | 13 | 25 | 50 |

- Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{NOP}{PROD}$$

- In more advanced COCOMO II models, 12 a variety of scale factors, cost drivers, and adjustment procedures are required.

**COCOMO Cost Drivers**

- **Personnel Factors**
  - Programming language experience
  - Personnel capability and experience
  - Language and tool experience etc
- **Product Factors**
  - Database size
  - Required reusability
  - Product reliability and complexity etc
- **Project Factors**
  - Use of software tools
  - Required development schedule
  - Multi-site development etc