Chapter 1

SOFTWARE ENGINEERING, THE SOFTWARE PROCESS

1.1

SOFTWARE:

"SET OF instructions (computer programs) that when executed provide desired features, function, and performance"

Why we use software:

- Software delivers the most important product of our time—**information.**
- It transforms personal data so that the data can

be more useful in a local context;

- It **manages** business information to enhance competitiveness;
- It provides a **gateway to worldwide information networks** (e.g., the

Internet), and provides the means for acquiring information in all of its form.

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

Nature of Software

Characteristics of software

- Software is developed or engineered; it is not manufactured
- Software doesn't "wear out."

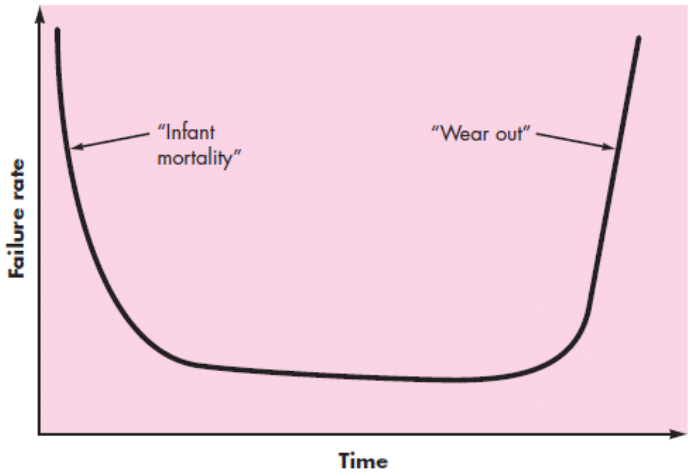


Fig- "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life hardware wear out

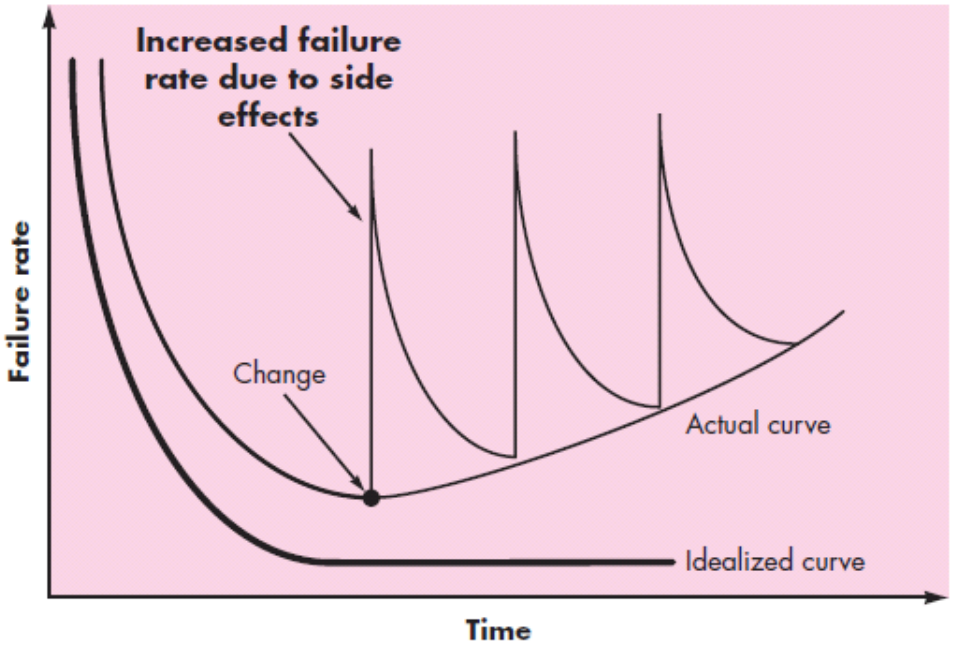


Figure shows During its life,2 software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change

Software Application Domain

- **System software**—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities)

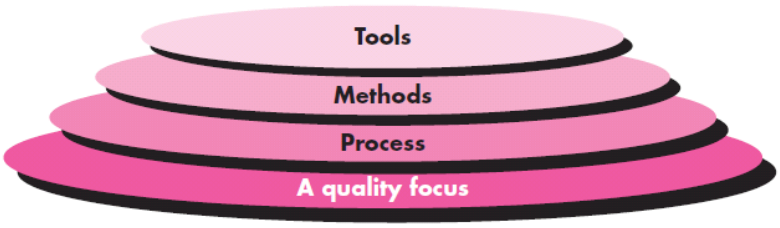
- **Application software**—stand-alone programs that solve a specific business need e.g., point-of-sale transaction processing, real-time manufacturing process control
- **Engineering/scientific software**—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
- **Web applications**—called "WebApps"- this network-centric software category spans a wide array of applications.
- **Artificial intelligence software**—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis

Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases much older.

Software Engineering

Def: "The application of a systematic, disciplined, quantifiable approach to the development, operate, and maintenance of software; that is, the software engineering"



Software engineering is a layered technology. Referring to Figure any engineering approach (including software engineering) must rest on an organizational commitment to quality.

- Total **quality** management, Six Sigma, effective approaches to software engineering.
- The foundation for software engineering is the **process layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework
- Software engineering **methods** provide the technical how-to's for building software. Methods encompass a broad array of tasks that

include communication, requirements analysis, design modeling, program construction, testing, and support.

- Software engineering **tools** provide automated or semiautomated support. When tools are integrated so that information created by one tool can be used by another

Software Process:

A **generic process framework for software engineering encompasses five activities**:

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and otherstakeholders11 the intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity centesimal" that helps guide the team as it makes the journey. The map—called a 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**. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
- **Construction**. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
- **Deployment**. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical **umbrella activities** include:

- **Software project tracking and control**—allows the software team 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 of the project or the quality of the product. Software quality assurance—defines and conducts the activities required to ensure software quality.
- **Technical reviews**—assesses software engineering work products in an effort 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; can be used in conjunction with all other framework and umbrella activities.
- **Software configuration management**—manages the effects of change throughout the software process.
- **Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
- **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Software Engineering Practice

Essence of Practice

Understand the problem. (communication and analysis).
• Identify **stakeholders**
• Solve **data, functions, and features are required**
• **Represent in smaller problems**
• Represent problem **graphically**

Plan the solution. (modeling and software design).
• Understand patterns
• Has a similar problem been solved so can be **reusable**
• Define the **subproblems**

Carry out the plan. (code generation).
• Design **Source code** according to model
• Prepare **solution in part wise** and correct

Examine the result

• Test each component part of the solution
• Solution produce results that conform to the data, functions, and features that are required

General Principals

The First Principle: The Reason It All Exists

A software system exists for one reason: **to provide value to its users**.

The Second Principle:(Keep It Simple, Stupid!)

Software design is not a difficult process. There are many factors to considering any design effort. All design should be as simple as possible.

The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project.

The Fourth Principle: What You Produce, Others Will Consume

Seldom implement knowing someone else will have to understand what you are doing.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. In today's computing environments, to reuse of an entire system.

The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort. The reuse of code and designs have been proclaimed as a major benefit of using technologies. Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

The Seventh principle: Think!

Placing clear, complete thought before action almost always produces better results.

Software Crisis

The word crisis has definition: "the turning point in the course of a disease, when it becomes clear whether the patient will live or die."

This give us a clue about the real nature of the problems that have plagued software development.

Somewhere crisis is also deal with the failures.

Regardless the set of problems that are encountered in the development of computer software is not limited to software that "doesn't function properly.

Rather, it encompasses problems associated with how we develop software

How we support a growing volume of existing software, how we can expect to keep pace with a growing demand for more software.

And yet, things would be much better if we could find and broadly apply a cure

Software Myths

Management myths.

Managers with software responsibility, most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

- **Myth:** We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

  Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

- **Myth:** If we get behind schedule, we can add more programmers and catch up.

  Reality: Software development is not a mechanistic process like manufacturing. "adding people to a late software project makes it later." However, as new people are added, people who we're working must spend time educating the newcomers

- **Myth:** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

  Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths. A customer who requests computer software

- **Myth:** A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

  Reality: Although a comprehensive and stable statement of requirements is not always possible. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

- **Myth:** Software requirements continually change, but change can be easily accommodated because software is flexible.

  Reality: It is true that software requirements change, but the impact of change varies with the time changes are requested early the cost impact is relatively small than the cost impact of change at commitment level

Practitioner's myths. Myths that are still believed by software practitioners

- **Myth:** Once we write the program and get it to work, our job is done.

  Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate 60 and 80% of all effort expended on software will be expended after it is delivered to the customer for the first time.

- **Myth:** Until I get the program "running" I have no way of assessing its quality.

  Reality: the technical review. Software reviews are a "quality filter" that have been found to be more

- **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 includes many elements. A variety of work products provide a foundation for successful engineering

- **Myth:** Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

  Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework.

1.2

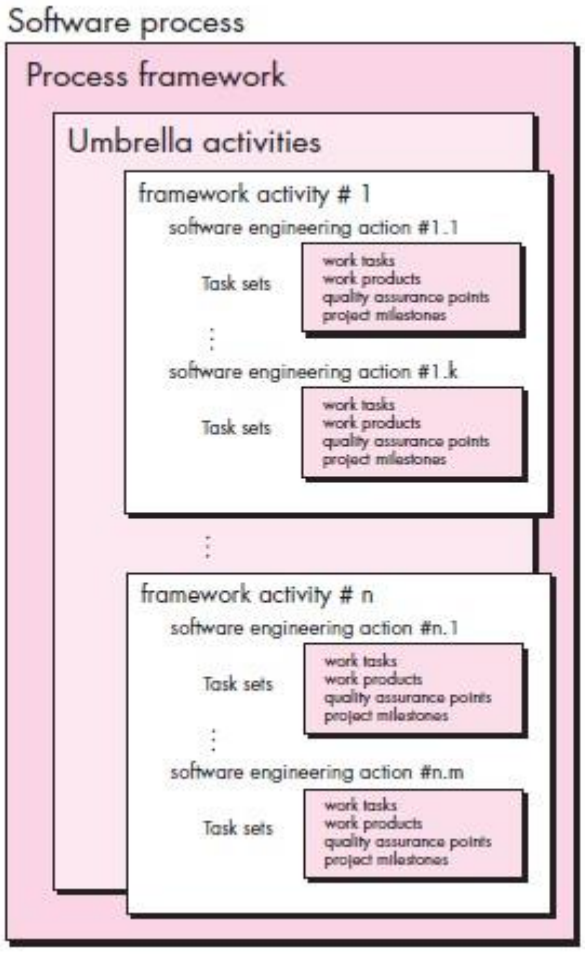
PROCESS MODELS

Generic Process Model

The software process is represented schematically each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work

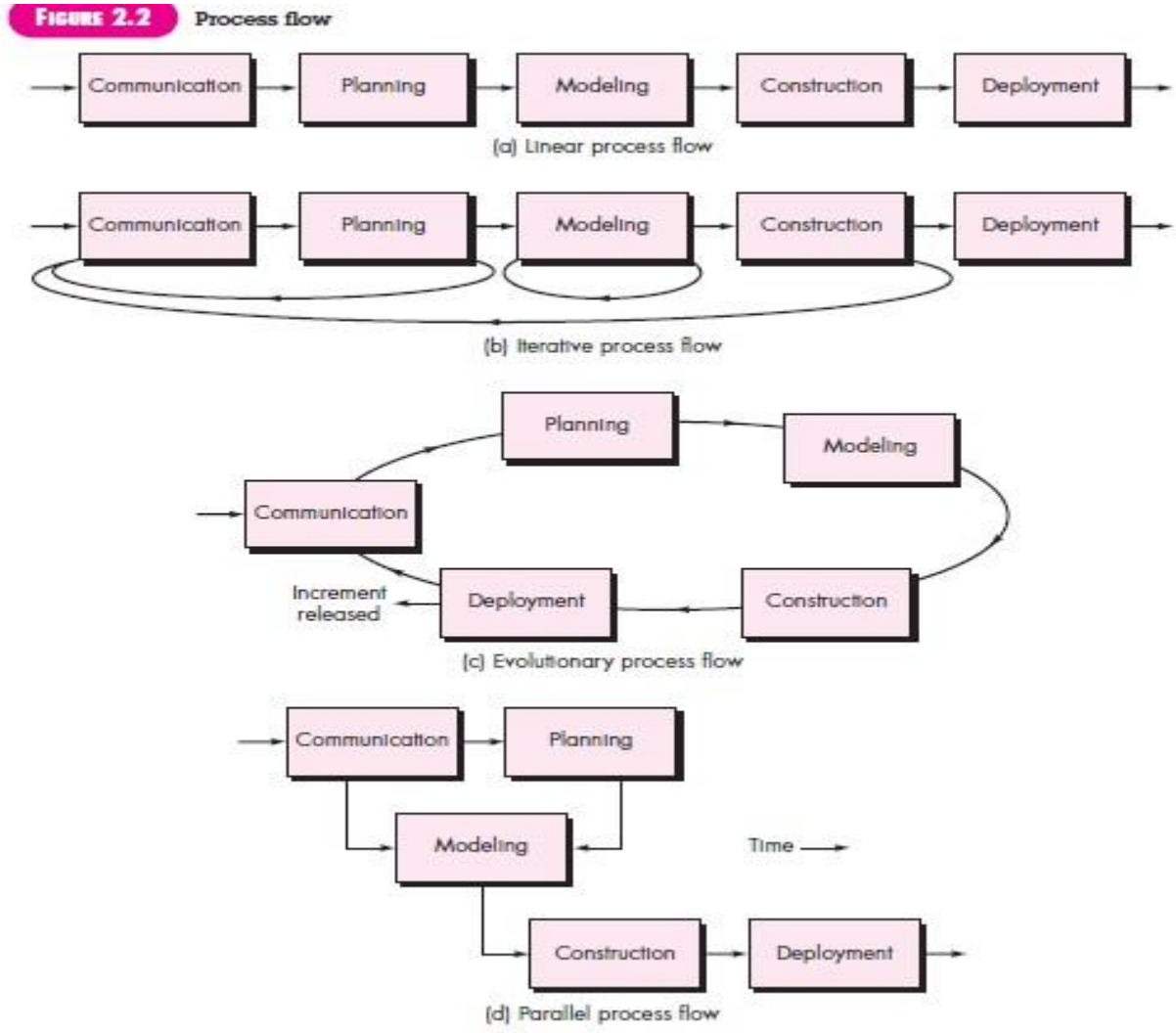
Defining a Framework Activity

a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.



Defining a Framework Activity

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 who are sponsoring the project



Identifying a Task Set

Select a task set that best accommodates the needs of the project and the characteristics of team. This implies that a software engineering action can be adapted to the specific needs of the software project

Process Patterns

1. Stage pattern—defines a problem associated with a framework activity for the process. Ex: **Establishing Communication.**

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **Requirements Gathering** is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is **iterative** in nature. Ex: **Spiral Model**

Process Assessment and Improvement

A number of different approaches to software process assessment and improvement have been proposed

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)**— provides a five-step process assessment model that incorporates five phases: **initiating, diagnosing, establishing, acting, and learning.**
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**— provides a diagnostic technique for assessing the relative maturity of a software organization.
- **SPICE (ISO/IEC15504)** — The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].
- **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies

The Waterfall Model

There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion.

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

A variation in the representation of the waterfall model is called the V-model. Represented in Figure 2.4, the V-model [Buc99] depicts the relationship of quality's

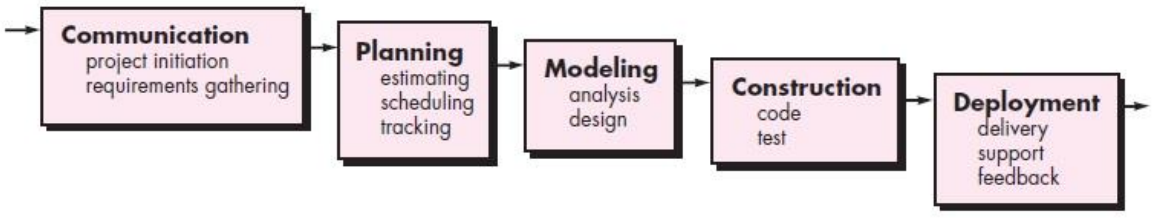


Fig: Waterfall model

Incremental Process Models

The incremental model combines elements of linear and parallel process flows the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software.

Example: word-processing software developed using the incremental paradigm

- **1st** Increment: Deliver basic file management, editing, and document production functions
- 2nd Increment: more sophisticated editing and document production capabilities
- 3rd Increment: spelling and grammar checking;
- 4th Increment: Advanced page layout capability.

When an incremental model is used, the first increment is often a core product. The core product is used by the customer plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer
This process is repeated following the delivery of each increment, until the complete product is produced.
The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product,

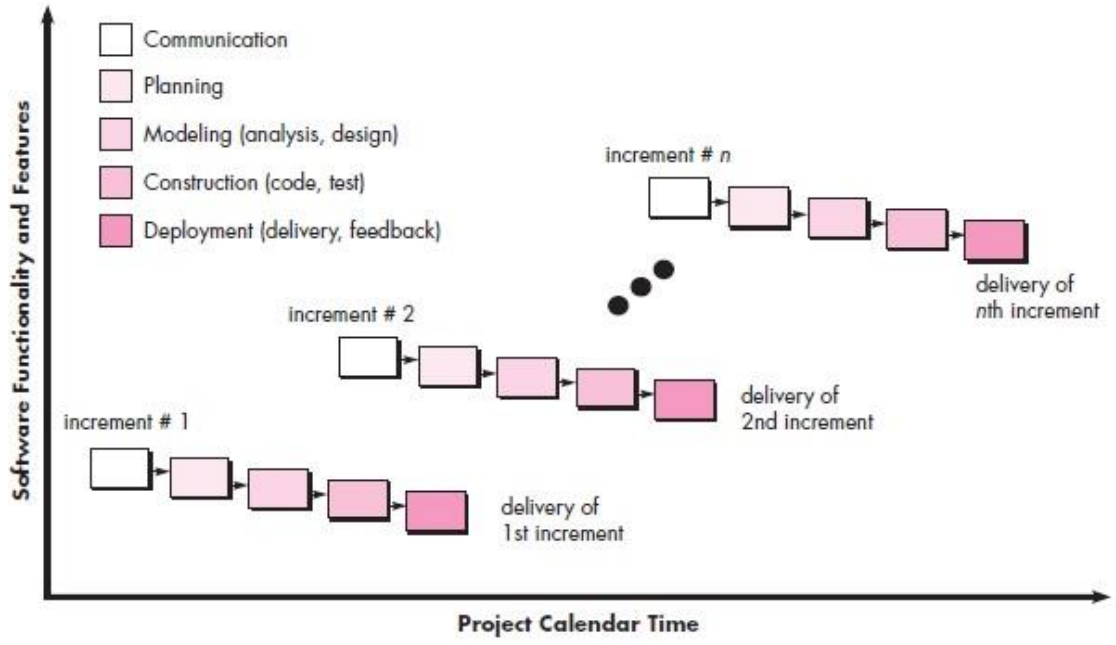
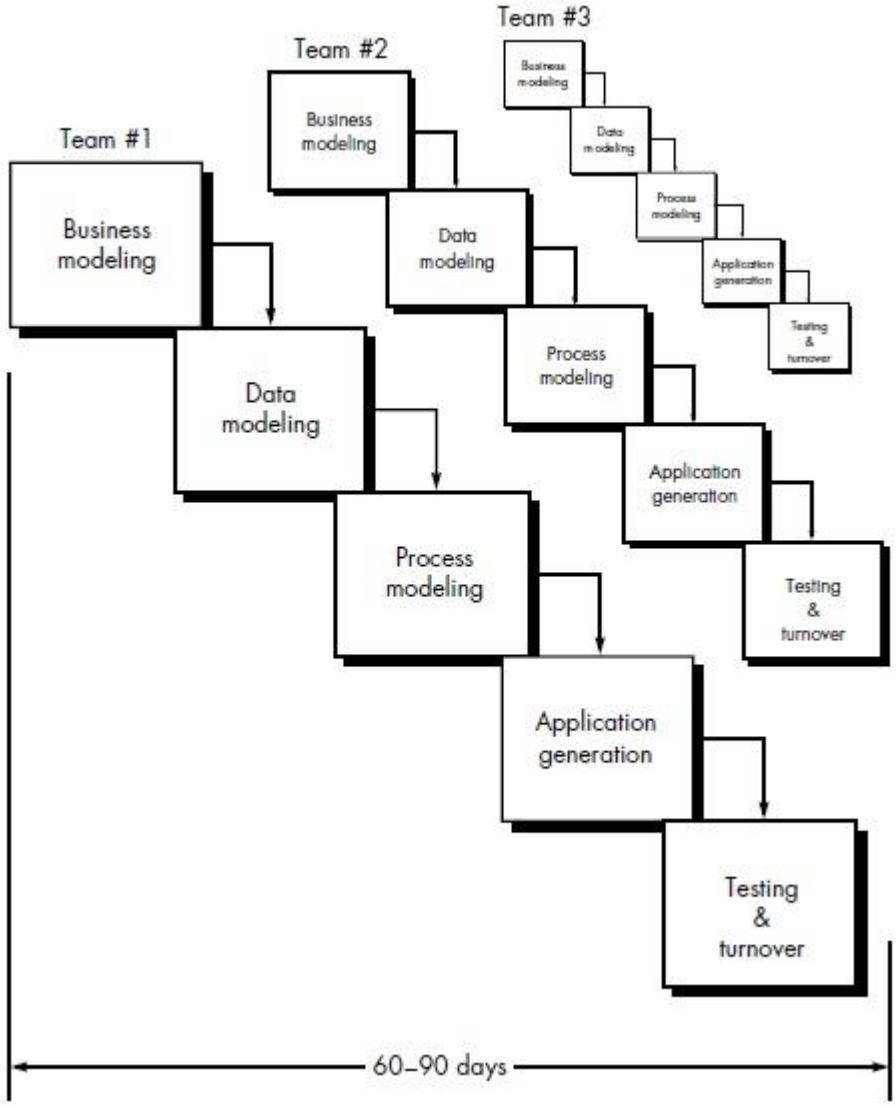


Fig: Incremental Process Model

RAD Model (Rapid Application Development)



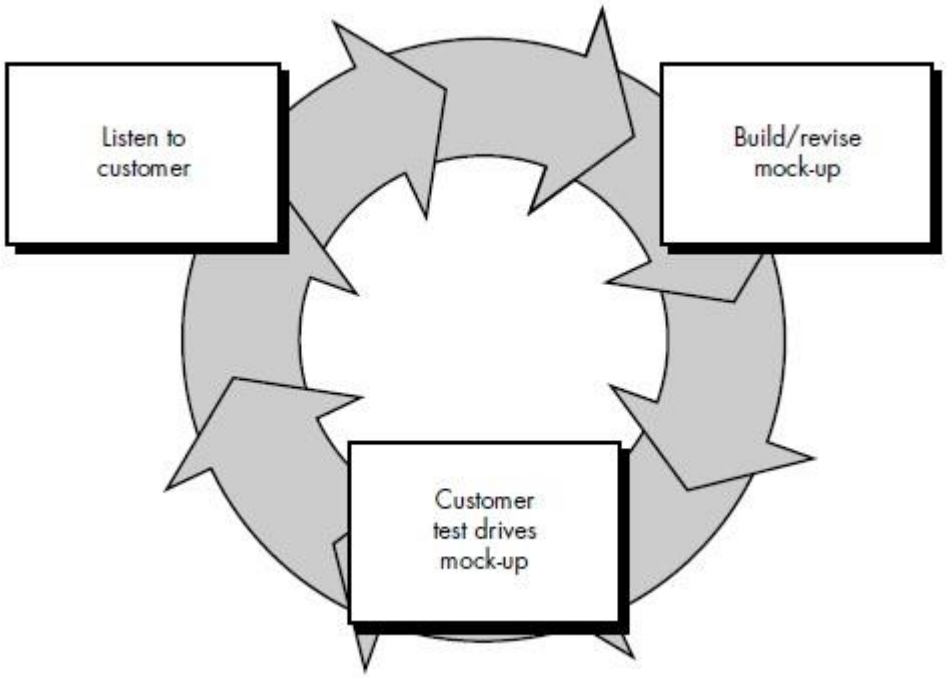
- Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle.
- The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction.
- RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days)

Business modeling. The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?
Data modeling. Set of data objects that are needed to support the business. The attributes of each object are identified and the relationships between these objects defined.

Process modeling. The Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.
Application generation. AD assumes the use of fourth generation techniques RAD process works to reuse existing program components. Automated tools are used to facilitate construction of the software.
Testing and turnover. Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time.

Prototype model

- The prototyping paradigm assists you and other stakeholders/customer to better understand what is to be built when requirements are fuzzy
- The quick design leads to the construction of a prototype.
- The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
- Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately.

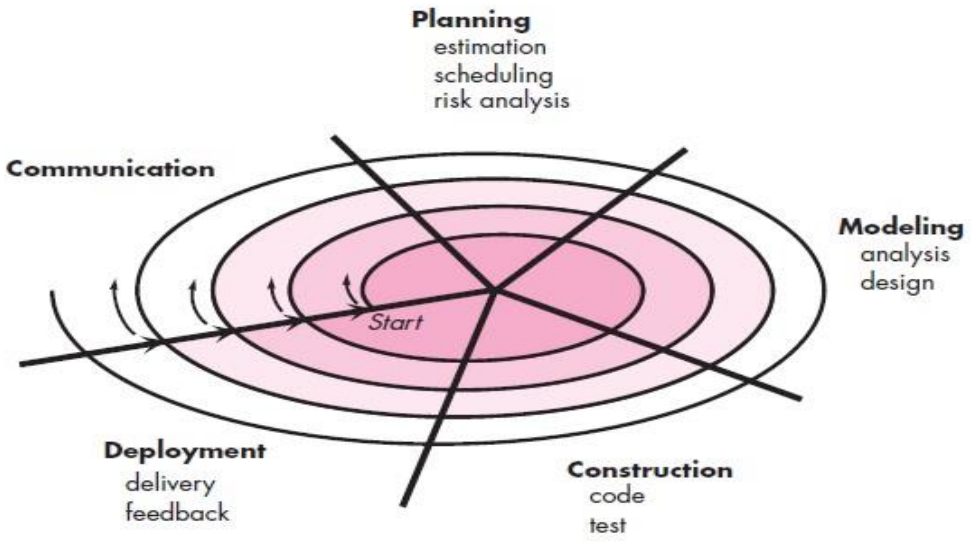


- The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user

- The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed.
- Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done
- Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built

Spiral Model

- The spiral model, originally proposed by Boehm is an evolutionary software.
- It provides the potential for rapid development of incremental versions of the software.
- Using the spiral model, software is developed in a series of incremental releases.
- During early iterations, the incremental release might be a paper model or prototype.
- During later iterations, increasingly more complete versions of the engineered system are produced.
- A spiral model is divided into a number of framework activities, also called task
- The software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center
- first circuit around the spiral might represent a “concept development project” that starts at the core of the spiral and continues for multiple iterations



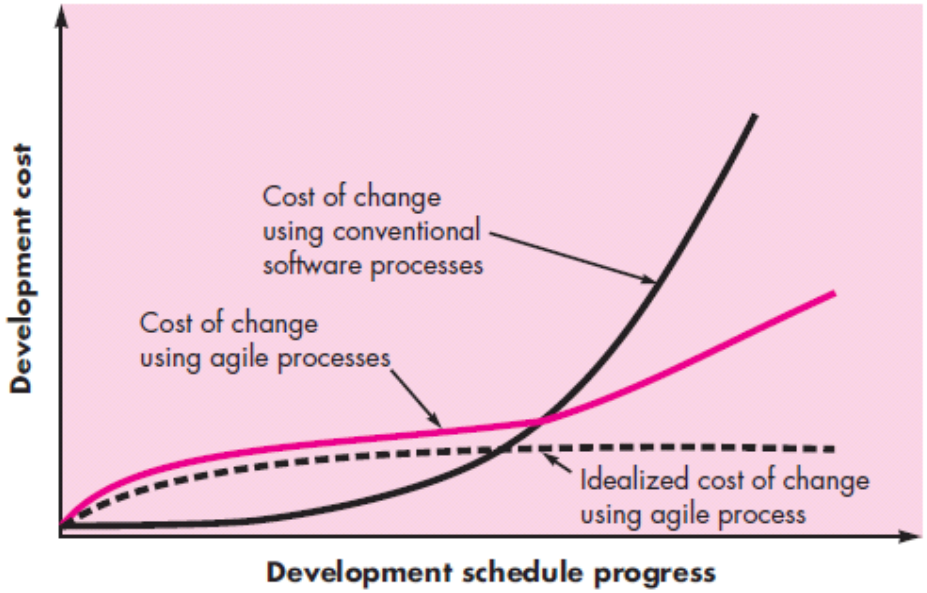
1.3

Agile Development

What is Agility...?

- But agility is effective response to change.
- It encourages team structures and attitudes that make communication among team members, between technologists and business people, between software engineers and their managers more facile.
- It emphasizes rapid delivery of operational software
- It adopts the customer as a part of the development team
- Agility can be applied to any software process emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type

Agility and cost of change



- It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project).
- Costs escalate quickly, and the time and cost required to ensure that the change is made.
- A well-designed agile process “flattens” the cost of change curve
- a software team to accommodate changes late in a software project without cost and time impact.

- agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices(module) the cost of making a change is attenuated.
- significant reduction in the cost of change can be achieved

What is an Agile Process...?

- 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
- It is difficult to predict how much design is necessary before construction is used to prove the design.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.
- How do we create a process that can manage unpredictability? in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be adaptable.
- An agile software process must adapt incrementally. To accomplish incremental adaptation,
- An agile team requires customer feedback (so that the appropriate adaptations can be made). Hence, an incremental development strategy should be effective.
- This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team,

Agility Principles

The agility principles for those who want to achieve agility:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done is essential.
- The best architectures, requirements, and designs emerge from self–organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Human Factors

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members?

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with **ambiguity** and will continually be buffeted by change. In some cases, the team must accept the

fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what a "**jelled" team**. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

Self-organization. In the context of agile development, self-organization implies three things:

(1) the agile team organizes itself for the work to be done,

(2) the team organizes the process to best accommodate its local environment,

(3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

1.4

Project Management Concepts

PEOPLE

Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section we examine the players who participate in the software process and the manner in which they are organized to perform effective software engineering

The Players

1. Senior managers who define the business issues that often have significant
influence on the project.

2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work.

3. Practitioners who deliver the technical skills that are necessary to engineer
a product or application.

4. Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.

5. End-users interact with the software once it is released for production use.

Team Leaders

Leadership includes:

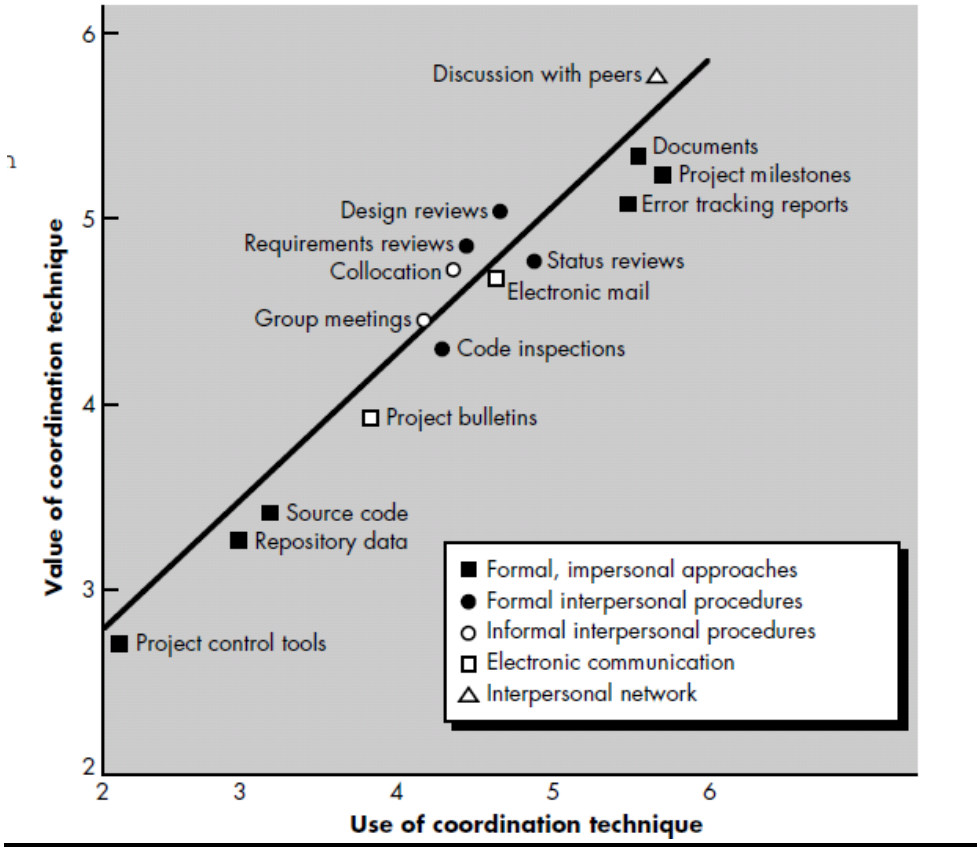
- **Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.
- **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

- **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically

To achieve a high-performance team:
• Team members must have trust in one another.
• The distribution of skills must be appropriate to the problem.

Coordination and Communication Issues

- Formal, impersonal approaches
- Formal, interpersonal procedures
- Informal, interpersonal procedures
- Electronic communication
- Interpersonal networking



THE PRODUCT

Product is the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded

Software Scope

- **Context.** How does the software to be built fit into a larger system?
- **Information objectives.** What customer-visible data objects are produced as output from the software What data objects are required for input
- **Function and performance.** What function does the software perform to
  transform after giving input data into output.
  Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be

Problem Decomposition

Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis

Rather, decomposition is applied in two major areas:
(1) the functionality that must be delivered
(2) the process that will be used to deliver it.

Human beings tend to apply a divide and conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable.

PROCESS:

The generic phases that characterize the software process—definition, development, and support—are applicable to all software

• the linear sequential model

• the prototyping models

• the RAD model

• the incremental model

• the spiral model

• the WINWIN spiral model

• the component-based development model

• the concurrent development models

• the formal methods model

• the fourth-generation techniques model

Melding the Product and the Process

• Customer communication—tasks required to establish effective requirements
elicitation between developer and customer.

• Planning—tasks required to define resources, timelines, and other project related information.

• Risk analysis—tasks required to assess both technical and management risks.

• Engineering—tasks required to build one or more representations of the application.

• Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

• Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

Process Decomposition

Once the process model has been chosen, the common process framework (CPF) is adapted to it. In every case, communication, planning, risk analysis, engineering, construction and release, customer evaluation—can be fitted to the paradigm.

It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models.

Example:

Following work tasks requires for the customer communication activity

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

THE PROJECT

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right.

1. Start on the right foot. This is accomplished by working hard (very hard)

- to **understand the problem** that is to be solved
- set **realistic objects** and expectations for everyone who will be involved in the project.
- building the **right team**
- giving the **team the autonomy, authority**, and technology needed to do the job.

2. Maintain momentum. Many projects get off to a **good start** and then slowly disintegrate. To maintain momentum,

- the project manager must provide **incentives** to keep turnover of personnel to an absolute minimum,
- the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of

3. Track progress. For a software project, progress is tracked as work products
are produced and approved (using formal technical reviews) as part of a quality assurance activity.

4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to **"keep it simple."** Whenever possible, decide to use commercial off-the-shelf software or existing software components,

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from

team members and customers, and record findings in written form.

THE W5HH PRINCIPLE

Boehm's W5HH principle is applicable regardless of the size or complexity of a software project

Why is the system being developed?
The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done, by when?
The answers to these questions help the team to establish a project schedule by **identifying key project tasks** and the **milestone**s that are required by the customer.

Who is responsible for a function?
Earlier in this chapter, we noted that the role and responsibility of each member of the **software team** must be defined. The answer to this question helps accomplish this.

Where are they organizationally located?
Not all roles and **responsibilities reside within the software team** itself. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially?
Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed?
The answer to this question is derived by developing estimates based on answers to earlier questions.

CRITICAL PRACTICES

Formal risk management. What are the top ten risks for this project? For each of the risks, what is the chance that the risk will become a problem and what is the impact if it does?

Empirical cost and schedule estimation. What is the current **estimated** size?
of the application software (excluding system software) that will be delivered into operation? How was it derived?

Metric-based project management. Do you have in place a metrics program
to give an early indication of evolving problems? If so, what is the **current requirements** volatility?

Earned value tracking. report **monthly earned value** metrics If so,
are these metrics computed from an activity network of tasks for the entire?
effort to the **next deliver**y

Defect tracking against quality targets. Do you track and periodically report?
the number of defects found by each inspection (formal technical review) and execution test from program inception and the number of defects currently closed and open?

People-aware program management. What is the **average staff turnover?** for the past three months for each of the suppliers/developers involved in the development of software for this system.

Chapter 2

SOFTWARE PROJECTS MANAGEMENT

2.1

Metrics in Process and Project Domain

Measurement is less common in the software engineering world. Metrics should be collected so that process and product indicators can be ascertained.

Process indicators

- It enables software engineering organization to get the efficacy of an existing process.
- They enable managers and practitioners to assess what works and what doesn't.
- Process metrics are collected across all projects and over long periods of time.

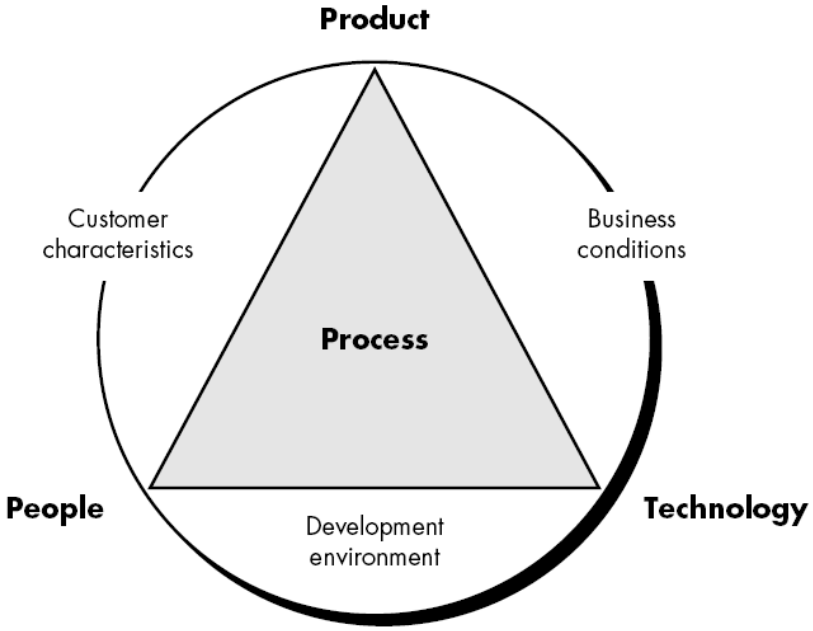
Project indicators

Enable a software project manager to

- assess the status of an ongoing project,
- track potential risks,
- uncover problem areas before they go "critical,"
- adjust work flow or tasks, and
- evaluate the project team's ability to control quality of software work products.

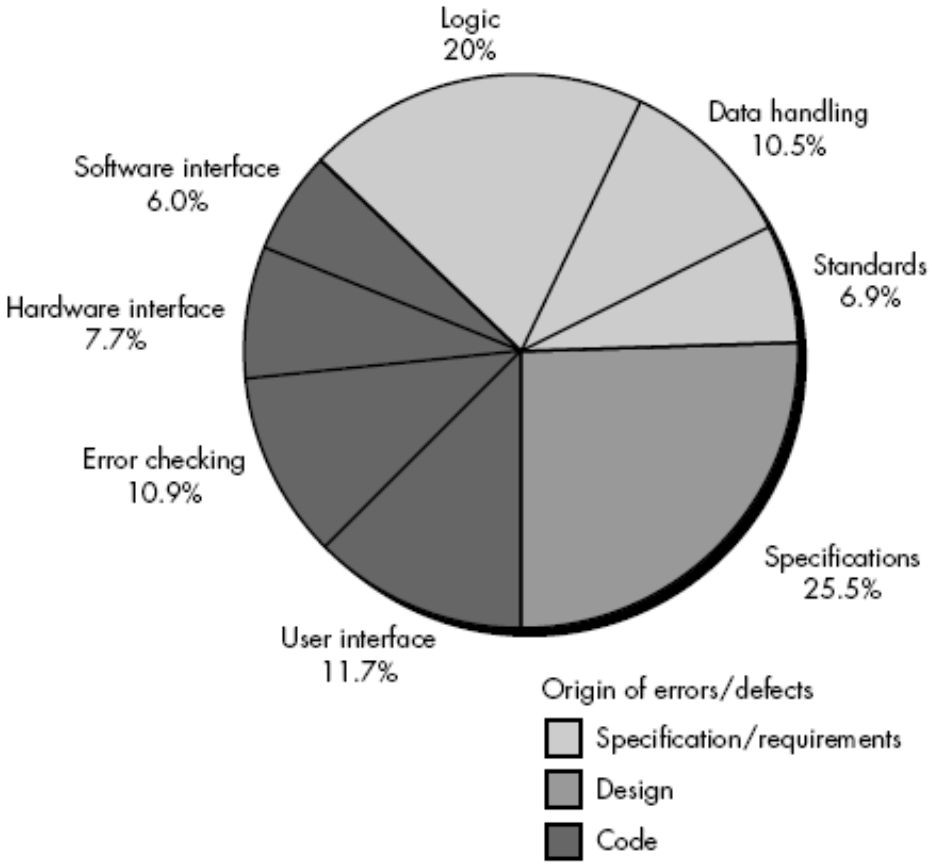
Process Metric

- Process is only one of a number of "controllable factors in **improving software quality and organizational performance.**
- Process **sits at the center** of a triangle connecting three factors that have an influence on software quality.
- We measure the efficacy of a software process indirectly.



- **It shows them how to define processes and how to measure their quality** and productivity.
- Private process data can serve as an important driver as the individual software engineer works to improve.
- Some **process metrics are private to the software project team but public** to all team members.
- Project level defect rates (absolutely not attributed to an individual), **effort, calendar times, and related** data are collected can improve organizational **proces**s performance.
- Software process metrics can provide significant benefit as an organization works to improve its overall level.
- Process Metrics can be useful for failure analysis as

    1. All **errors and defects are categorized** by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).
    2. The **cost to correct** each error and defect is recorded.
    3. The number **of errors and defects** in each category is **counted** and ranked in descending order.
    4. **The overall cost of errors and defects in each category is computed**.
    5. Resultant data are analyzed to uncover the categories that result in highest cost to the organization.
    6. **Plans are developed to modify the process** with the intent of eliminating (or reducing the frequency of) the class of errors and defects that is most costly.



Project Metric

- Project metrics and the indicators derived from them are used to **adapt project work flow and technical activities.**
- The application of **project metrics** on most software projects **occurs during estimation**. **Metrics collected from past projects are used as a basis from which effort and time estimates** are made for current software work.
- As a project proceeds, measures of **effort and calendar time expended are compared to original estimates**
- The project manager **uses data to monitor** and control progress.
- Project metrics begin to have significance Production rates represented in **terms of pages of documentation, review hours, function points, and delivered source lines are measured**.
- The project metrics are used to **minimize the development schedule** by making the adjustments necessary to avoid delays
- **Project metrics are used to assess product quality** on an ongoing basis and, when necessary, modify the technical approach to improve quality.

- **Project metric is leads to a reduction in overall project cost.**

Software project metrics suggests that every project should measure:

• Inputs—measures of the resources required to do the work.
• Outputs—measures of the deliverables or work products created during the software engineering process.
• Results—measures that indicate the effectiveness of the deliverables

Software Measurements

- Measurements in the physical world can be categorized in two ways:
  - ✓ direct measures (e.g., the length of a bolt) and
  - ✓ indirect measures (e.g., the "quality").

- Software metrics can be categorized similarly.
  - Direct measures of the software engineering process include
    cost and effort applied.
    include lines of code (LOC) produced,
    execution speed,
    memory size, and
    defects reported over some set period of time.
  - Indirect measures of the product include
    functionality,
    quality,
    complexity,
    efficiency,
    reliability,
    maintainability,

Size-Oriented Metrics

- Size-oriented measures the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures,
- Such as the one shown in Figure for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. 365 pages of documentation were developed, 134 errors were recorded and 29 defects
- It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

- set of simple size-oriented metrics can be developed for each project:
- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- Cost per LOC.
- Page of documentation per KLOC.
- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

- Size-oriented metrics are not universally accepted as the best way to measure the process of software development

Function-Oriented Metrics

- Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value.
- Since **'functionality' cannot be measured directly**, it must be derived indirectly using other direct measures.
- Its measurements of function points, as

Number of user inputs. Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an **on-line input that results in the generation of some immediate software response** in the form of
an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another.

Metrics for Software Quality

The goal of software engineering is to produce a high-quality system, application, to achieve this goal, software engineers must apply effective methods. Although many quality measures can be collected, the **primary thrust at the project level is to measure errors and defects.** Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework.

Measuring Quality
Although there are many measures of software quality,

Correctness.
Correctness is the degree to which the software performs its required function. The most **common measure for correctness is defects per KLOC,** where a defect is defined as a verified lack of conformance to requirements.

Maintainability.
Maintainability **is the ease with which a program can be corrected if an error is encountered**, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures.

Integrity. Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's **ability to withstand attacks (both accidental and intentional) to its security**. Attacks can be made on all three components of software

Usability.
If a program is not **user-friendly**, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:
(1) the physical and or intellectual skill required to learn the system,
(2) the **time required to become moderately efficient in the use of the system**,
(3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and
(4) a subjective assessment (sometimes obtained through a questionnaire) of user attitudes toward the system.

Integrating Metrics Within the Software Process

- It's important to measure the process of software engineering and the product(software) that it produces.
- **If we do not measure, there no real way of determining whether we are improving**.
- And if we are not improving, we are lost. Hence, metrics is used to establish a process baseline from which improvements can be assessed.
- The collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.
- Metrics includes technical guide for improvement
    - **user requirements are to change**
    - To find **components in this system are most error prone**
    - **No of Testing** should be planned for each component
    - **No of errors expecting during testing**
- The metrics baseline **consists of data collected from past software development projects**.

- Process improvement and/or cost and effort estimation, baseline data must have the following attributes:
    - **data must be** reasonably **accurate**
    - data should be collected for as many projects as possible;
    - measures must be consistent, for example, a line of code must be interpreted consistently across all projects for which data are collected;
    - applications should be similar to work that is to be estimated

Metrics for Small Organizations

- It is reasonable to suggest that software organizations measure and then use the resultant **metrics to help improve their local software process** and the quality
- and timeliness of the products they produce.
- Keep measurements simple, tailored them to each organization, and ensured that they produced valuable information
- **Keep it simple, customize to meet local needs**, and be sure it adds value.
- "Keep it simple" is a guideline that works reasonably well in many activities.
- We begin by focusing not on measurement but rather on results.
- A small organization might select the following set of easily collected measures:
  • **Time (hours or days) elapsed from the time a request is made** until
  Evaluation is complete
  • **Effort (person-hours) to perform** the evaluation
  • Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, teal.
  • **Effort (person-hours) required to make the change**
  • **Time required (hours or days) to make the change**
  • **Errors uncovered during work to make change**, Exchange.
  • **Defects uncovered after change is released** to the customer base,

- Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation.

2.2

Observations on Estimating

- Estimation of **resources, cost, and schedule** for a software engineering effort requires experience in it
- It's an access **to good historical information, requires for the development of software.**
- Although **estimating is as much art as it is science**, this important action need not be conducted
- **Useful techniques** for time and effort estimation do exist.
- Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates.
- **Past experience** can aid immeasurably as estimates are developed and reviewed.
- Because estimation lays **a foundation for all other project planning actions, and project planning provides the road map for successful software engineering**,
- Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists.
- Problem decomposition, an important approach to estimating, becomes more difficult.
- The availability of historical information has a strong influence on estimation risk. By looking back, you can emulate things that worked and improve areas where problems arose.
- **When comprehensive software metrics are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced**.
- Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule.
- **If project scope is poorly understood** or project requirements are subject to change
- As a planner, you and the customer should recognize **that variability in software requirements means instability in cost and schedule**.
- However, you should not become obsessive about estimation. Modern software engineering approaches take an iterative view of development.

- In such approaches, it is possible—although not always politically acceptable—to revisit the estimate and revise it when the customer makes changes to requirements

Project Planning Process

- The objective of software project planning is to provide a framework that enables the manager to make reasonable **estimates of resources**, cost, and schedule.
- In addition, estimates should attempt to define **best-case** and **worst-case** scenarios so that project outcomes can be bounded.
- **The plan must be adapted and updated as the project proceeds**.

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**.

Software Scope and Feasibility

- Software **scope describes the functions and features that are to be delivered to end users**;
- The **first activity in software project planning is the determination of software scope**.
- **Function and performance allocated to software during system engineering should be assessed** to establish a project scope.

- A statement of software scope must be bounded. Software scope describes the data **and control to be processed, function, performance, constraints, interfaces, and reliability**.
- Functions described in the statement of scope are evaluated and, in some cases, refined to provide more detail prior to the beginning of estimation.
- **As both cost and schedule estimates are functionally oriented, some degree of decomposition** is often useful.
- Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.
- Initiate the communication that is essential to establish the scope of the project.
- But a question and answer meeting format are not an approach that has been overwhelmingly successful for understanding the scope
- The Question **&Answers session** s**hould** be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation, and specification.
- Customers and software engineers often have an unconscious "us and them" mindset.
- **A number of independent investigators have developed a team-oriented approach to requirements gathering that can be applied to help establish the scope of** a project.
- **Facilized application specification techniques (FAST)**, this approach encourages the **creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution**, negotiate different approaches, and specify a preliminary set of requirements

Feasibility

- Once scope has been identified it is reasonable to ask: “Can we build software to meet this scope? Is the project feasible?”
- Software engineers rush past these questions only to become mired in a project that is doomed from the onset.
- After a few hours or sometimes a few weeks of investigation, you are sure you can do it again.
- Projects on the margins of your experience are not so easy.
- A team may have to spend several months discovering what the central, difficult-to-implement requirements of a new application actually are.

- Do some of these requirements pose risks that would make the project infeasible? Can these risks be overcome?
- The feasibility team ought to carry initial architecture and design of the high-risk requirements to the point at which it can answer these questions.
- In some cases, when the team gets negative answers, a reduction in requirements may be negotiated.
- Once scope is understood, the software team and others must work to determine if it can be done within the dimensions just noted.
- This is a crucial, although often overlooked, part of the estimation process.

Software feasibility has four solid dimensions:

- Technology—
  1) Is a project **technically feasible**?
  2) Is it **within the state of the art**?
  3) Can **defects be reduced** to a level matching the application's needs?

- Finance—
  1) Is it **financially feasible**?
  2) Can development **be completed at a cost the software organization**?
  3) **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?

Resources

The second planning task is estimation of the resources required to accomplish the software development effort.

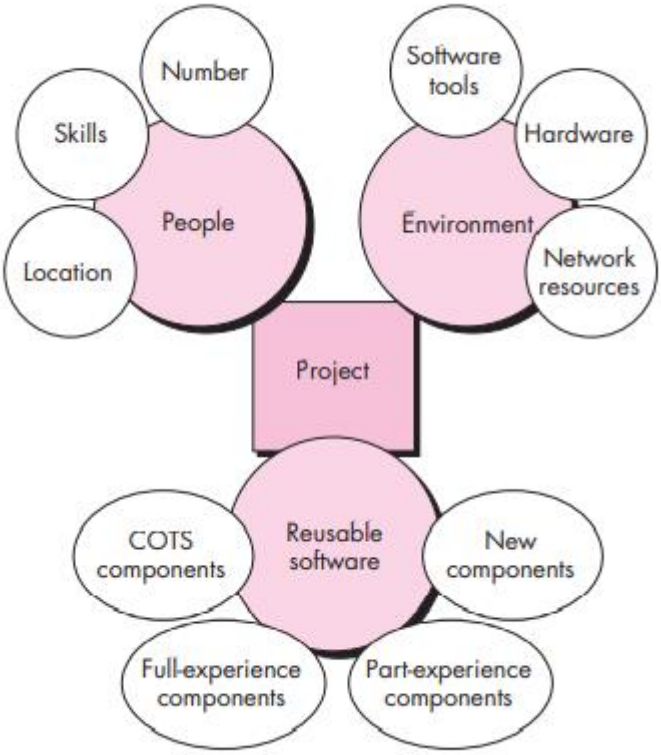


Fig: **Project Resources**

Figure shows the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools).

Human Resources (People)

- 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., telecommunications, database, client-server) are specified
- For larger projects, the **software team** may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified. The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

Reusable Software Resources

- Component-based software engineering (CBSE) emphasizes reusability—that is, the creation and **reuse of software building blocks**.

- Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.
- 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.
- 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.
- Ironically, **reusable software components are often neglected during planning**, only to become a paramount concern later in the software process.

Environmental Resources

- **The environment that supports a software project**, often called the software engineering environment (SEE), incorporates hardware and software.
- **Hardware provides platform** that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.
- When a computer-based system is to be engineered, the software team may require access to hardware elements being developed by other engineering teams.

Software Project Estimation

- Software cost and effort estimation will never be an exact science. Too many variables—**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. we can achieve 100 percent accurate estimates **after the project is complete**
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.

- **Longer you wait, the more you know**, and the more you know, the less likely you are to make serious errors in your estimates.
- 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.
- Ideally, the techniques noted for each option should be applied in tandem; each used as a **cross-check** for the other.
- **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.
- If no historical data exist, costing rests on a very based foundation.

Decomposition Techniques

- Software project estimation is a form of problem solving,
- Estimation uses one or both forms of partitioning.
- But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."
- **Software Sizing** The accuracy of a software project estimate is predicated on a number of things:
  (1) the degree to which you have properly estimated the **size of the product** to be built;
  (2) the ability to translate the **size estimate into human effort, calendar time, and costs**
  (3) the degree to which the project plan reflects the **abilities of the software team**; and
  (4) the stability of product requirements and the environment that supports the **software engineering effort**.
- In the context of project planning, **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).

Various approaches to the sizing problem:

- ✓ "**Fuzzy logic" sizing**. To apply this approach, the **planner must identify the type of application**, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

- ✓ **Function point sizing**. The **planner develops estimates of the information domain** characteristics discussed.
- ✓ **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.
- ✓ **Change sizing**. 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.

Example: LOC-Based Estimation

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 function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

2.3

Software Risks

- Risk always involves two characteristics:
    a. Uncertainty—the risk may or may not happen; that is, there are no 100 percent
    b. Loss—if the risk becomes a reality, unwanted consequences or losses will occur.
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk.
- To accomplish this, different categories of risks are considered.
    - ✓ **Project risks** threaten the project plan-That **project schedule will slip and that costs will increase**.
    - ✓ **Technical risks** threaten the **quality and timeliness** of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.
    - ✓ **Business risks** Candidates for the top five business risks are

(1) building an excellent product or system that no one really wants (**market risk),**
(2) building a product that no longer fits into the overall business strategy
for the company (**strategic risk**),
(3) building a product that the sales force doesn't understand how to sell
(sales risk),
(4) losing the support of senior management due to a change in focus or a
change in **people (management risk**), and
(5) losing budgetary or personnel commitment **(budget risks).**

Risk Identification

- Risk identification is a systematic attempt to specify threats to the project plan
- The checklist can be used for risk identification
    - ✓ Product size—risks associated with the **overall size** of the software to be built or modified.

- ✓ Business impact—risks associated with constraints imposed by management or the **marketplace.**
- ✓ Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer's **ability to communicate with stakeholders** in a timely manner.
- ✓ Process definition—risks associated with the degree to which the **software process has been defined** and is followed by the development organization.
- ✓ Development environment—risks associated with the availability and quality of the tools to be used to build the product.
- ✓ Technology to be built—risks associated with **the complexity of the system to be buil**t and the "newness" of the technology that is packaged by the system.
- ✓ Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Components are defined in the following manner:

- 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 categories—**negligible, marginal, critical, or catastrophic**.

Risk Projection

- Risk projection, also called risk estimation
- Four risk projection steps:
  1. **Establish a scale** that reflects the perceived likelihood of a risk.
  2. **Sketch the consequences** of the risk.
  3. **Estimate the impact of the risk** on the project and the product.
  4. **Assess the overall accuracy of the risk projection** so that there will be no misunderstandings.
- The intent of these steps is to consider risks in a manner that leads to **prioritization**

- **By prioritizing risks, you can allocate resources** where they will have the most impact
- **Risk Projection by developing Risk Table**

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				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Here,

PS: project size risk,
BU implies a business risk
CU: cost
TE: Technology
Testify

- The software team defines a project risk in the following manner:
  Risk identification. Percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.
  Risk probability. Likely as chances.
  Risk impact. Impact on schedule, cost
  Risk exposure. Again, requirement of funds

Risk Refinement

- It may be possible to **refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage**.
- **One way to do this is to represent the risk in condition-transition-consequence.**

- **all reusable software components must conform to specific design standards and that some do not conform**, then there is concern that (possibly).
- only 70 percent of the planned reusable modules may actually be integrated into the as-built system,
- This general condition can be refined in the following manner:

  Sub condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

  Sub condition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

  Sub condition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

- The consequences associated with these refined sub conditions remain the same.

Risk Mitigation Monitoring and Management

- The goal of the risk mitigation, monitoring and management plan is to identify as many potential risks as possible.
- An effective strategy must consider three issues: risk avoidance or mitigation, risk monitoring, and risk management and contingency planning.

RISK MITIGATION:

- If a software team adopts a proactive approach to risk, avoidance is always the best strategy.
- This is achieved by developing a plan for risk mitigation.

RISK MONITORING:

- The project manager monitors factors that may provide an indication of whether the risk is becoming more or less.

RISK MANAGEMENT:

- Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality.

Example:

Risk: Late Delivery

- **Mitigation**

The cost associated with a late delivery is critical. A late delivery will result in a late delivery of a letter of acceptance from the customer. Without the letter of acceptance, the group will receive a failing grade for the course. Steps have been taken to ensure a timely delivery by gauging the scope of project based on the delivery deadline.

- **Monitoring**

A schedule has been established to monitor project status. Falling behind schedule would indicate a potential for late delivery. The schedule will be followed closely during all development stages.

- **Management**

Late delivery would be a catastrophic failure in the project development. If the project cannot be delivered on time the development team will not pass the course. If it becomes apparent that the project will not be completed on time, the only course of action available would be to request an extension to the deadline form the customer.

2.4

Software Configuration Management

- The items that comprise **all information produced as part of the software process** are collectively called a software configuration.
- Software configuration **management is a set of activities that have been developed to manage change** throughout the life cycle of computer software.
- **SCM can be viewed as a software quality assurance activity that is applied throughout the software process**.

SCM Repository

- In the early days of software engineering, software configuration items were maintained as paper documents (or punched computer cards!), placed in file folders or This approach was problematic for many reasons:
- During the early history of software engineering, the repository was indeed a person—the **programmer who had to remember the location of all information relevant** to a software project, who had **to recall information** that was never written down and **reconstruct information that had been lost**
- **Today, the repository is a "thing"— a database that acts as the center for both accumulation and storage of software engineering information.**
- The role of the person (the software engineer) is to interact with the repository using tools that are integrated with it.

Advantages

- **finding a configuration item** when it was needed is easier
- Determining **which items were changed**, **when and by whom** was often successfully
- **constructing a new version** of an existing program was
- **describing detailed or complex relationships** between configuration items are easy

The Role of the Repository

- The SCM repository is the set of mechanisms and data structures that **allow a software team to manage change** in an effective manner.

- It provides the obvious functions of a **modern database management system by** ensuring data integrity, sharing, and integration.
- In addition, the SCM repository **provides a hub for the integration of software tools,**
- Is **central to the flow of the software process, and can enforce uniform structure and format for software engineering work products**.
- It determines how **data can be accessed by tools** and viewed by software engineers,
- It determines how well **data security** and **integrity** can be maintained,
- It determines **how easily the existing model can be extended** to accommodate new needs

General Features and Content

- The features and content of the repository are best understood from **two perspectives:**
  1. **what is to be stored** in the repository and
  2. **what specific services are provided** by the repository?
- A repository provides two different classes of services:
  1. the same types of **services that might be expected from any sophisticated database management system** and
  2. services that **are specific to the software engineering** environment.
- A repository that serves a software engineering team should also
  1. integrate with or **directly support process management functions**
  2. support **specific rules** that govern the **SCM function** and the data **maintained within the repository**,
  3. Provide an **interface to other software engineering tools, and**
  4. Accommodate **storage of sophisticated data objects** (e.g., text, graphics, video, audio).

SCM Features

Versioning.

- As a project progresses, **many versions** of individual **work products will be created**.
- **The repository** must be able to **save all of these versions** to enable effective management of product releases and

- **permit developers to go back to previous versions during testing and debugging**.
- A mature repository **tracks version.**

Dependency tracking and change management.

- The **repository manages a wide variety of relationships among the data elements stored** in it,
- Include **relationships between enterprise entities and processes**, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on.
- Some of these relationships are merely **associations**, and some are **dependencies** or mandatory relationships.
- The ability to keep track of all of these relationships **is crucial to the integrity of the information stored in the repository**

.

Requirements tracing.

- **provides the ability to track all the design and construction components** and deliverables that result from a specific requirements specification (forward tracing).
- it provides the ability to **identify which requirement generated** any given work product (backward tracing).

Configuration management.

A configuration management facility keeps **track of a series of configurations representing specific project milestones or production releases**.

Audit trails.

An audit trail establishes additional information about **when, why, and by whom changes are made**.

SCM Process

The software configuration management process four primary objectives:

1) to **identify all items** that collectively define the software configuration
2) **to manage changes to one or more of these items**
3) to facilitate the **construction of different versions of an application**
4) to **ensure that software quality is maintained** as the configuration evolves

Identification of Objects in the Software Configuration

- To control and manage software configuration items, **each should be separately named and then organized using an object-oriented approach**.
- Two types of objects can be identified
  - basic objects and
  - aggregate objects.
- A basic object is a unit of **information that you create during analysis, design, code,** or test.
  - For example, a basic object might be a section of a requirements specification, **part of a design model, source code for a component**, or a suite of test cases that are used to exercise in code.
- An aggregate object **is a collection of basic objects and other** aggregate objects.
  - For example, a **Design Specification is an aggregate** object.
- Each **object has a set of distinct features** that identify it uniquely: **a name, a description, a list of resources, and a "realization."**
- The **object name is a character string** that identifies the object unambiguously.
- The **object description is a list of data items** that identify the SCI type (e.g., model element, program, data)

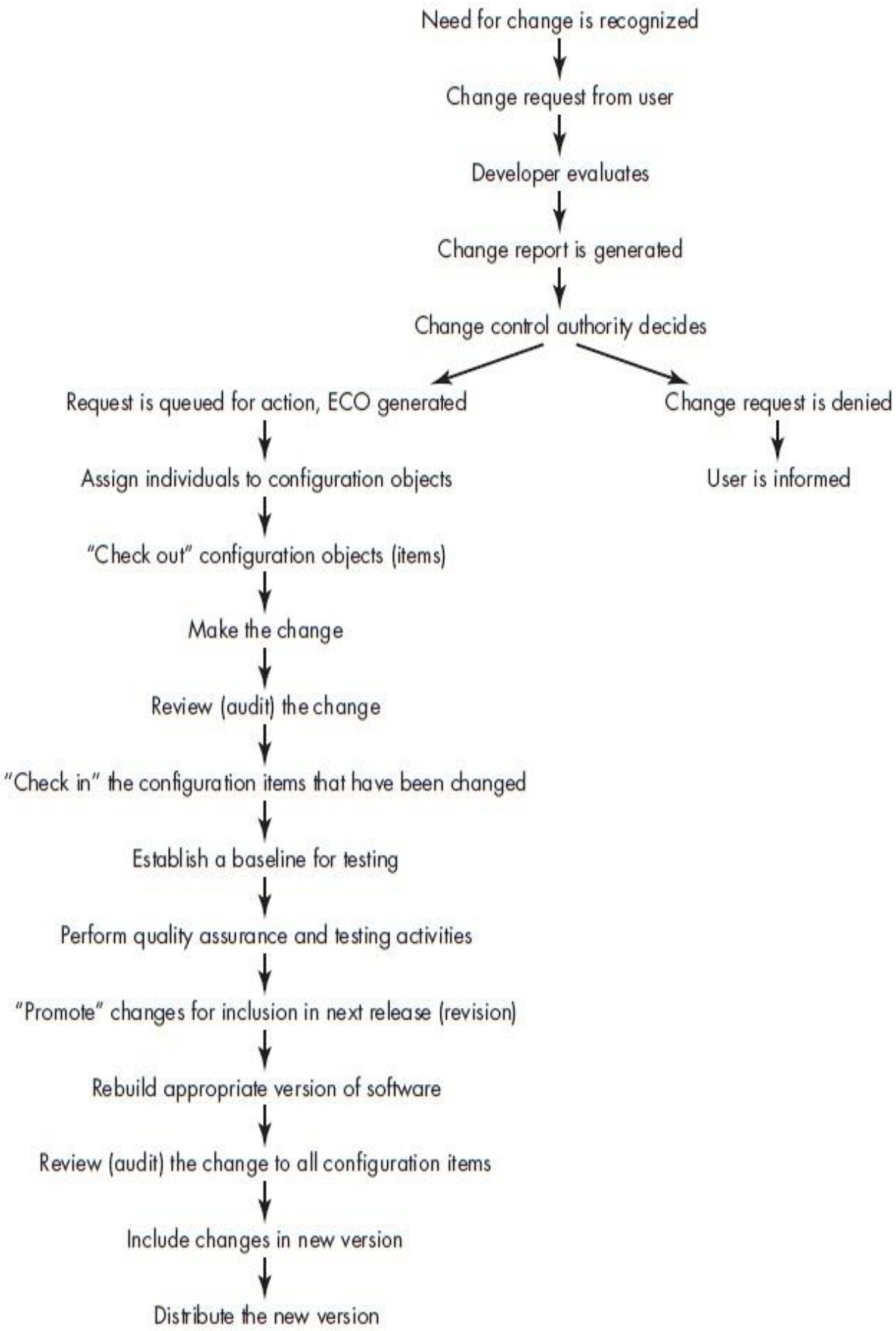
Version Control

- Version control combines procedures and tools to **manage different versions** of configuration objects **that are created during the software process**.
- A version control system implements or **is directly integrated with given a project database (repository)** that stores all relevant configuration objects,
- a **version management capability that stores all versions of a configuration object**
- It enables any **version to be constructed using differences from past versions**
- It enables to collect all relevant configuration and objects and **construct a specific version of the software**.
- In addition, version control **enables the team to record and track the status of all outstanding issues associated with each configuration object**.

- A number of version control systems establish a **collection of all changes that are required to create a specific version** of the software.
- A version control **enables to construct a version of the software by specifying the change sets**

Change Control

- Worry about change because it can create a big failure in the product, but it can also fix a big failure.
- Worry about change because a single developer could sink the project; but a change control process could effectively discourage them from doing creative.
- For a large software project, uncontrolled change rapidly leads to Failure. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change.
- The change control process is illustrated schematically in Figure
  A **change request is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change**.
- The results of the evaluation are presented as a change report, which is used by a change control authority (CCA)—a person or group that makes a final decision.
- An engineering change order (ECO) is generated for each approved change.
- The ECO describes the change to be made, the constraints that must be respected.


Need for change is recognized
Change request from user
Developer evaluates
Change report is generated
Change control authority decides
Request is queued for action, ECO generated
Change request is denied
Assign individuals to configuration objects
User is informed
"Check out" configuration objects (items)
Make the change
Review (audit) the change
"Check in" the configuration items that have been changed
Establish a baseline for testing
Perform quality assurance and testing activities
"Promote" changes for inclusion in next release (revision)
Rebuild appropriate version of software
Review (audit) the change to all configuration items
Include changes in new version
Distribute the new version

Chapter 3

SOFTWARE REQUIREMENT AND SCHEDULING

REQUIREMENTS ENGINEERING

- The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering.
- From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modelling activity.
- Requirements engineering HELPS for understanding what the customer wants, analysing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

It encompasses distinct tasks

Inception.

- But in general, most projects begin when a business need is identified or a potential new market or service is discovered.
- Stakeholders from the business community define a business case for the idea, try to identify the breadth and depth of the market,
- All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization
- It establishes a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation.

- It tells the objectives for the system or product are, what is to be accomplished,
- It shows how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.
- problems that are encountered as elicitation occurs.
    - Problems of scope.
    - Problems of understanding.
    - Problems of volatility. (The requirements change over time)

Elaboration.

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.

- It identifies various aspects of software function, behaviour, and information.
- Elaboration is driven by the creation and refinement of user scenarios
- It describes how the end user (and other actors) will interact with the system.

Negotiation.

- It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources.
- Its need to reconcile these conflicts through a process of negotiation.
- Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
- Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated,

Specification.

- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- a "standard template" should be developed and used for a specification,
- However, it is sometimes necessary to remain flexible when a specification is to be developed.
- For large systems, a written document, combining natural language descriptions and graphical models may be the best approach

Validation.

- The work products produced as a requirement engineering are assessed for quality during a validation step.
- Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected;
- The primary requirements validation mechanism is the technical review
- The review team that validates requirements examine the specification looking for errors in content or interpretation,

Requirements management.

- Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system.
- Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- Many of these activities are identical to the software configuration management (SCM) techniques.

SRS (SOFTWARE REQUIREMENTS SPECIFICATION)

- Is a document which is used as a communication medium between the customer and the supplier.
- When the software requirement specification is completed and is accepted by all parties, the end of the requirements engineering phase has been reached.
- After the acceptance phase, any of the requirements cannot be changed, but the changes must be tightly controlled.
- The software requirement specification should be edited by both the customer and the supplier,

NEED OF SRS

- company publishing a software requirement specification to companies for competitive tendering,
- company writing their own software requirement specification in response to a user requirement document.
- SRS allow s number of different suppliers to propose solutions,
- Use to identify any constraints which must be applied.
- SRS is used to capture the user's requirements and
- SRS highlights inconsistencies and conflicting requirements and define system and acceptance testing activities.

CHARACTERISTICS OF A GOOD SRS

Complete

- Description of all major requirements relating to functionality, performance, design constraints and external interfaces.
- Definition of the response of the software system to all reasonable situations.
- Conformity to any software standards, detailing any sections which are not appropriate.
- Have full labelling and references of all tables and references, definitions of all terms and units of measure.
- Be fully defined, if there are sections in the software requirements specification still to be defined, the software

Consistent

types of confliction:

- Multiple descriptors - This is where two or more words are used to reference the same item, i.e. where the term cue and prompt are used interchangeably.
- Opposing physical requirements - This is where the description of real-world objects clash, e.g. one requirement states that the warning indicator is orange, and another states that the indicator is red.
- Opposing functional requirements - This is where functional characteristics conflict, e.g. perform function X after both A and B has occurred, or perform function X after A or B has occurred

Traceable

A software requirement specification is traceable if both the origins and the references of the requirements are available. Traceability of the origin or a requirement can help understand who asked for the requirement and also what modifications have been made to the requirement to bring the requirement to its current state. Traceability of references are used to aid the modification of future documents by stating where a requirement has been referenced. By having foreword traceability, consistency can be more easily contained

Unambiguous

As the Oxford English dictionary states the word unambiguous means "not having two or more possible meanings". This means that each requirement can have one and only one interpretation. If it is unavoidable to use an ambiguous term in the requirements specification, then there should be clarification text describing the context of the term. One way of removing ambiguity is to use a formal requirements specification language. The advantage to using a formal language is the relative ease of detecting errors by using lexical syntactic analysers to detect ambiguity. The disadvantage of using a formal requirements specification language is the learning time and loss of understanding of the system by the client.

Verifiable

A software requirement specification is verifiable if all of the requirements contained within the specification are verifiable. A requirement is verifiable if there exists a finite cost-effective method by which a person or machine can check that the software product meets the requirement. Non-verifiable requirements include "The system should have a good user interface" or "the software must work well under most conditions" because the performance words of good, well and most are subjective and open to interpretation. If a method cannot be devised to determine whether the software meets a requirement, then the requirement should be removed or revised

COMPONENTS OF THE SRS

Given below are the system properties that an SRS should specify. The basic issues, an SRS must address are:

1. Functional requirements

2. Non-functional requirements
3. Performance requirements
4. Design constraints
5. External interface requirements

Conceptually, any SRS should have these components. Now we will discuss them one by one.

1. Functional Requirements

- Functional requirements specify what output should be produced from the given inputs. So, they basically describe the connectivity between the input and output of the system. For each functional requirement:
- A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:
- All the operations to be performed on the input data obtain the output should be specified, and
- Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.
- It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behaviour of the system for invalid inputs and invalid outputs.

2.Non-functional Requirements

Characteristics of the system which cannot be expressed as functions:

- Maintainability,
- Portability,
- Usability,
- Security,
- Safety, etc.

3. Performance Requirements (Speed Requirements)

This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transaction per second or response time from the system for a user event or screen refresh time or a combination of these. It is a good idea to pin down performance requirements for the most used or critical transactions, user events and screens.

4. Design Constraints

The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may

have an impact on the design of the system. An SRS should identify and specify all such constraints.

Standard Compliance: It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

Hardware Limitations: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can include the types of machines to be used operating system availability memory space etc.

Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

Security: Currently security requirements have become essential and major for all types of systems. Security requirements place restriction s on the use of certain commands control access to database, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

5. External Interface Requirements

For each external interface requirements:

- All the possible interactions of the software with people hardware and other software should be clearly specified,
- The characteristics of each user interface of the software product should be specified and
- The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

STRUCTURE OF SRS

A software requirements specification is a document which is used as a communication medium between the customer and the supplier. When the software requirement specification is completed and is accepted by all parties, the end of the requirements engineering phase has been reached.
The requirements document is devised in a manner that is easier to write, review, and maintain. It is organized into independent sections and each section is organized into modules or units. Note that the level of detail to be included in the SRS depends on the type of the system to be developed and the process model chosen for its development.
Document comprises the following sections.

1. Introduction: This provides an overview of the entire information described in SRS. This involves purpose and the scope of SRS, which states the functions to be performed by the system. In addition, it describes definitions, abbreviations, and the acronyms used. The references used in SRS provide a list of documents that is referenced in the document.

2. Overall description: It determines the factors which affect the requirements of the system. It provides a brief description of the requirements to be defined in the next section called 'specific requirement'. It comprises the following sub-sections.

3. Product perspective: It determines whether the product is an independent product or an integral part of the larger product. It determines the interface with hardware, software, system, and communication. It also defines memory constraints and operations utilized by the user.

4. Product functions: It provides a summary of the functions to be performed by the software. The functions are organized in a list so that they are easily understandable by the user:

5. User characteristics: It determines general characteristics of the users.

6. Constraints: It provides the genera1 description of the constraints such as regulatory policies, audit functions, reliability requirements, and so on.

7. Assumption and dependency: It provides a list of assumptions and factors that affect the requirements as stated in this document.

8. Apportioning of requirements: It determines the requirements that can be delayed until release of future versions of the system.

9. Specific requirements: These determine all requirements in detail so that the designers can design the system in accordance with them. The requirements include description of every input and output of the system and functions performed in response to the input provided.

10. External interface: It determines the interface of the software with other systems, which can include interface with operating system and so on. External interface also specifies the interaction of the software with users, hardware, or other software. The characteristics of each user interface of the software product are specified in SRS. For the hardware interface, SRS specifies the logical characteristics of each interface among the software and hardware components. If the software is to be executed on the existing hardware, then characteristics such as memory restrictions are also specified.

11. Functions: It determines the functional capabilities of the system. For each functional requirement, the accepting and processing of inputs in order to generate outputs are specified. This includes validity checks on inputs, exact sequence of operations, relationship of inputs to output, and so on.

12. Performance requirements: It determines the performance constraints of the software system. Performance requirement is of two types: static requirements and dynamic requirements. Static requirements (also known as capacity requirements) do not impose

constraints on the execution characteristics of the system. These include requirements like number of terminals and users to be supported. Dynamic requirements determine the constraints on the execution of the behaviour of the system, which includes response time (the time between the start and ending of an operation under specified conditions) and throughput (total amount of work done in a given time).

13. Logical database of requirements: It determines logical requirements to be stored in the database. This includes type of information used, frequency of usage, data entities and relationships among them, and so on.

14. Design constraint: It determines all design constraints that are imposed by standards, hardware limitations, and so on. Standard compliance determines requirements for the system, which are in compliance with the specified standards. These standards can include accounting procedures and report format. Hardware limitations implies when the software can operate on existing hardware or some pre-determined hardware. This can impose restrictions while developing the software design. Hardware limitations include hardware configuration of the machine and operating system to be used.

15. Software system attributes: It provide attributes such as reliability, availability, maintainability and portability. It is essential to describe all these attributes to verify that they are achieved in the final system.

16. Organizing Specific Requirements: It determines the requirements so that they can be properly organized for optimal understanding. The requirements can be organized on the basis of mode of operation, user classes, objects, feature, response, and functional hierarchy.

17. Change management process: It determines the change management process in order to identify, evaluate, and update SRS to reflect changes in the project scope and requirements.

18. Document approvals: These provide information about the approvers of the SRS document with the details such as approver's name, signature, date, and so on.

19. Supporting information: It provides information such as table of contents, index, and so on. This is necessary especially when SRS is prepared for large and complex projects.

IEEE 830-1998 Standard for SRS

- Title
- Table of Contents
- 1. Introduction
  - 1.1 Purpose
  - 1.2 Scope
  - 1.3 Definitions. Acronyms, and Abbreviations
  - 1.4 References
  - 1.5 Overview
- 2. Overall Description
- 3. **Specific Requirements**
- Appendices
- Index

•Describe purpose of the system
•Describe intended audience

•What the system will and will not do

•Define the vocabulary of the SRS (may also be in appendix)

•List all referenced documents and their sources SRS (may also be in appendix)

•Describe how the SRS is organized

IEEE 830-1998 Standard – Section 2 of SRS

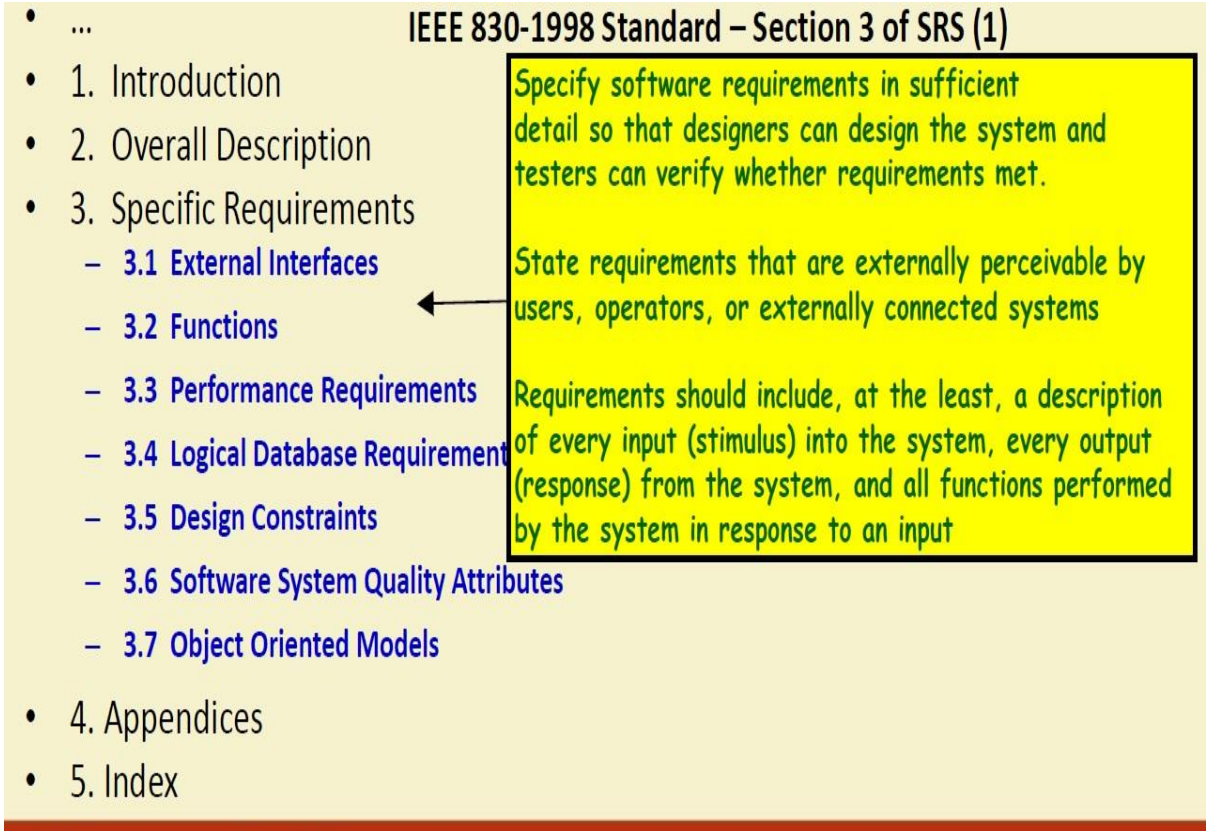
- Title
- Table of Contents
- 1. Introduction
- 2. **Overall Description**
  - 2.1 Product Perspective
  - 2.2 Product Functions
  - 2.3 User Characteristics
  - 2.4 Constraints
  - 2.5 Assumptions and Dependencies
- 3. Specific Requirements
- 4. Appendices
- 5. Index

•Present the business case and operational concept of the system
•Describe external interfaces: system, user, hardware, software, communication
•Describe constraints: memory, operational, site adaptation

•Summarize the major functional capabilities

•Describe technical skills of each user class

•Describe other constraints that will limit developer's options; e.g., regulatory policies; target platform, database, network, development standards requirements



WHAT ARE THE VALIDATION METHODS PERFORMED?

There are three methods of validation performed:

1. **Domain-validated certificates:** Only the verified owner of the domain name can purchase an SSL certificate for the domain. Validation is done via email sent to the domain owner. Domain validated SSL certificates can be issued very quickly - often in minutes.

2. **Organization-validated certificates:** When corporate identity validation is important, an SSL Certificate for the organization assures customers that the website is trustworthy and secure. Only verified representatives of the organization may purchase these certificates and business licences or other proof is required. The Certificate Authority will verify through phone call to ensure that the certificate request is legitimate.

3. **Extended Validation (EV) certificates:** With Extended Validation, as well as displaying the certificate seal, the address bar is displayed in green, providing customers with an extra level of confidence. The green address bar is a strong visual indication that the site has an Extended Validation Certificate. The Security Status bar displays the organization name and the name of the Certificate Authority (CA).

   In order to be approved for an Extended Validation certificate, the certificate authority will actively check the Organization and the individual applying for the certificate.

This is to verify that the Organization is positively the Organization they claim to be, and the individual requesting the certificate is someone who is authorized to request a digital certificate. Extended Validation may take as long as one week to complete.

ESTABLISHING GROUNDWORK

- The requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team.
- The steps required to establish the groundwork for an understanding of software requirements—

Identifying Stakeholders

- Identify benefits in a direct or indirect way from the system, also identify beneficiaries which is being developed.
- At inception, you should create a list of people who will contribute input as requirements.

Recognizing Multiple Viewpoints

- As many different stakeholders exist, the requirements of the system will be explored from many different points of view.
- Also, be consider views nontechnical stakeholders
- Support engineers may focus on the maintainability of the Software, each of these constituencies will contribute information to the requirements engineering process. As information from multiple viewpoints is collected,
- Emerging requirements may be inconsistent or may conflict with one another.
- It is Important to take consider all stakeholders view

Working toward Collaboration

- If five stakeholders are involved in a software project, we may have five different opinions about the proper set of requirements. Consider and collaborate all views
- customers and other stakeholders must collaborate among themselves and with software engineering practitioners its job of a requirements engineer to identify areas of commonality
- Also identify and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder).

Asking the Questions

- Questions asked at the inception of the project the first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits.

- For example, we might ask:
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution?
  - Is there another source for the solution that you need?
- These questions help to identify all stakeholders who will have interest in the software to be built.
- The questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development

ELICITING REQUIREMENTS

- Requirements elicitation also called requirements gathering, combines elements of problem solving, elaboration, negotiation, and specification.
- In team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose solution, negotiate etc

Collaborative Requirements Gathering

- Possibility that Each stakeholder makes use of a slightly different scenario, but all apply some variation
- For this
  - Meetings are conducted and attended by both software engineers and other stakeholders.
  - Rules for preparation and participation are established.
  - An agenda enough to cover all important points

Quality Function Deployment

- Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- It concentrates on maximizing customer satisfaction from the software engineering process"
- QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.
- QFD identifies three types of requirements
  Normal requirements. (graphical displays, specific system functions,)
  Expected requirements. (ease of human/machine interaction,
  correctness and reliability, and ease of software installation.)
  Exciting requirements. (e.g., multitouch screen, visual voice mail)

Usage Scenarios

- It is difficult to understand how functions and features will be used by different classes of end users.
- The developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed.

Elicitation Work Products

work products include

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.

DEVELOPING USE CASES

- use case depicts the software or system from the end user's point of view.
- The first step in writing a use case is to define the set of "actors" that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behaviour that is to be described.
- Actors represent the roles that people (or devices) play as the system operates.
- Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.
- Every actor has one or more goals when using the system.
- It is important to note that an actor and an end user are not necessarily the same thing.
- A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case.
- As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines.
- After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode.
- Therefore, four actors can be defined: programmer, tester, monitor, and trouble-shooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.
- Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration.
- It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system.

- Primary actors interact to achieve required system function and derive the intended benefit from the system.
- They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions that should be answered by a use case:

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

VALIDATING REQUIREMENTS

The requirements represented by the model are prioritized by the stakeholders and requirements will be implemented as software increments.

A review of the requirements model addresses the following questions before validation:

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

Project Scheduling

Basic Concept

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks

Reasons for software delivered late:

• An unrealistic deadline established

• Changing customer requirements that are not reflected in schedule changes.

• An honest underestimate of the amount of effort and/or the number of resources.

• Predictable and/or unpredictable risks that were not considered.

• Technical difficulties that could not have been foreseen in advance.

• Human difficulties that could not have been foreseen in advance.

• Miscommunication among project staff that results in delays, etc.

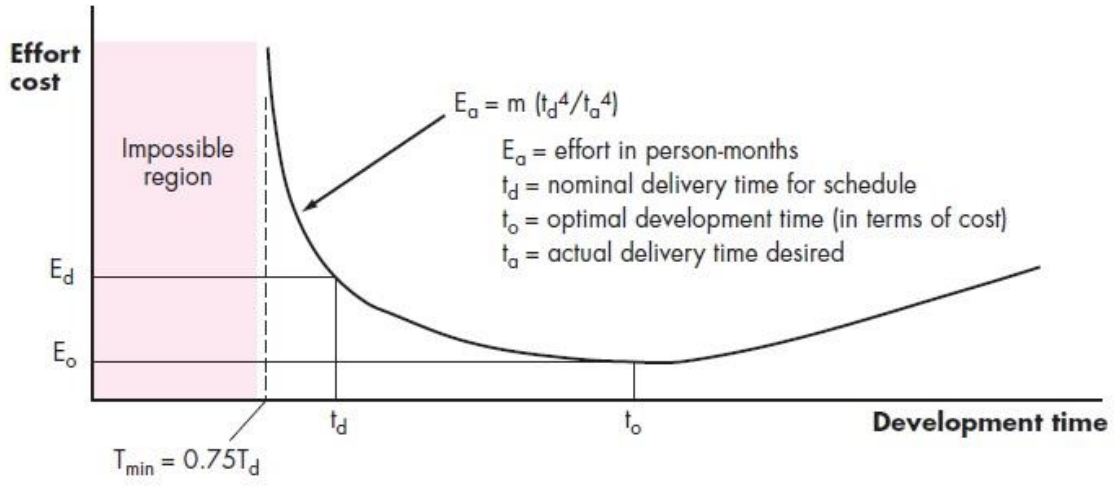
Basic Principles Software project scheduling

- **Compartmentalization**. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
- **Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.
- **Time allocation.** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
- **Effort validation.** Every project has a defined number of people on the software team. As time allocation occurs, you must 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. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. 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.

People and Effort

- In a small software development of project, a single person can analyse requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.
- There is a common **myth** that is still believed by many managers who are responsible for software development projects: **"If we fall behind schedule, we can always add more programmers and catch up later in the project."** Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further.
- The people who **are added must learn the system**, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.
- Over the years, eit founds that **project schedules are elastic**. That is, it is possible to compress a desired project completion date by adding additional resources to some extent.
- It is also possible to extend a completion date (by reducing the number of resources). The Putnam-Norden-Rayleigh (PNR) Curve5 provides an indication of the relationship between effort applied and delivery time for a software project.



A curve, representing project effort as a function of delivery time, is shown in Figure

The curve indicates a minimum value to that indicates the least cost for delivery

we assume that a project team has estimated a level of effort **Ed**

will be required to achieve a nominal delivery time **td** that is optimal in terms of schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of td. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond 0.75td. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high.

The PNR curve also indicates that the lowest cost delivery option,
to _ 2td. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.

Effort Distribution

- Each of the software project estimation, leads to estimates of work unit person-months required to complete software development.
- A recommended distribution of effort across the software process is often referred to as the **40–20–40** rule.
- **Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing**. And coding (20 percent of effort) is deemphasized.
- This effort distribution should be used as a guideline only. The characteristics of each project dictate the distribution of effort.
- **Work expended on project planning rarely accounts for more than** 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk.
- Customer communication and requirements analysis may comprise 10 to 25 percent of project effort.
- A range of 20 to 25 percent of effort is normally applied to software design.
- The criticality of the software often dictates the amount of testing that is required. If software is human rated, even higher percentages are typical

Task Set

- 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
- Task Set enable you to define, develop, and support computer software
- Process model that is chosen, or the work that a software team performs is achieved through a set of tasks that
- No single task set is appropriate for all projects. The set of tasks that would be requires
- An effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

Example of Task Set

Concept development projects are approached by applying the following actions (Concept development projects that are initiated to explore some new business concept or application of some new technology.)

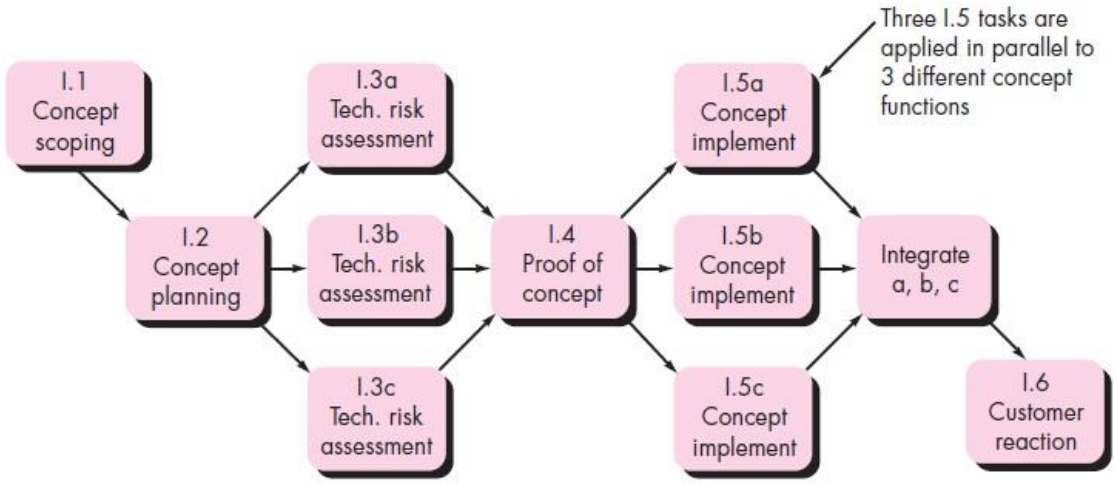
- Concept **scoping determines** the overall scope of the project.
- Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.
- **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.
- Proof of concept demonstrates the viability of a **new technology in the software context**.

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.

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

Task Network

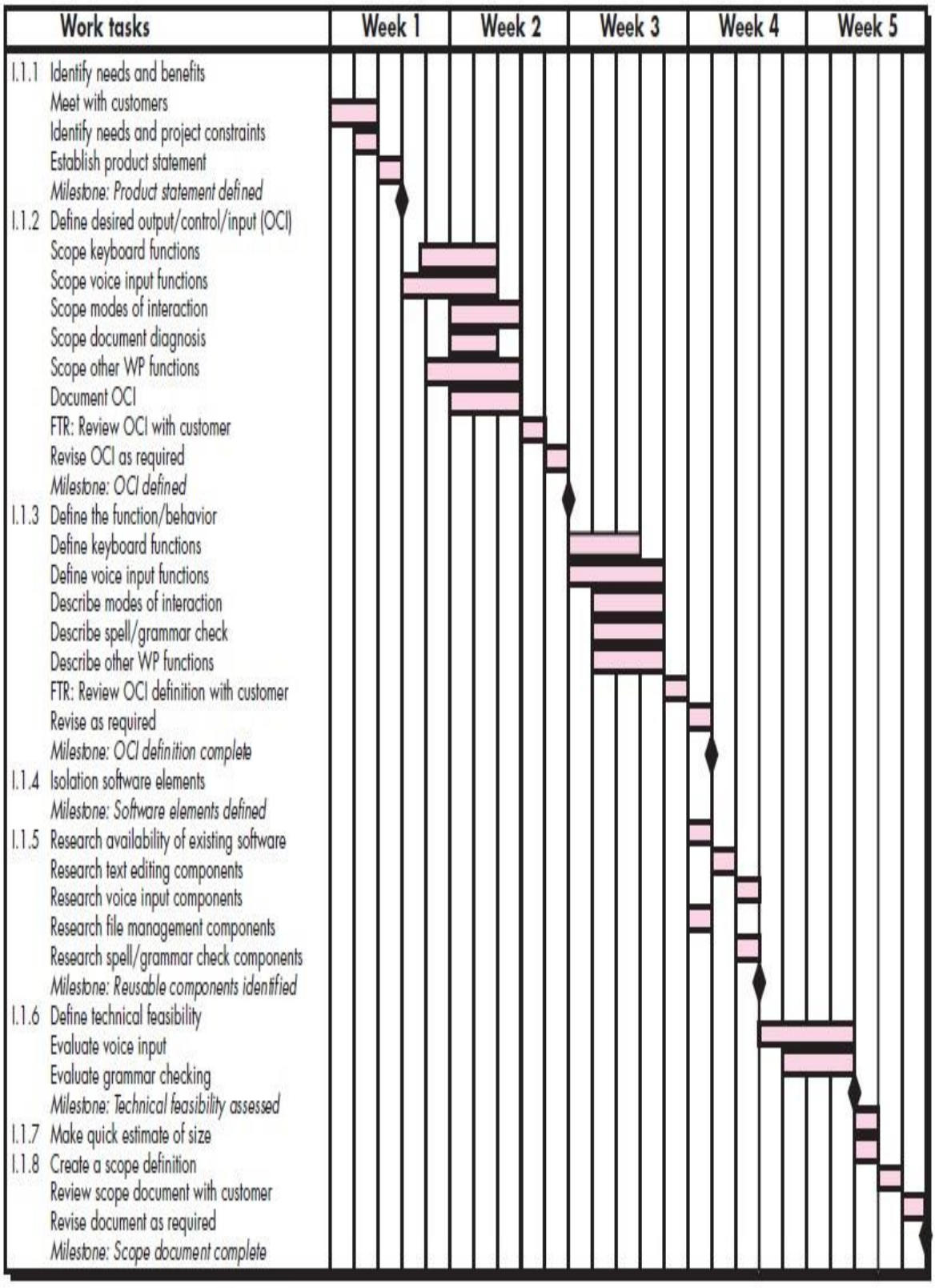
- **Individual tasks and subtasks** have interdependencies based on their sequence.
- when more than one person is involved in a software engineering project,development activities and tasks will be performed in parallel, concurrent tasks must be coordinated
- A task network, also called an activity network, is a graphic representation of the task flow for a project.
- It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool.
- the task network depicts major software engineering actions.



- The concurrent nature of software engineering actions leads to a number of important scheduling requirements.

GANTT CHART

- Time-line chart, also called a Gantt chart,A time-line chart can be developed for the entire project.
- Alternatively, separate charts can be developed for each project function or for each individual working on the project.
- Figure illustrates the format of a time-line chart. It depicts a part of a software project schedule
- All project tasks (for concept scoping) are listed in the lefthand column.
- The horizontal bars indicate the duration of each task.
- When multiple bars occur at the same time on the calendar, task concurrency is implied.
- The diamonds indicate milestones.

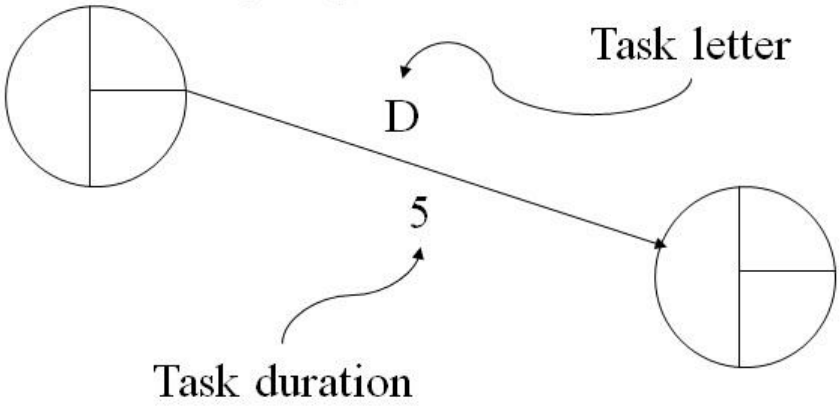

Work tasks
Week 1
Week 2
Week 3
Week 4
Week 5
I.1.1 Identify needs and benefits
Meet with customers
Identify needs and project constraints
Establish product statement
Milestone: Product statement defined
I.1.2 Define desired output/control/input (OCI)
Scope keyboard functions
Scope voice input functions
Scope modes of interaction
Scope document diagnosis
Scope other WP functions
Document OCI
FTR: Review OCI with customer
Revise OCI as required
Milestone: OCI defined
I.1.3 Define the function/behavior
Define keyboard functions
Define voice input functions
Describe modes of interaction
Describe spell/grammar check
Describe other WP functions
FTR: Review OCI definition with customer
Revise as required
Milestone: OCI definition complete
I.1.4 Isolation software elements
Milestone: Software elements defined
I.1.5 Research availability of existing software
Research text editing components
Research voice input components
Research file management components
Research spell/grammar check components
Milestone: Reusable components identified
I.1.6 Define technical feasibility
Evaluate voice input
Evaluate grammar checking
Milestone: Technical feasibility assessed
I.1.7 Make quick estimate of size
I.1.8 Create a scope definition
Review scope document with customer
Revise document as required
Milestone: Scope document complete

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	require more
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	effort/time
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							

PERT (PROJECT EVALUATION REVIEW TECHNIQUE)

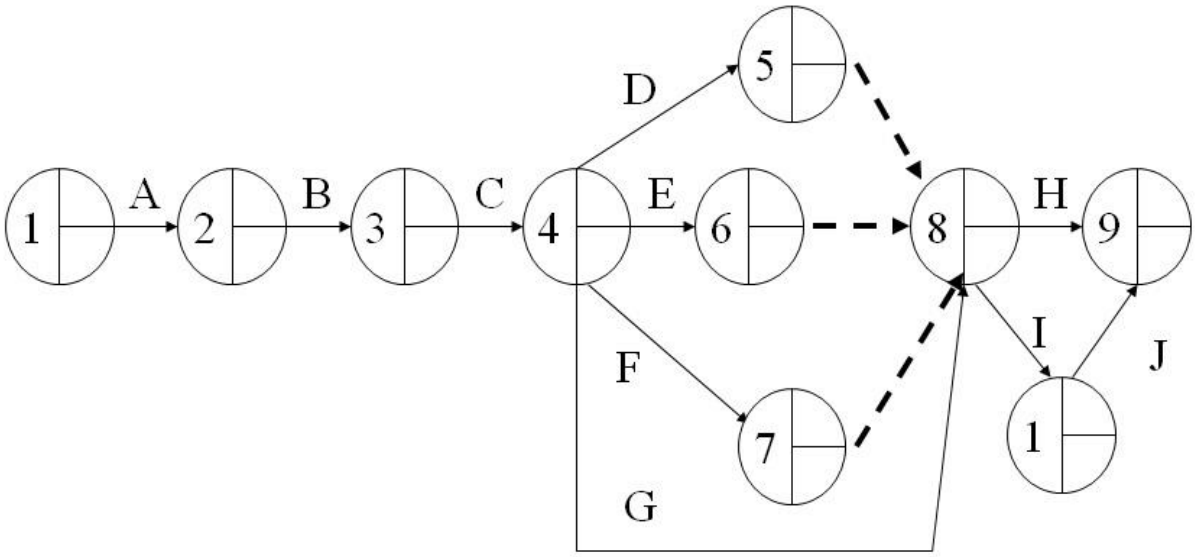
TASK ID	Task Description	Preceed ID	Succ. ID	Duration
A	Specification	1	2	3
B	High Level Design	2	3	2
C	Detailed Design	3	4	2
D	Code/Test Main	4	5	7
E	Code/Test DB	4	6	6
F	Code/Test UI	4	7	3
G	Write test plan	4	8	2
	Dummy Task	5	8	
	Dummy Task	6	8	
	Dummy Task	7	8	
H	Integrate/System Test	8	9	5
I	Write User Manual	8	10	2
J	Typeset User Manual	10	9	1

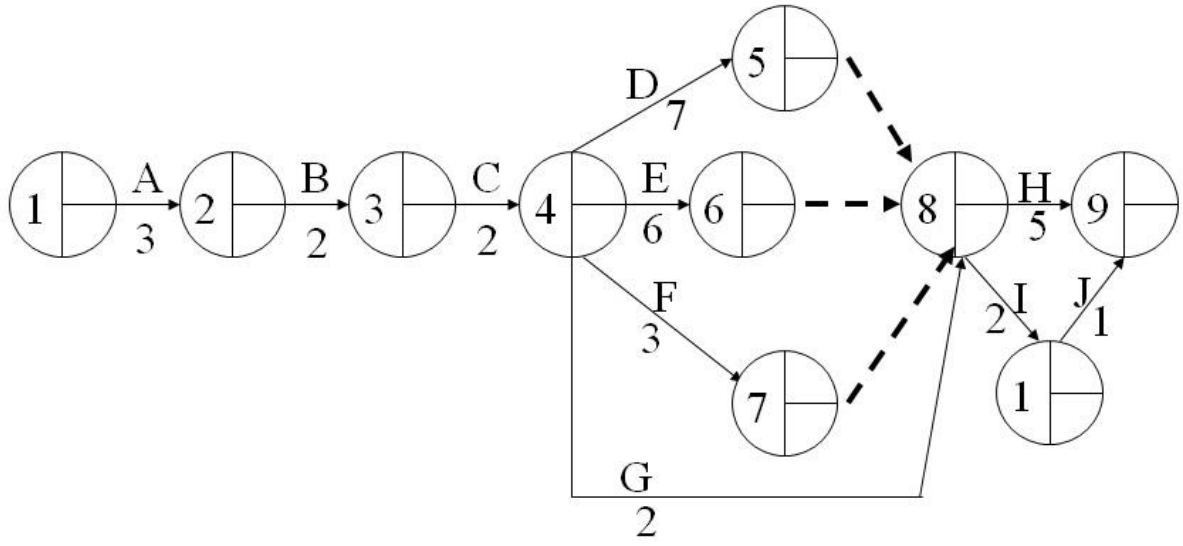
- **Parts of a task/activity edge**



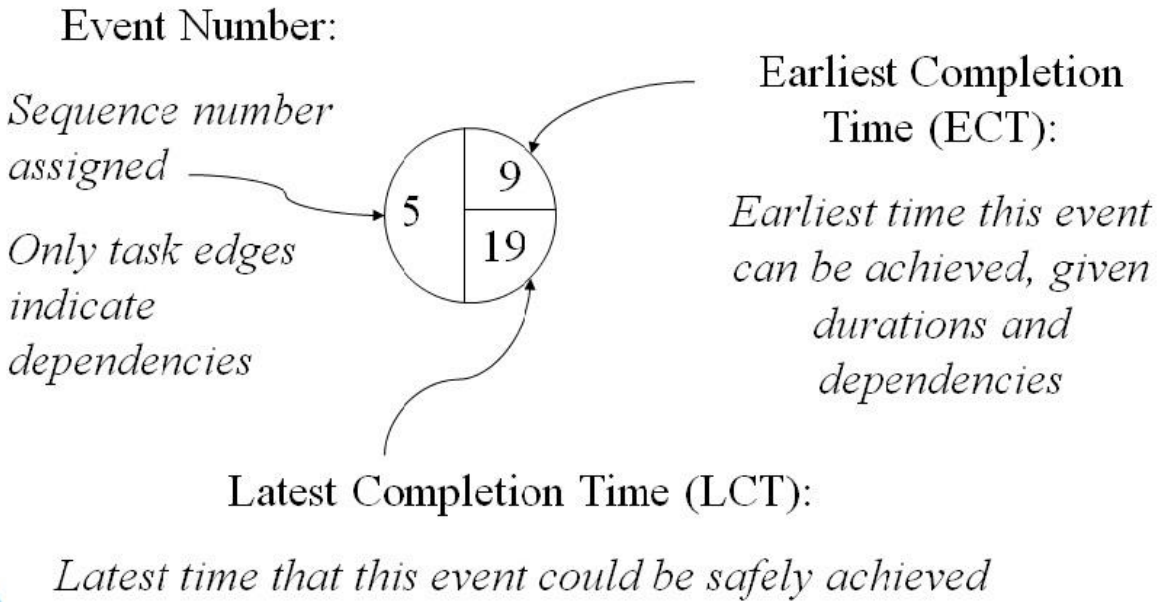
- **Task letter:**
  - Often keyed to a legend to tell which task it represents
- **Task duration = how long (e.g. days, hours) task will take**

Construct a project network





Calculate ECT and LCT



ECT Earliest completion Time

ECT = earliest time event can be completed

To calculate:

- For an event not depending on others: ECT = 0

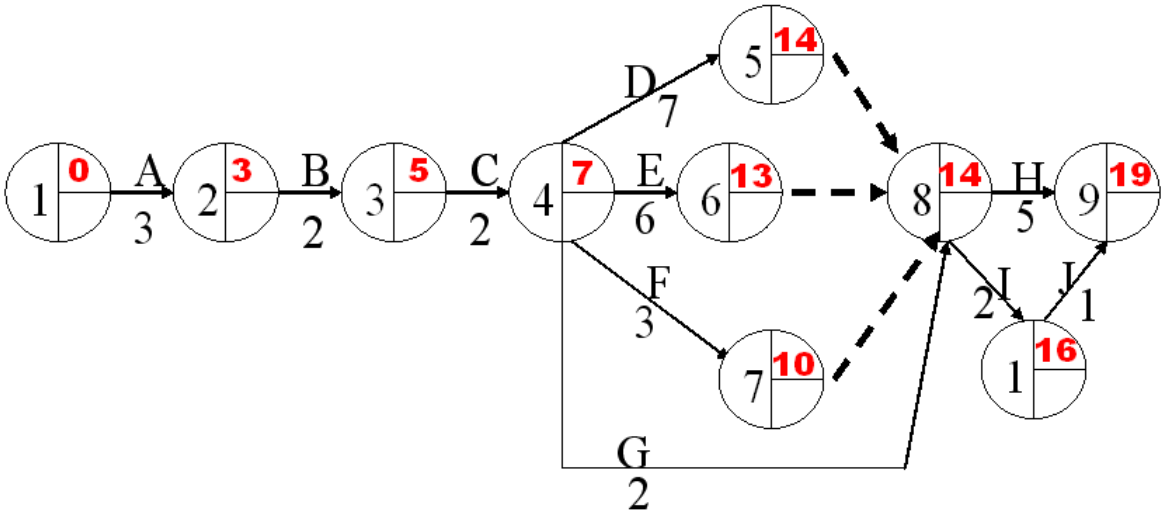
Usually this is the first event

- For an event E depending on one or more others:

Calculate ECTs of event(s) that E depends on
Add duration(s) of task(s) leading to E
If E depends on more than one event, take MAX

Proceed left to right (→) through the chart

Exercise: calculate the ECT for our example.



LCT Latest Completion Time

LCT = latest time event can be completed, while still finishing last ask at indicated time

To calculate:

- For an event which no other events depend on: LCT = ECT

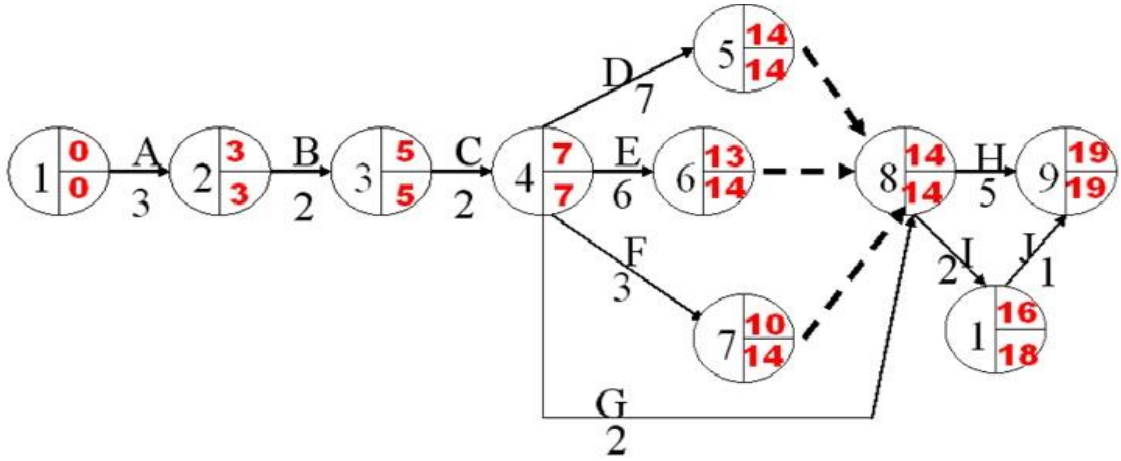
Generally there will only be one such event

- For an event E which one or more others depend on:

Calculate LCTs of event(s) that depend on E
Subtract duration(s) of task(s) leading from E
If more than one event depends on E, take **MINIMUM**

Proceed right to left (←) through PERT chart

Exercise: calculate LCT for our example



Find Critical Path

Critical Path: **Path through chart such that if any deadline slips, the final deadline slips (where all events have ECT = LCT (usually there is only one)**

In software example:

- Task I is not on the critical path: even if we don't finish it until time 18, we're still okay
- Task D is on the critical path: if we don't finish it until for example, time 16, then:

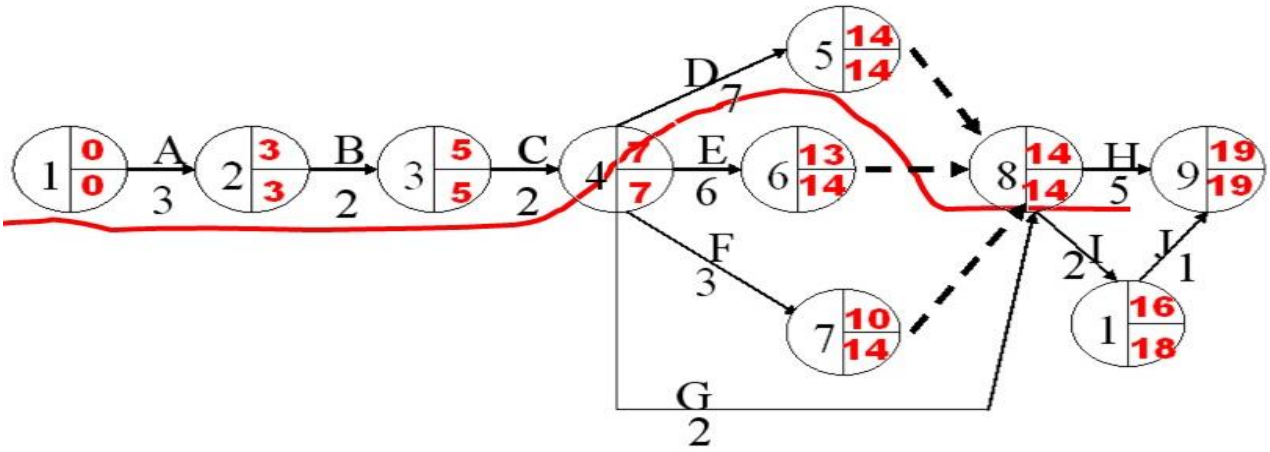
We can't start task H (duration 3) until time 16

So we can't complete task H until time 21

We can use PERT charts for

- Identifying the critical path
- Reallocating resources, e.g. from non-critical to critical tasks.

Conditions :1. Select nodes where ECT=LCT



CPM= 1-2-3-4-5-8-9

=3+2+2+7+5=19

Ex-

Task	Prec Tasks	Description	Time(hrs)
A	none	decide on date for party	1
B	A	book bouncy castle	1
C	A	send invitations	4
D	C	receive replies	7
E	D	buy toys and balloons	1
F	D	buy food	3
G	E	blow up balloons	2
H	F	make food	1
I	H, G	decorate	1
J	B	get bouncy castle	1
K	J, I	have party	1
L	K	clean up	4
M	K	send back bouncy castle	1
N	L	send thank you letters	3
O	M	donate unwanted gifts	3

For given data of PERT,

1.Construct system network

2.Find values of ECT and LCTs

3.Find the CPM

Chapter 4

REQUIREMENTS MODELING

Requirement Modeling Strategies

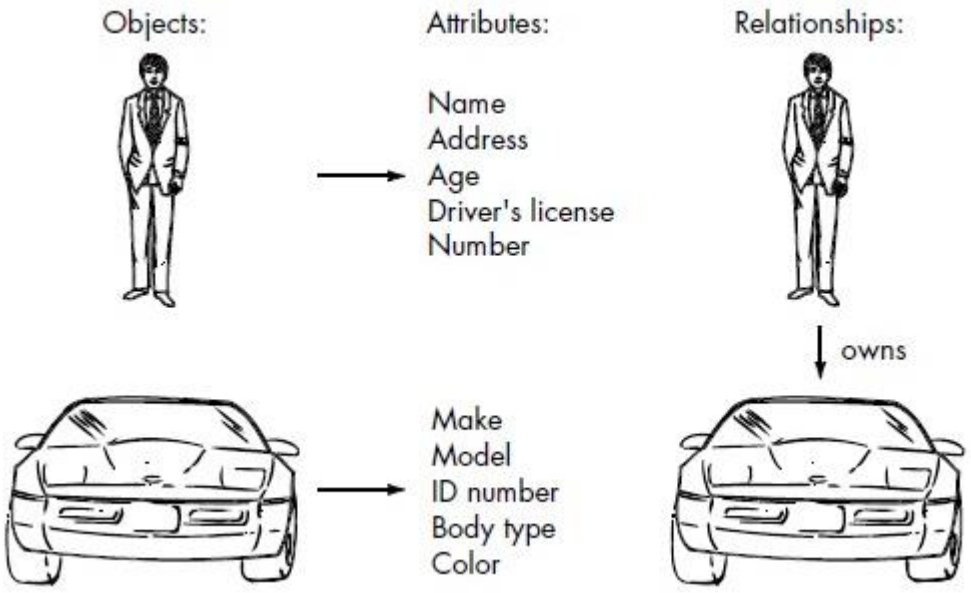
- The requirements model has many different dimensions about flow-oriented models, behavioural models, and the special requirements analysis considerations that come into play when WebApps are developed
- One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities.
- Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
- A second approach to analysis modeled, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

Data Modeling

- Very useful in data processing application.
- Helpful in in identifying the primary data objects to be processed by the system.
- Helpful in the composition of each data object and their attributes, and their relationships between

Data Objects, Attributes, and Relationships

- The data model consists of three interrelated pieces of information: the data object, the and the relationships objects to one another



Data objects.

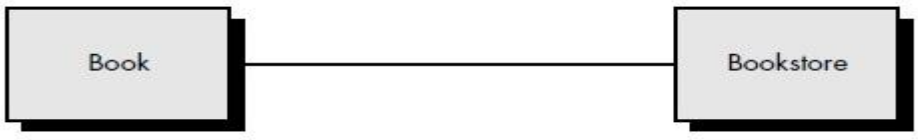
- A data object can be an external entity, a thing, an occurrence or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- For example, a person or a car
- Data objects (represented in bold) are related to one another. For example, person can own car, where the relationship own

Attributes.

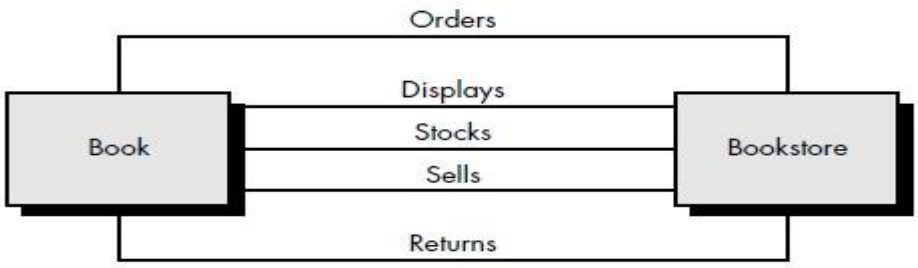
- Attributes define the properties of a data object and take on one of three
- They can be used to (1) name an instance of the data object,
  (2) describe the instance, or
  (3) make reference to another instance in another table.
- Example: The attributes for car might
  ID number,
  body type and color,
  interior code,
  transmission type, would have to be added more

Relationships.

- Data objects are connected to one another in different ways. Consider
- two data objects, book and bookstore. These objects can be represented using the simple notation illustrated in Figure
- A connection is established between book and bookstore because the two objects are related. But what are the relationships?
  • A bookstore orders books.
  • A bookstore displays books.
  • A bookstore stocks books.
  • A bookstore sells books.
  • A bookstore returns books.



(a) A basic connection between objects



(b) Relationships between objects

Cardinality

- how many occurrences of object X are related to how many occurrences of objectY. This leads to a data modeling concept called cardinality.
- Cardinality is the specification of the number of occurrences of one object that can be related to the number of occurrences of another object.
- Cardinality is usually expressed as **simply 'one' or 'many".**
    - **One-to-one (l:l)—**

An occurrence of object'A' can relate to one and only one occurrence of object 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'

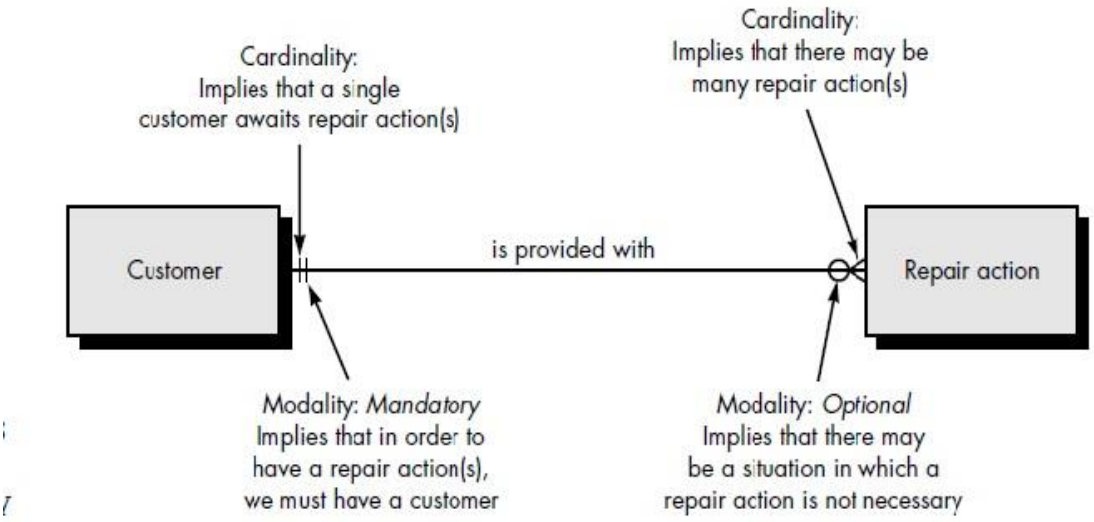
- Example, a husband have a wife

- **One-to-many (l:N)—**One occurrence of object 'A' can relate to one or many occurrences of 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'
    - For example, a mother can have many children, but a child can have only one mother.

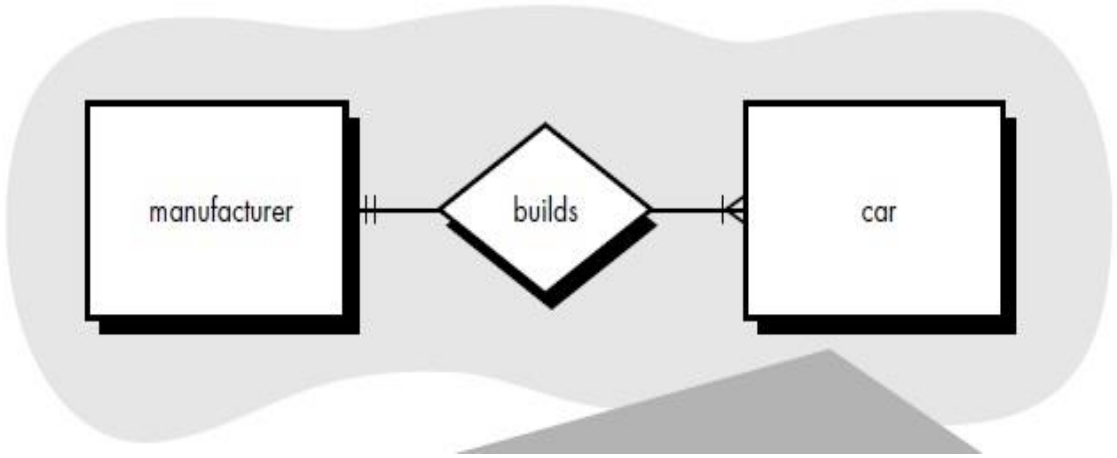
- **Many-to-many (M:N)—**An occurrence of 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'
    - For example, an uncle can have many nephews, while a nephew can have many uncles.

Modality.

- The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.
- The modality is 1 if an occurrence of the relationship is mandatory.
- To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required.



Ex: person – hobby (perform, Have)
Student – book ,
Student- course
CUSTOMER- ORDER
Godown –stores- item
Customer –purchase- item
Prepare ERD for Railway seat booking system, showing cardinality and modality



Cardinality:
1 manufacturer can builds many cars
1 car can be build by 1 manufacturer

Modality:
Builds-> mandatory

Entity/Relationship Diagrams

- ERD comprises : data objects, attributes, relationships, and various type indicators.
- The primary purpose of the ERD is to represent data objects and their relationships.
- Data objects are represented by a labeled rectangle.
- Relationships are indicated with a labeled line connecting objects.
- ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality

FLOW-ORIENTED MODELING

Data Flow Diagram DFD

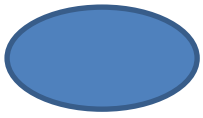
- The DFD takes an **input-process-output** view of a system.
- That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software.
- 2 horizontal parallel lines- Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles).

  The DFD is presented

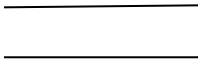
  - in a hierarchical fashion. That is, the first data flow model (sometimes called
  - a level 0 DFD or context diagram) represents the system as a whole. Subsequent data
  - flow diagrams refine the context diagram, providing increasing detail with each
  - subsequent level

Shows External Entities

Shows flow of data into system



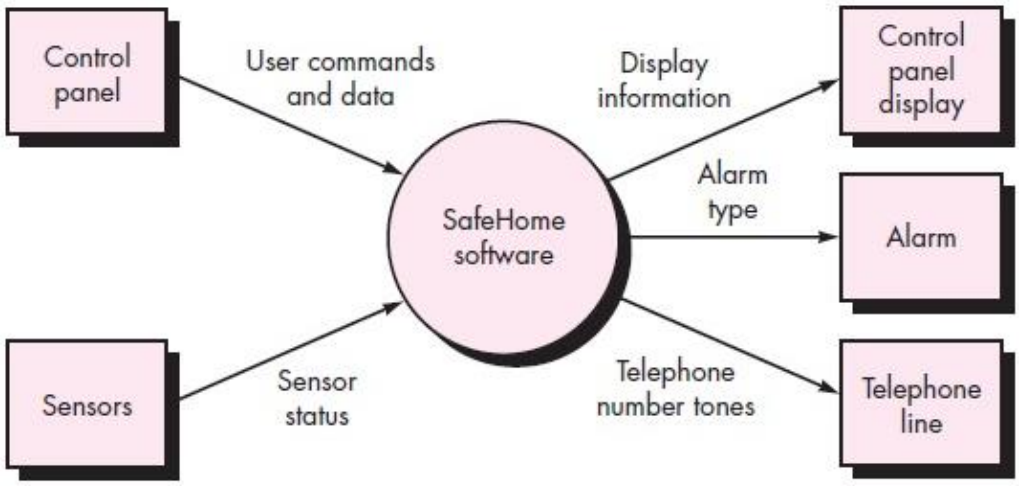
Shows, process that transform data i/p to data o/p



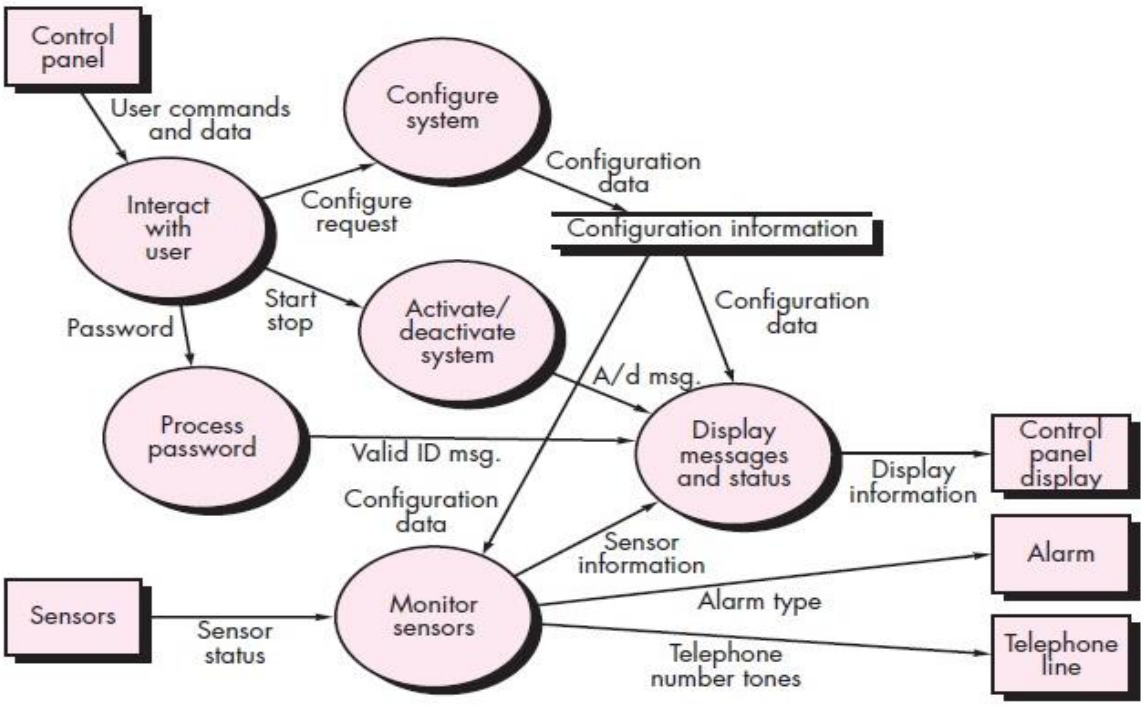
Shows Data Repository/ database

- **Level 0:** Shows Context level process
  level 0 data flow diagram shows-single bubble; primary input and output should be carefully noted;
- **Level 1:** Single process expanded with multiple processes
- **Level 2:** The processes represented at DFD level 1 can be further refined into lower levels

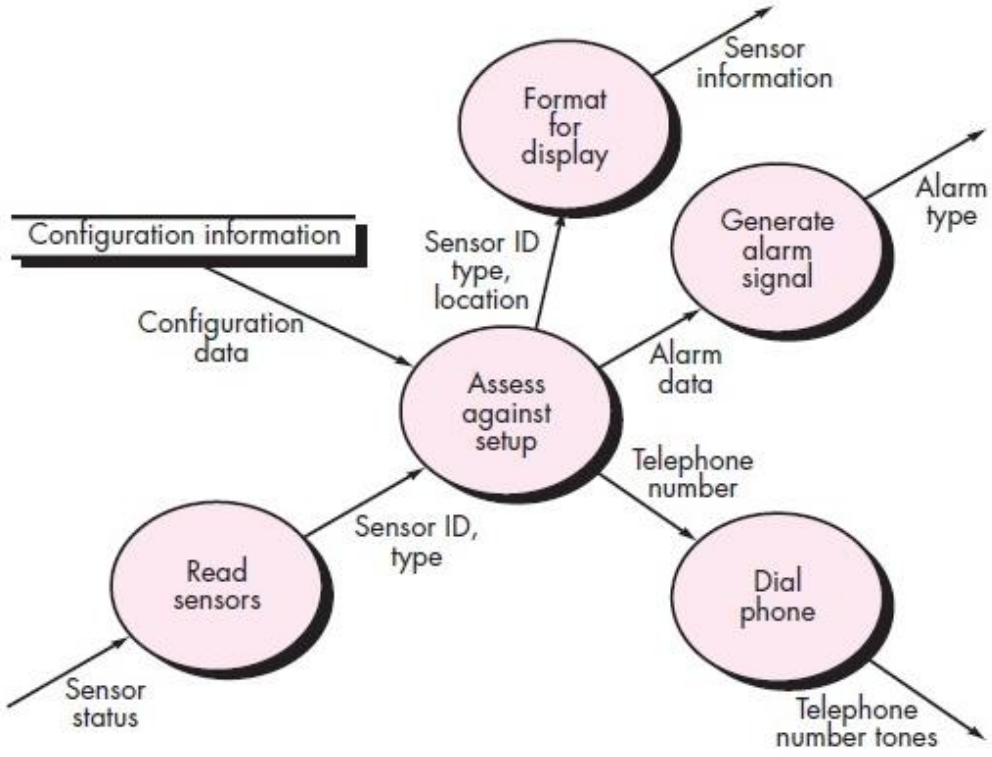
Level 0



Level 01



Level 02



Examples:1. Draw a DFD for ATM system

2. Draw a DFD for Online word to pdf converter system

3. Draw a DFD for ONLINE SHOPPING SYSTEM

4. Draw DFD for Water wending machine

Control Flow Model

For some types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements.

Such applications require the use of control flow modeling in addition to **data flow modeling**

An event or control item is implemented as a Boolean value.

The following guidelines are for select potential candidate events:

- List all sensors that are “read” by the software.
- List all interrupt conditions.
- List all “switches” that are actuated by an operator.
- List all data conditions.

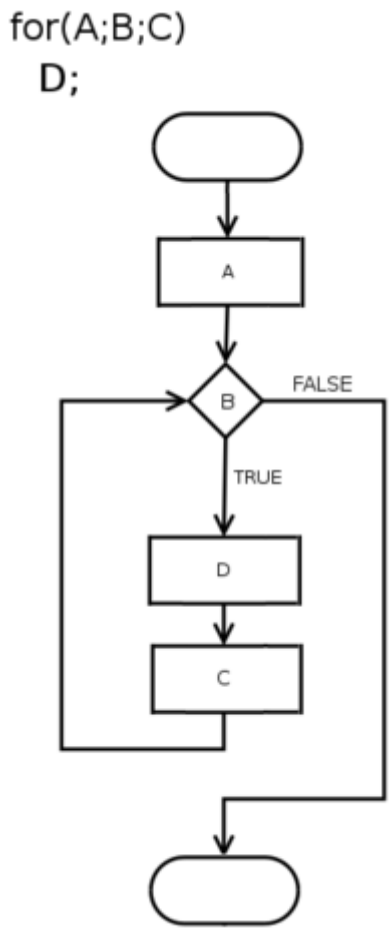
· Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible control specification inputs/outputs.

· Describe the behaviour of a system by identifying its states, identify how each state is reached, and define the transitions between states.

· Focus on possible omissions—a very common error in specifying control; for example, ask: "Is there any other way I can get to this state or exit from it?"

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Continuation at a different statement (unconditional branch or jump)
- Executing a set of statements only if some condition is met (choice - i.e., conditional branch)
- Executing a set of statements zero or more times, until some condition is met (i.e., loop - the same as conditional branch)
- Executing a set of distant statements, after which the flow of control usually returns (subroutines, coroutines, and continuations)
- Stopping the program, preventing any further execution (unconditional halt)



The Control Specification

A **control specification** (CSPEC) represents the **behaviour of the system** in two different ways.

The CSPEC contains a **state diagram** that is a sequential specification of behaviour. It can also contain a **program activation table** a **combinatorial** specification of behaviour.

A somewhat different mode of behavioural representation is the process activation table. The PAT represents information contained in the state diagram in the context of processes, not states.

That is, the table indicates which processes (bubbles) in the flow model will be invoked when an event occurs.

The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level.

The CSPEC describes the behaviour of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behaviour.

The Process Specification

The **process specification** (PSPEC) is used to describe all flow model processes that appear at the **final level of refinement**.

The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, or UML activity diagrams.

By providing a PSPEC to accompany each bubble in the flow model, you can create a "mini-spec" that serves as a guide for design of the software component that will implement the bubble.

Creating a Behavioural Model

- To make a transition to the **dynamic behaviour** of the system. represent the behaviour of the system as a function of specific events and time.
- The ***behavioural model*** **indicates how software will respond to external events or external stimuli**. To create the model, should perform the following steps:

1. **Evaluate all use cases** to fully understand the sequence of interaction
   Within the system.

2. **Identify events** that drive the interaction sequence and understand how These events relate to specific objects.
3. **Create a sequence for each use case**.
4. **Build a state diagram** for the system.
5. **Review the behavioural model to verify accuracy and consistency**.

Each of these steps is discussed in the sections that follow.

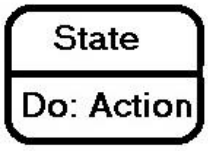
Identifying Events with the Use Case

- The use case represents a sequence of activities that involves
- actors and the system. In general, an event occurs whenever the system and an actor exchange information.
- A use case is examined for points of information exchange.
- The underlined portions of the use case scenario indicate events.
- An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

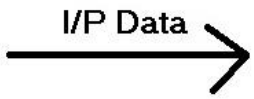
State Diagram

-Start

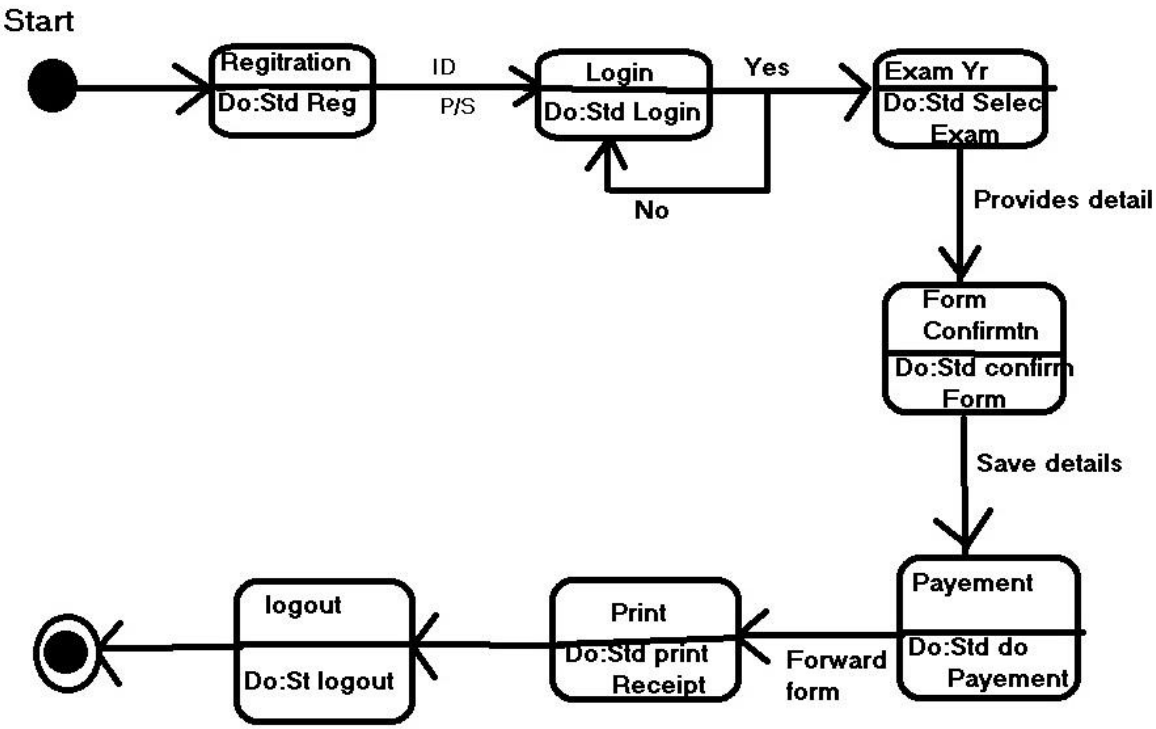
-Stop



-State



-Input data.



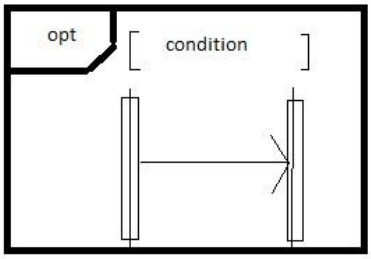
Ex: Prepare state diagram for

1. **New email account opening**
2. **Computer hardware assembling**

Sequence Diagram (Notations)

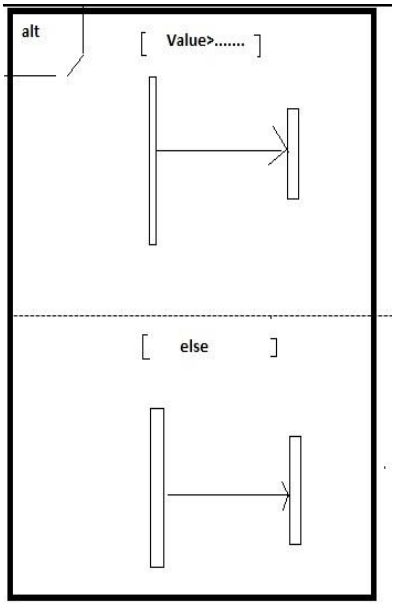
Object/Actor

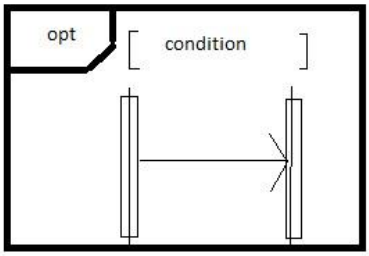
Active on lifeline



If

If else

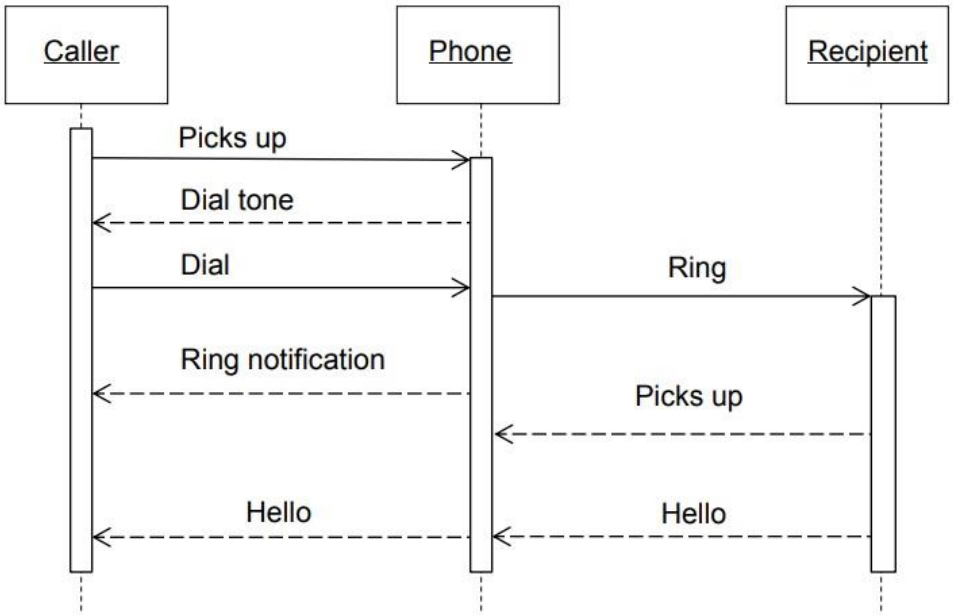




Loop

Example:

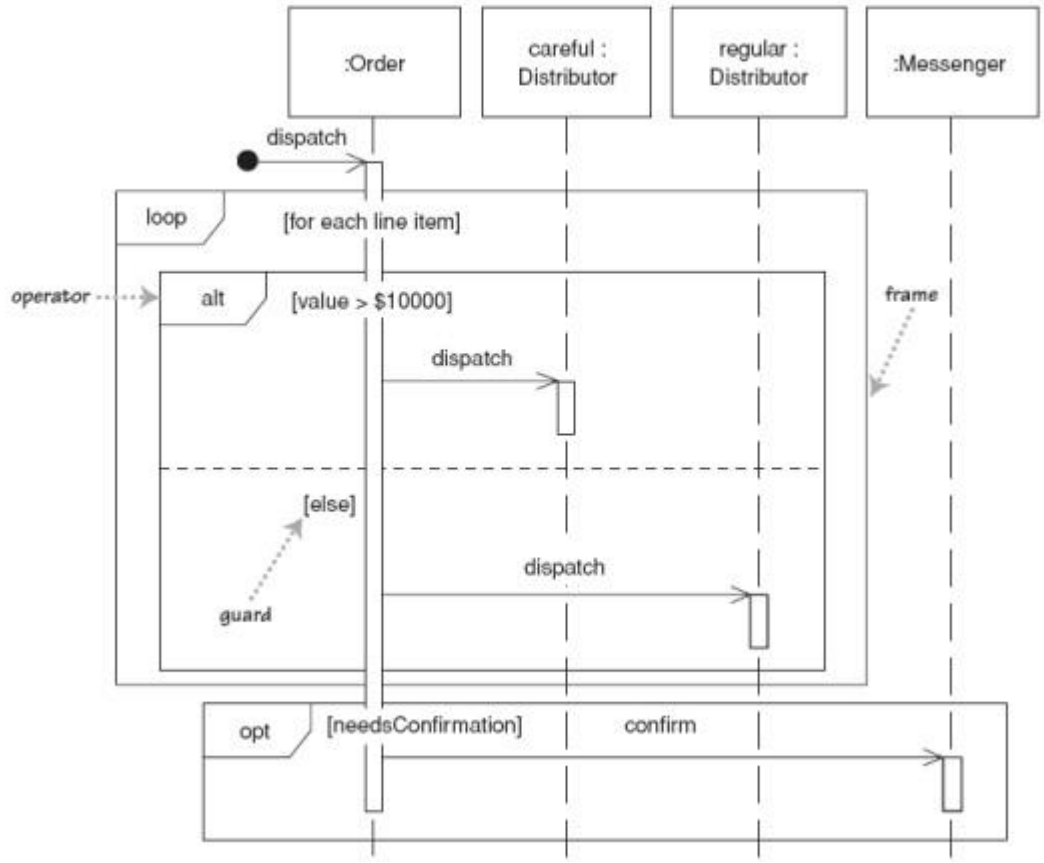
Sequence Diagram (make a phone call)



H/w

1. Client Server

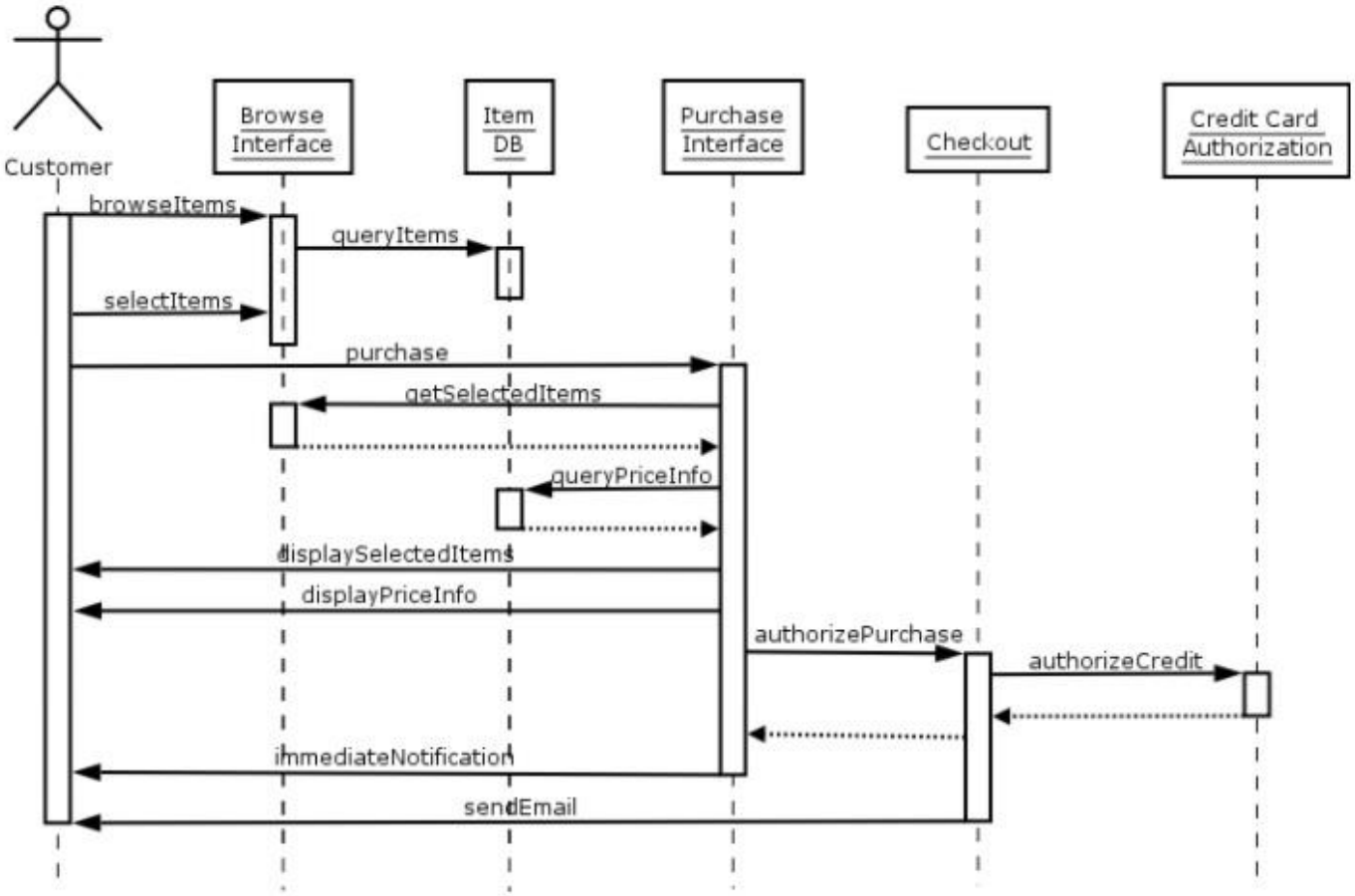
Sequence Diagram for Dispatching order and giving confirmation



H/w

1.Online Bookstore

Sequence diagram for online shopping



Sequence diagram for online shopping cloud service

Patterns for requirement Modeling

- Software patterns are a mechanism for capturing domain knowledge, that allows to be reapplied when a new problem is encountered.
- In some cases, the domain knowledge is applied to a new problem within the same application domain.
- The domain knowledge captured by a pattern can be applied to a completely different application domain.
- The original author of an analysis pattern does not "create" the pattern, but, rather, *discovers* it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented by describing "explicitly the general problem to which the pattern is applicable, then prescribes solution,
- Also, have consider the assumptions and constraints of using the pattern in practice, and often some other information about the pattern,
- Motivation and driving forces for using the pattern, discussion of the pattern's advantages and disadvantages, and references to some known examples of using that pattern in practical applications"
- The pattern can be reused when performing requirements modeling for an application within a domain.
- patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them.
- Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name

Discovering Analysis Pattern

- The requirements model is comprised of a wide variety of elements: scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioural.
- Each provides an opportunity to discover patterns that may occur throughout an application domain
- The most basic element in the description of a requirements model is the use case.
- A semantic analysis pattern (SAP) is a pattern that describes a small set of coherent use cases
- This use case implies a variety of functionality that would be refined and elaborated into a coherent set of use cases during requirements gathering and modeling.
- Regardless of how much elaboration is accomplished, the use cases suggest a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system.

A Requirements Pattern Example: Actuator-Sensor

- By considering One of the requirements of the Safe Home security function is the ability to monitor security sensors
- Internet-based extensions to Safe Home will require the ability to control the movement of a security camera within a residence.
- Actuator-Sensor that provides useful guidance for modeling this requirement within Safe Home software.
- An abbreviated version of the Actuator-Sensor pattern,
- originally developed for automotive applications, follows.

  Pattern Name. Actuator-Sensor

- **Intent**: Specify various kinds of sensors and actuators in an embedded system

- **Motivation**.

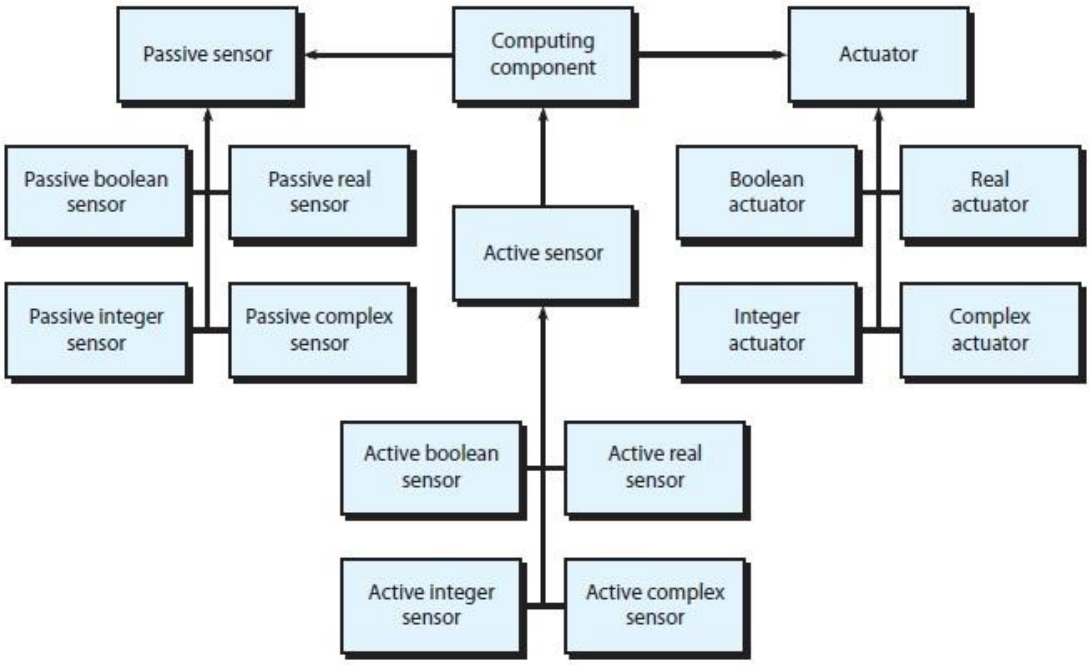
  Embedded systems usually have various kinds of sensors and actuators.
  These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different their behaviour is similar enough to structure them into a pattern.
  The pattern shows how to specify the sensors and actuators for a system, including attributes and operations.
  The Actuator-Sensor pattern uses a pull mechanism for Passive Sensors and a push mechanism for the Active Sensors.

- **Constraints**
  - Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
  - Each active sensor must have capabilities to broadcast update messages when its value changes.
  - Each active sensor should send a life tick, a status message issued within a specified time frame, to detect malfunctions.
  - Each actuator must have some method to invoke the appropriate response determined by the Computing Component.
  - Each sensor and actuator should have a function implemented to check its own operation state.
  - Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

- **Applicability.** Useful in any system in which multiple sensors and actuators are present.

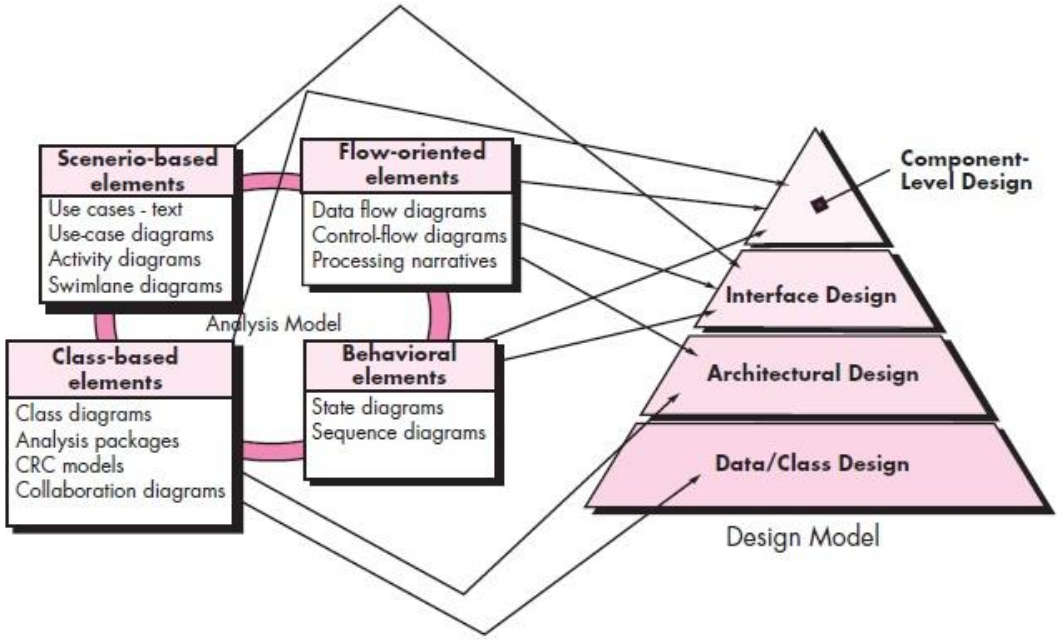
- **Structure**.

- Actuator, Passive Sensor, and Active Sensor are abstract classes and denoted in italics.
- There are four different types of sensors and actuators in this pattern.
  The Boolean,
  Integer and
  Real classes represent the most common types
- The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device.
- Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.



DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- Beginning once software requirements have been analyzed and modeled, **software design is the last software engineering action within the modeling activity and sets the stage for construction**



- Each of the elements of the **requirements model provides information that is necessary to create the four** design models required for a complete specification of design.

- The requirements model, **manifested by scenario-based, class-based,**
  flow-oriented, and behavioral elements, feed the design task.

- Using design notation and design methods, **design produces a data/class design, an architectural design, an interface design, and a component design**.

- The data/class design **transforms class models into design class** realizations and the requisite data structures required to implement the software.

- Part of **class design may occur in conjunction with the design of software architecture**.

- More d**etailed class design occurs as each software component is designed**.

- The architectural design **defines the relationship between major structural elements of the software**, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

- The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.

Design Process

- **Software design is an iterative process through which requirements are translated into a "blueprint**" for constructing the software.

- Initially, **it depicts a holistic view of software means** design is represented at a **high level of abstraction—**

- A level that can be directly traced to the specific system objective and more **detailed data, functional, and behavioural** requirements.

- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

Software Quality Guidelines and Attributes

Quality Guidelines.

In order to evaluate the quality of a design representation, **members of the software team must establish technical criteria for good design**.

1. A design should **exhibit an architecture** that

(1) has been created **using recognizable architectural styles** or patterns, (2) is **composed of components that exhibit good design** characteristics

(3) can be **implemented in an evolutionary fashion**,2 thereby facilitating implementation and testing.

2. A **design should be modular**; that is, the software should be logically **partitioned into** elements or subsystems.

3. A **design should contain distinct representations of data**, architecture,
interfaces, and components.

4. A **design should lead to data structures that are appropriate for the classes** to be implemented and are drawn from recognizable data patterns.

5. A design should **lead to components that exhibit independent functional**
characteristics.

6. A design should **lead to interfaces that reduce the complexity** of connections between components and with the external environment.

7. A design **should be derived using a repeatable method** that is driven by information obtained during software requirements analysis.

8. A design **should be represented using a notation that effectively communicates** its meaning.

Quality Attributes.

Set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**.

- ***Functionality*** is assessed by evaluating the **feature set and capabilities** of the
program, the generality of the functions that are delivered, and the security of
the overall system

- ***Usability*** is assessed by **considering human factors** overall aesthetics, consistency, and documentation.

- *R**eliability*** **is evaluated by measuring the frequency and severity of failure**, the accuracy of output results, the mean-time-to-failure (MTTF), **the ability to**
recover from failure, and the predictability of the program.

- ***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—these three attributes represent a more common term, ***maintainability***—and in addition, testability, compatibility, configurability

The Evolution of Software Design

- The **evolution of software design is a continuing process** that has now spanned almost six decades.

- Early design work **concentrated on criteria for the development of**
  modular programs and methods for refining software structures in a topdown manner

- **Procedural aspects** of design definition evolved into a philosophy called ***structured programming***

- **Later work proposed methods for the translation of data flow or data structure** into a design definition.

- **Newer design approaches proposed an object-oriented approach** to design derivation.

- **More recent emphasis in software design has been on software architecture and the design patterns** that can be used to implement software architectures and lower levels of design abstractions.

- Growing emphasis on **aspect-oriented methods, model-driven development , and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created**.

- A number of design methods, are being applied throughout the industry.

  Ex: (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.**

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering.

Abstraction

- **When we consider a modular solution to any problem, many levels of abstraction can be posed**.

- At the **highest level of abstraction, a solution is stated in broad terms**
  using the language of the problem environment.

- **At lower levels of abstraction, a more detailed description** of the solution is provided.

- As **different levels of abstraction** are developed, it have work to create **both procedural and data abstractions**.

- A ***procedural abstraction*** **refers to a sequence of instructions that have a specific and limited function**.
    - **The name of a procedural abstraction implies these functions, but specific details are suppressed**.
    - An **example** of a procedural abstraction would be the word ***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.)**

- **A *data abstraction*** is a named collection of data that describes a data object.

- In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

- It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

Architecture

- *Software architecture* means to "**the overall structure of the software and the ways in which that structure provides conceptual integrity for a system**"

- Architecture is the structure or organization of program components (modules**), the manner in which these components interact, and the structure of data that are used by the components**.

- components can be **generalized to represent major system elements and their interactions.**

- **One goal of software design is to derive an architectural version of a system**. **This version serves as a framework from which more detailed design activities are conducted**.

- **A set of architectural patterns enables a software engineer to solve**
  common design problems.

- A set of **properties** that should be specified as part of an architectural design:
  - **Structural properties**
  - **Extra-functional properties**
  - **Families of related systems**

.

- **He architectural design can be represented using various models**
  - **Structural Models**
  - **Framework model**
  - **Dynamic models**
  - **Process Model**

- **Architectural Discretional Languages (ADLs) have been developed**
  to represent these models

- Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another

Patterns

- A pattern conveys the essence of **a proven solution to a recurring problem** within a certain context

- Stated in another way, a design pattern **describes a design structure that solves a particular design problem within a specific context**

- It **enables a designer to determine**
  (1) **whether the pattern is applicable to the current work**,
  (2) **whether the pattern can be reused** (hence, saving design time), and

(3) **whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern**

Separation of Concerns

- Separation of concerns is a design concept, **suggests that any complex problem can be more easily handled if it is subdivided into pieces** that can each be solved and/or optimized 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, and therefore more manageable pieces, a problem takes less effort and time to solve**.

- Ex:For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2.

- As by using this, **it does take more time to solve a difficult problem**.

- It also follows that **the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately**.

- It is nothing but leads to **a divide-and-conquer strategy**—it's easier to solve a complex problem when you break it into manageable pieces.

- This has important implications with regard to software modularity.

- Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement.

Modularity

- Modularity **is the most common manifestation of separation of concerns**.

- **Software is divided into separately named and addressable components**, sometimes called modules, that are integrated to satisfy problem requirements.

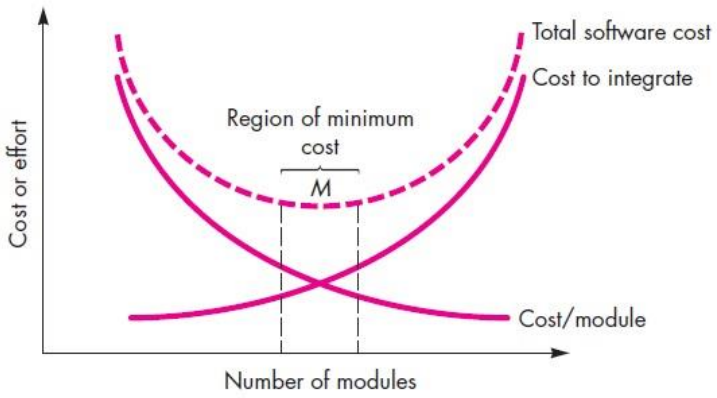
- It has been stated that **"modularity is the single attribute of software that allows a program to be intellectually manageable"**.

- **Monolithic software (a single module) cannot be easily grasped by a software engineer. complexity would make understanding close to impossible**.

- **In almost all instances, you should break the design into many modules**, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

- **If subdivide software indefinitely the effort required to develop it will become negligibly small**!

- Referring to Figure the effort (cost) to develop an individual software module does decrease as the total number of modules increases.



Modularity and software cost

- **However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.**

Information Hiding

- The principle of **information hiding suggests that modules be "characterized by design decisions that each hides from all others**.

- The **modules should be specified and designed so that information contained within a module is inaccessible to other modules** that have no need for such information.

- Hiding implies that **effective modularity can be achieved by defining a set of independent modules** that communicate with one another only that information necessary to achieve software function.

- **Abstraction helps to define the procedural (or informational) entities that make up the software**.

- **Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module**

- The use of **information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance**.

Functional Independence

- The concept of functional independence **is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding**.

- **Functional independence is achieved by developing modules with single minded function and an avoiding excessive interaction with other modules**.

- **Design software so that each module addresses a specific subset of requirements** and has a simple interface when viewed from other parts of the program structure.

- **Software with independent modules, is easier to develop because** function can be compartmentalized and **interfaces are simplified**

- **Independent modules are easier to maintain because secondary effects caused by design or code modification are limited**

- **To summarize, functional independence is a key to good design,** and design is the key to software quality.

- **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**.

Refinement(decomposition)

- **Stepwise refinement is a top-down design** strategy
- A **program is developed by successively refining levels** of procedural
  detail.
- A **hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached**.
- Refinement **is actually a process of *elaboration*.**
- The statement **describes function conceptually but provides no information about the internal workings of the function**s
- **Elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs**.
- **Abstraction and refinement are complementary to the each other**.
- Refinement helps to reveal low-level details as design progresses.

Aspects

- **As design begins, requirements are refined into a modular design representation**.
- **It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur**.
- In an ideal context, **an aspect is implemented as a separate module rather than as software fragments that are "scattered" or "tangled" throughout many components**.
- To accomplish this, the design architecture should support a mechanism for defining an aspect
- A module that enables the concern to be implemented across all other concerns that it crosscuts

Refactoring

- Refactoring is **a reorganization technique that simplifies the design of a component without changing its function or behavior**.

- Fowler defines refactoring in the following manner "**Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."**

- **When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms**, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

- **For example, a first design iteration might yield a component that exhibits low cohesion** .

- **After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion**.

- **The result will be software that is easier to integrate, easier to test, and easier to maintain**.

- Intent of refactoring is to modify the code in a manner that does not alter its external behavior.

Object-Oriented Design Concepts and Design Classes

- The object-oriented (OO) paradigm **is widely used in modern software engineering.**

- It has been **provided with design concepts such as classes and objects, inheritance, messages, and polymorphism, among others**

- **The requirements model defines a set of analysis classes**.

- **Each describes some element of the problem domain, focusing on aspects of the problem that are user visible**.

- **The level of abstraction of an analysis class is relatively high.**

Five different types of design classes, each representing a different layer of the
design architecture,

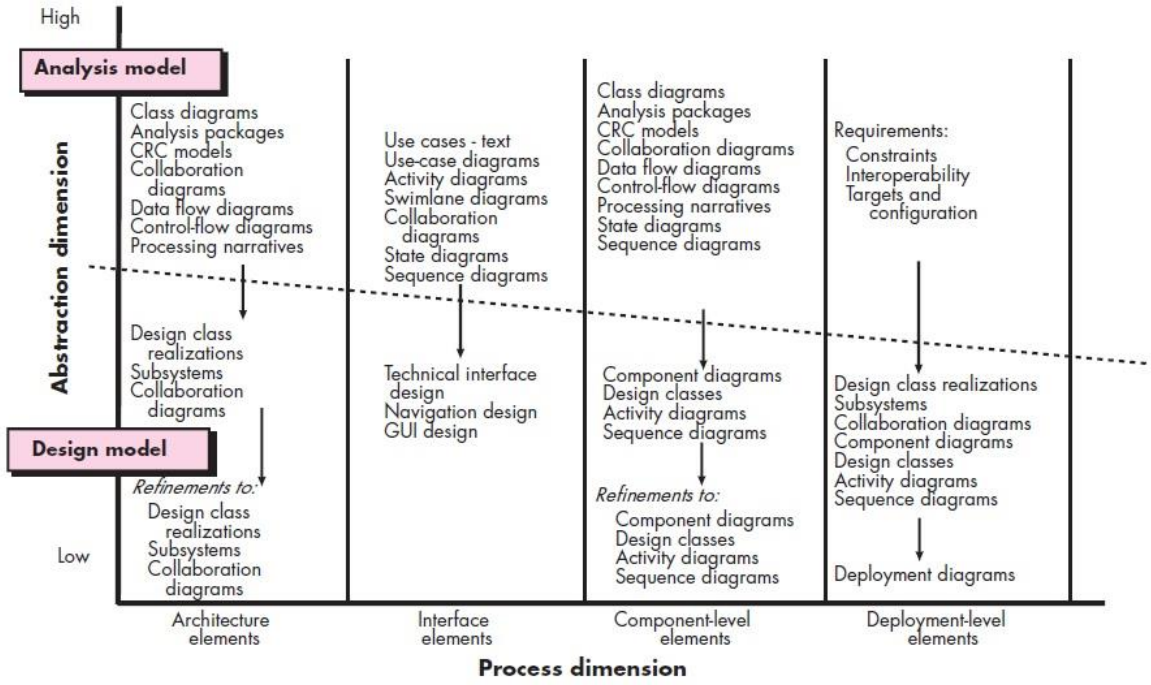
- ***User interface classes*** define all abstractions that are necessary for **human computer interaction (HCI)**. In many cases, HCI occurs within the context of
- ***Business domain classes*** are often refinements of the analysis classes defined earlier. **The classes identify the attributes and services (methods) that are required to implement some element of the business domain**.
- ***Process classes*** **implement lower-level 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 with the outside world**

  Characteristics of a design classes
  - ✓ **Complete and sufficient**
  - ✓ **Primitiveness**
  - ✓ **High cohesion**
  - ✓ **Low coupling**

Design Model

- The design model can be viewed in two different dimensions
- The ***process dimension*** **indicates the evolution of the design model as design tasks are executed as part of the software process**.
- The ***abstraction dimension*** **represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively**. As shown fig
- The **dashed line** indicates the boundary between the analysis and design models.
- **In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious**.
- **The elements of the design model use many of the same UML diagrams that were used in the analysis model**.
- The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided,



Data Design Elements:

- Like other software engineering activities, **data design creates a model of data and/or information that is represented at a high level of abstraction**

- This data model refined into progressively more implementation-specific representations that can be processed by the computer-based system.

- In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

- **The structure of data has always been an important part of software design**.

- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

- **At the application level, the translation of a data model-into a database is pivotal to achieving the business objectives of a system**.

- At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining
  or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements

- The architectural design for software is the equivalent to the **floor plan of a house.**
- The floor plan **depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms**.

- The floor plan gives us **an overall view of the house**. Architectural design **elements give us an overall view of the software**.

- The architectural model **is derived from three sources**:
  (1) **information** about the application domain for the software to be built;
  (2) specific **requirements model elements** such as data flow diagrams or
    analysis classes, their relationships and collaborations for the problem at hand; and
  (3) **the availability of architectural styles and patterns**

- **The architectural design element is usually depicted as a set of interconnected subsystems**, often derived from analysis packages within the requirements model.

- **Each subsystem may have it's own architecture** (e.g., a graphical user interface might be structured according to a pre existing architectural style for user interfaces).

Interface Design Elements

- **The interface design for software is analogous to a set of detailed drawings**.
  Example: Interface design for House, a set of detailed drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone)come into the house and are distributed among the rooms depicted in the floor plan.

- **The interface design elements for software depict information flows into and out of the system** and how it is communicated among the components defined as part of the architecture.

- There are **three important elements** of **interface design**:
  (1) the **user interface (UI);**
  (2) **external interfaces to other systems**, devices, networks, or other

producers or consumers of information; and
(3) **internal interfaces between various design components**.

- **It allows the software to communicate externally and enable internal communication and collaboration among the components**

- UI design (increasingly called usability design) is a major software engineering action, the

- Usability design **incorporates aesthetic elements e.g., layout, color, graphics, interaction mechanisms**,

- Usability design incorporates **ergonomic elements** e.g., information layout and placement, metaphors, UI navigation,

- Usability design incorporates **technical elements** (e.g., UI patterns, reusable components).

- In general, the UI is a unique subsystem within the overall application architecture.

- The design of external interfaces requires definitive information about the entity to which information is sent or received.

- In every case, this information should be collected during requirements engineering and verified once the interface design commences.

- The design of external interfaces **should incorporate error checking and some security features**.

- The **design of internal interfaces is closely aligned with component-level design**

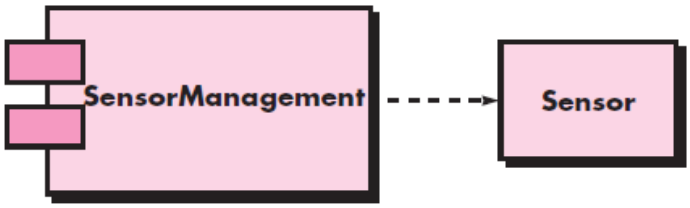
- Design realizations of analysis classes represent all operations and the
messaging schemes required to enable communication and collaboration
between operations in various classes.

Component-Level Design Elements

The component-level design for software fully describes the internal detail of each software component.

the component-level design **defines data structures for all local data objects** and algorithmic detail

Within the context of object-oriented software engineering, a component is represented in UML diagramatic form



In this figure, a component named Sensor Management (part of the SafeHome security function) is represented

A dashed arrow connects the component to a class named Sensor that is assigned to it

The design details of a component can be modeled at many different levels of abstraction

A **UML activity diagram can be used to represent processing logic**

Detailed procedural flow for a component or some other diagrammatic form e.g., flowchart or box diagram

Data structures, selected based on the nature of the data objects to be processed, are usually modeled using the programming language to be used for implementation

> **Example-** The component-level design for software is the equivalent to a set of detailed drawings and specifications for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the
> flooring to be used, the moldings to be applied, and every other detail associated with a room.

Deployment- Level Design Elements

- It **Indicates, how software functionality and subsystems will be allocated within the physical computing environment that will support the software**.

- For example, the elements of the any software product are configured
  to operate within three primary computing environments

- During design, a UML deployment diagram is developed and then refined as Subsystems (functionality) housed within each computing element

- An external access subsystem has been designed to manage all attempts to access the main system from an external source.

- Each subsystem would be elaborated to indicate the components that it implements.

- **The deployment diagram shows the computing environment but does not explicitly indicate configuration details**.

- **These details are provided when the deployment diagram is revisited in instance form during the latter stages of design or as construction begins**.

- **Each instance of the deployment is identified**

DESIGNING CLASS-BASED COMPONENTS

Basic Design Principles

- Four basic design principles are applicable to component-level design and have been widely adopted

- These principles can use as a guide as each software component is developed.

The Open-Closed Principle (OCP).

- "A **module should be open for extension but closed for modification**" This represents one of the most important characteristics of a good component-level design.

- Specify the component in a way that **allows it to be extended without**
  the need to make internal modifications to the component itself.

- Create abstractions that serve as a buffer between the functionality and the design class itself.

The Liskov Substitution Principle (LSP).

- "**Subclasses should be substitutable for their base classes**"

- This design principle, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

- LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it.

- A **precondition that must be true before the component uses a base class**

- A **postcondition that should be true after the component uses a base class**.

- To create derived classes, be sure they conform to the pre- and postconditions.

Dependency Inversion Principle (DIP).

- "**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**.

The Interface Segregation Principle (ISP).

- "**Many client-specific interfaces are better than one general purpose interface**" There are many instances in which multiple client components use the operations provided by a server class.

- ISP suggests that you should create a specialized interface to serve each major category of clients.

- **Only those operations that are relevant to a particular category of clients should be specified in the interface for that client**.

- **If multiple clients require the same operations, it should be specified in each of the specialized interfaces**.

The Release Reuse Equivalency Principle (REP).

- "**The granule of reuse is the granule of release**"

- When classes or components are designed for reuse, there is an implicit contract that is established between the **developer of the reusable entity**
  and the people who will use it.

The Common Closure Principle (CCP).

- **"Classes that change together belong together."**

- Classes should be packaged cohesively.

- when classes are packaged as part of a design, they should address the same functional or behavioural area.

- When some characteristic of that area must change, it is likely that only those classes within the package will require modification.

The Common Reuse Principle (CRP).

- **“Classes that aren’t reused together should not be grouped together”**

- When one or more classes within a package changes, the release number of the package changes.

- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested

- If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

.

Component-Level Design Guidelines

Components.

- Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

- Infrastructure components level classes should be named to reflect implementation-specific meaning.

- can choose to use stereotypes to identify the nature of components at the detailed design level.
  For example, <<infrastructure>> to identify an infrastructure component,
  <<database>> could be used to identify a database
  <<table>> can be used to identify a table within a database.

Interfaces.

- Interfaces provide important information about communication and collaboration

- Interface helping us to achieve the OCP.

- However, unfettered representation of interfaces tends to complicate component diagrams.

- Interface are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance.

- For improved readability, it is a good idea to model dependencies from left to right And inheritance from bottom (derived classes) to top (base classes).

- **The component interdependencies should be represented via interfaces**,

- It will help to make the system more maintainable.

Cohesion

- cohesion as the “single-mindedness” of a component.

- **Within the context of component-level design for object-oriented systems, cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another**

Different types of cohesion

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**,

- 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**.

- Such classes focus solely on the data in question, accessing and storing it.

- Classes and components that **exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain**.

Coupling

- Coupling is a qualitative measure of the degree to which classes are connected to one another.

- **As classes become more interdependent, coupling increases**.

- **An important objective in component-level design is to keep coupling as low as is possible**

Class coupling can manifest itself in a variety of ways.

- **Content coupling**.
  It Occurs when one component modifies data that is internal to another component

- **Common coupling**.
  It occurs when a number of components all make use of a global variable.

  Although this is sometimes necessary common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

- **Control coupling**.
  It occur when operation A() invokes operation B() and passes control flag to B. The control flag then "directs" logical flow within B.

  The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes.

- **Stamp coupling**.

It occurs when ClassB is declared as a type for an argument of an operation of ClassA.

Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

- **Data coupling.**
  Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

- **Routine call coupling**.
  It occurs when one operation invokes another.

  This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

- **Type use coupling**.
  It Occurs when component A uses a data type defined in component B

  If the type definition changes, every component that uses the definition must also change.

- **Inclusion or import coupling**.
  It occurs when component A imports or includes a package or the content of component B.

- **External coupling**.
  It occurs when a component communicates or collaborates with infrastructure components

  Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Garvin's Quality Dimensions

- **David Garvin suggests** that quality should be considered by taking a multidimensional viewpoint

- Although Garvin's **eight dimensions** of quality developed for0020when softwarequality is considered:

1. **Performance quality**.
   - Software deliver all content, functions, and features that are specified as part of the requirements from end user

2. **Feature quality**.
   - Does the software provide features that as per wish of user

3. **Reliability.**
   - Does the software deliver all features and capability without Failure?
   - Is it availablewhen it is needed? .
   - Does it deliver functionality that error-free?

4. **Conformance.**
   - Does the software conform to local and external software standards that are relevant to the application?
   - Does it conform to de factdesign and coding conventions?

5. **Durability**.
   - Can the software be maintained or corrected without unintended side effects?
   - Will changes cause the error rate or reliability to degrade with time?

6. **Serviceability**.
   - Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period?
   - Can support staff acquire all information they need to make changes or correct defects?

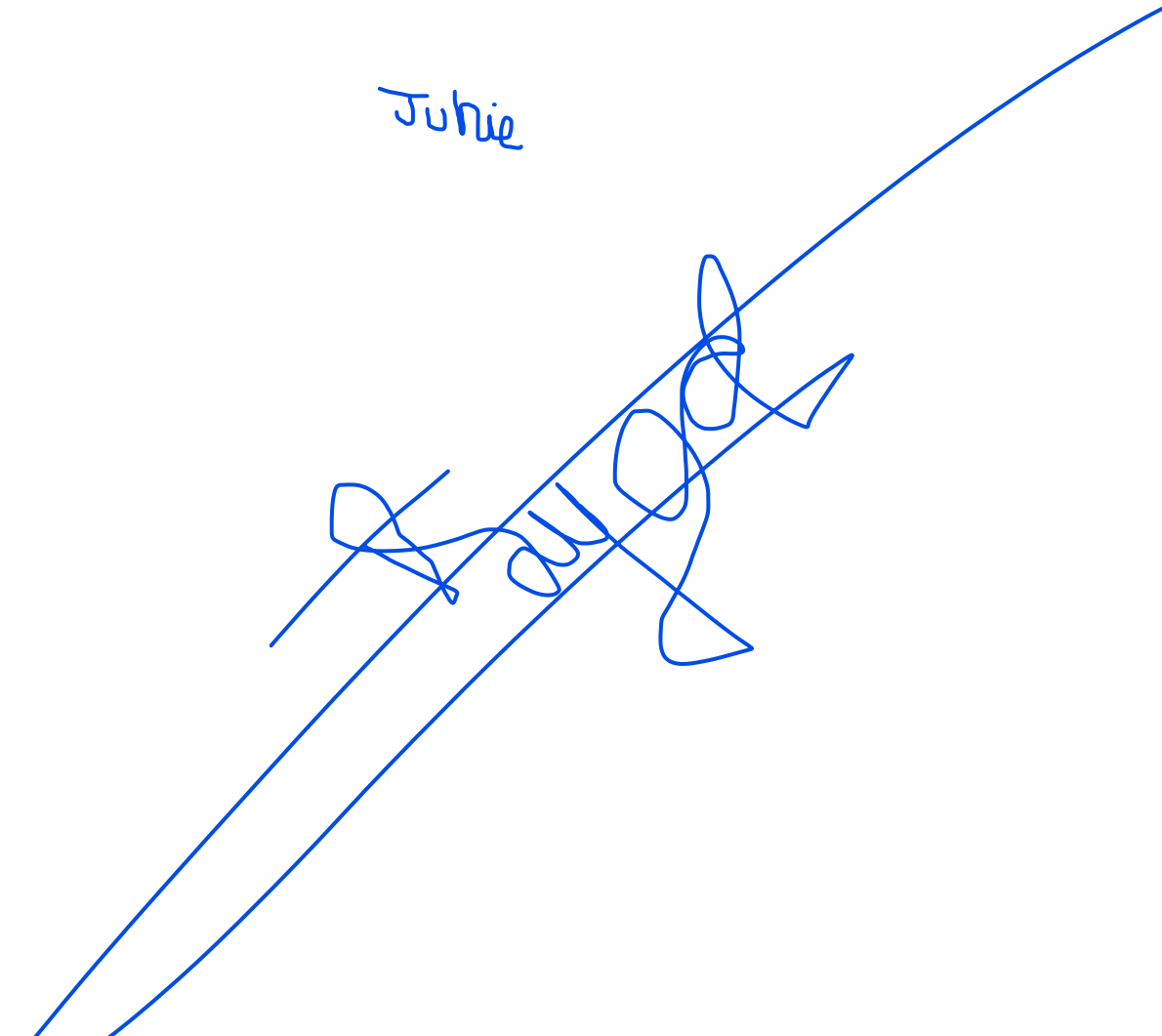
7 **Aesthetics**.

- an aesthetic entity has a certain **elegance, a unique flow**, and an obvious“presence” that are hard to quantify

8 **Perception**.

- In some situations, you have a set of prejudices that will influence your perception of quality.

  For example, if you are introduced to a software product that was built by a vendor who has produced **poor quality in the past**, your initial perception of the current software product quality might be negatively.
  Similarly, if a vendorhas an excellent reputation, you may perceive quality, even when it does notreally exist



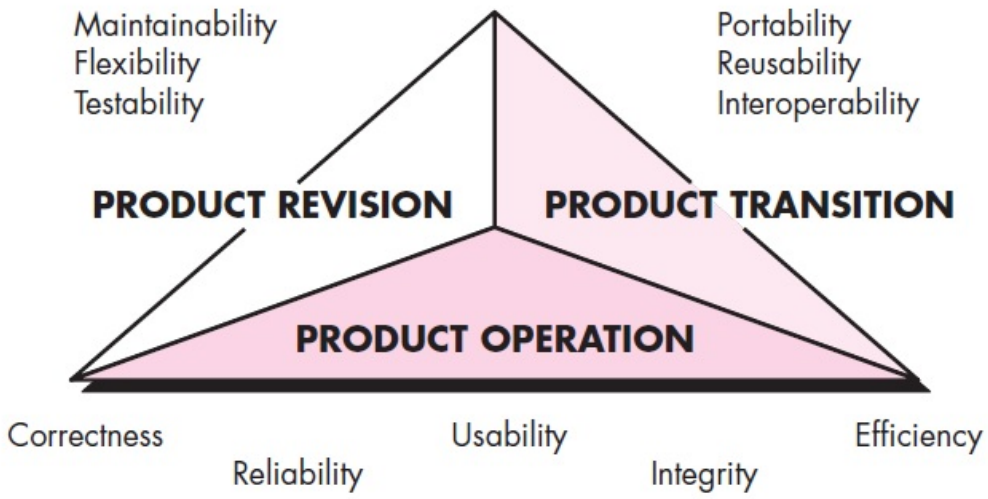
McCall's Quality Factors

McCall and his colleagues provide the following Quality Factors:

- ***Correctness***. The extent to which a program satisfies its specification and fulfills thecustomer's mission objectives.
- ***Reliability.***The extent to which a program can be expected to perform its intended function with required precision
- ***Efficiency***. The amount of computing resources and code required by a program to perform its function.
- ***Integrity.***Extent to which access to software or data by unauthorized persons can be controlled.
- ***Usability***. Effort required to learn, operate, prepare input for, and interpret output of aProgram
- ***Maintainability***. Effort required to locate and fix an error in a program. .
- ***Flexibility***. Effort required to modify an operational program.
- ***Testability***. Effort required to test a program to ensure that it performs its intended function.
- ***Portability***. Effort required to transfer the program from one hardware and/or software system environment to another.

- ***Reusability***. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

- ***Interoperability.***Effort required to couple one system to another.



ISO 9126 QUALITY FACTORS

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software.

The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

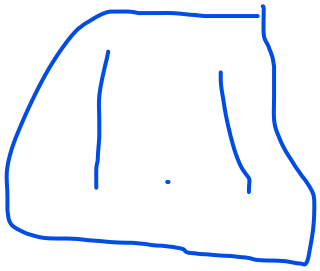
Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability

Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.



ACHIEVING SOFTWARE QUALITY

Software quality doesn't just appear. It is the result of good project management and solid software engineering practice.

Management and practice are applied withinthe context of four broad activities

Software Engineering Methods

- To build high-quality software**, need to understand the problem to be solved.**

- Software engineering methodscan lead to a reasonably complete understanding of the problem and a comprehensive design that establishes a solid foundation for the construction activity.

- By applying methods probability of getting high-quality software will increase

Project Management Techniques

- There is vital impact on quality of software because of poor management decisions

- Impact of poor Project management techniques involves
  - ✓ **Impact on delivery dates,**
  - ✓ **Irregular schedule**
  - ✓ **risk planning will be affected**

- The project plan should include explicit techniques for quality and change management.

Quality Control

- Quality control encompasses a set of software engineering actions that help toensure that each work product meets its quality goals.

- **Models are reviewed to ensure that they are complete and consistent.**

- **Code may be inspected in order to uncover and correct errors before testing commences**.

- A **series of testing steps is applied touncover errors in processing logic**, data manipulation, and interface communication.

- A combination of measurement and feedback process when any of these work products fail to meet quality goals.

Quality Assurance

- Quality assurance establishes the infrastructure that supports useful various software engineering methods, rational project management, and quality control actions—

- **All related methods are pivotal if it intend to build high-quality software**.

- **The quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions**.

- **The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality**,

- if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and applythe necessary resources to resolve quality issues

Software Quality Assurance

BACKGROUND ISSUES

- Definition- Software quality assurance is a "planned and systematic pattern of actions that are required to ensure high quality in software"

- **Q**uality **c**ontrol **and assurance are essential activities** for any business that produces products

- **Prior to the twentieth century, quality control was the sole responsibility of the craftsperson who built a product. As time passed and mass production, quality control became an activity performed by people other than the ones who built the product**.

- The **first formal quality assurance and control function** was introduced **at BellLabs in 1916** and spread rapidly throughout world.

- During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.

- **Today, every company has mechanisms to ensure quality in its products**.

- The history of quality assurance in software development parallels the history of quality in hardware manufacturing.

- **During the early days of computing quality was the sole responsibility of the programmer**.

- Standards for quality assurance for software were introduced in military contract software development during 1970s and have spread rapidly into software development in world

ELEMENTS OF SOFTWARE QUALITY ASSURANCE

Standards.

- The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents.

- **Standards may be adopted voluntarily** by a software engineering organization or imposed by the customer or other stakeholders.

- **The job ofSQA is to ensure that standards that have been adopted are followed andthat all work products conform to them**.

Reviews and audits.

- Technical reviews are a quality control activity performed by software engineers. Their **intent is to uncover errors**.

- Audits are a type of review performed bySQA personnel with the intent of ensuring that quality guidelines are beingfollowed for software engineering work.

Testing.

- Software **testing, isA quality control functionthat has primary goal to find errors**.

- **The job of SQA is to ensurethat testing is properly planned and efficiently conducted so that it has thehighest likelihood of achieving its primary goal**.

Error/defect collection and analysis.

- The only way to improve is tomeasure how you're doing.

- **SQA collects and analyzes error and defect datato better understand how errors are introduced and what software engineeringactivities are best suited to eliminating them**.

Change management.

- Change is one of the **most disruptive** aspects ofany software project.

- If it is **not properly managed**, change can **lead to confusion**, and confusion almost always leads to poor quality.

Education.

- **Every software organization**wants to improve its software engineering practices.

- A key contributor to improvement is education of software engineers, their managers, and other stakeholders.

- The **SQA organizationtakes the lead in software process improvement** and is akey proponent and sponsor of educational programs.

Vendor management.

- Three categories of software are acquired fromexternal software vendors—

  - ✓ ***shrink-wrapped packages***,
  - ✓ a ***tailored shell***that provides a basic skeletal structure that is customtailored to the needs of a purchaser, and
  - ✓ ***contracted software***that is customdesigned and constructed from specifications provided by the customerorganization.

- The job of the SQA organization is to ensure that high-qualitysoftware results by suggesting specific quality practices that the vendor should follow when possible, and incorporating quality mandates as part ofany contract with an external vendor.

Security management.

- With the increase in **cyber crime** and new governmentregulations regarding privacy,

- every software organization should institutepolicies that protect data at all levels, establish firewall protection forWebApps, and ensure that software has not been tampered with internally.

- **SQA ensures that appropriate process** and technology are used **to achievesoftware security.**

Safety.

- SQA may be responsible for assessing the impactof software failure and for initiating those steps required **to reduce risk**.

Risk management.

- Although the analysis and mitigation of risk is the concern of software engineers,

- The SQA organization ensures thatrisk management activities are properly conducted and that risk-relatedcontingency plans have been established.

SQA TASKS, GOALS, AND METRICS

SQA Tasks

SQA Tasksprepares an SQA plan for a project.

- The plan is developed as part of project planning and is reviewed by all stakeholders.

- Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan.

- The plan identifies
  - ✓ evaluations to be performed,
  - ✓ audits and reviews to be conducted,
  - ✓ standards that are applicable to the project,
  - ✓ procedures for error reporting and tracking,
  - ✓ work products that are produced by the SQA group, and
  - ✓ feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management.
Noncompliance items are tracked until they are resolved.
In addition to these actions, the SQA group coordinates the control and management

SQA goals and metrics

Requirements quality.

- The **correctness, completeness, and consistency**have a influence on the quality of work products that follow.

- SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality.

- Every element of the design model should be assessed by the software team to ensure that it exhibits high quality

- SQA looks for attributes of the design thatare indicators of quality.

Code quality.

- Source code and related work products must conform local coding standards and exhibit characteristics that will facilitate maintainability.

- SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness.

- A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result.

- SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

FORMAL APPROACHES TO SQA

- software quality is everyone's job and that it can be achieved through
  - ✓ competent software engineering practice
  - ✓ application of technical reviews,
  - ✓ a multi-tiered testing strategy,
  - ✓ better control of software work products and the changes made to them, and
  - ✓ the application of accepted software engineering standards.

- Software Quality can be definedin terms of quality attributes and measured using avariety of indices and metrics.

- Over the past three decades, a small, segment of the software engineeringcommunity demanding more formal approach to software quality assurance

- A computer program is a mathematical object. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements is available.

- If the requirements model and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

- Attempts to prove programs correct are not new. Dijkstra and Linger, Mills, and Witt [Lin79], advocated proofs of program correctness and tied these to the use of structured programming concepts

STATISTICAL SOFTWARE QUALITY ASSURANCE

- Statistical quality assurance reflects a **growing trend** throughout industry to become more quantitative about quality.

- Software, statistical quality assurance implies the following steps:
  - ✓ Information about software errors and defects is collected and categorized.
  - ✓ An attempt is made to trace each error and defect
  - ✓ Using the Pareto principle (80 percent of the defects can be traced to 20 percent), isolate the 20 percent (the *vital few*).
  - ✓ Once the vital few causes have been identified, move to correct the problemsthat have caused the errors and defects.

A Generic Example

- To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on errors and defects for a period of one year.

- Some of the errors are uncovered as software is being developed. Others (defects) are encountered after the software has been released to Its end users.

Although many different problems are uncovered, tracked the following causes:

- ✓ Incomplete or erroneous specifications (IES)
- ✓ Misinterpretation of customer communication (MCC)
- ✓ Intentional deviation from specifications (IDS)
- ✓ Violation of programming standards (VPS)
- ✓ Error in data representation (EDR)
- ✓ Inconsistent component interface (ICI)
- ✓ Error in design logic (EDL)
- ✓ Incomplete or erroneous testing (IET)
- ✓ Inaccurate or incomplete documentation (IID)
- ✓ Error in programming language translation of design (PLT)
- ✓ Ambiguous or inconsistent human/computer interface (HCI)
- ✓ Miscellaneous (MIS)

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

Six Sigma for Software Engineering

- *Six Sigma* is the most widely used strategy for statistical quality assurance in industry

- Six Sigma strategy
  - ✓ is a rigorous and disciplined methodology
  - ✓ uses data and statistical analysis tomeasure and improve a company's operational performance
  - ✓ company's operational performance measures by identifying and eliminatingdefects' in manufacturing and service-related processes

- The termSix Sigma is derived from six standard deviations—defects per millionoccurrences

- Used for extremely high quality

The Six Sigma methodology defines three core steps:

- ***Define***customer requirements and project goals via welldefined methods of customer communication.

- ***Measure***the existing process and its output to determine current qualityperformance (collect defect metrics).

- ***Analyze***defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- ***Improve***the process by eliminating the root causes of defects.

- ***Control***the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the **DMAIC**method. (define, measure, analyze, improve, and control)

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

- ***Design***the process to avoid the root causes of defects and to meetcustomer requirements.
- ***Verify***that the process model will, in fact, avoid defects and meet customerrequirements.

This variation is sometimes called the **DMADV** (define, measure, analyze, design, and verify) method.

SOFTWARE RELIABILITY

- *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time"

- **One failure**can be corrected within seconds, while another requires weeks or even months to correct.

- the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

Measures of Reliability and Availability

- Early work in software reliability deals with hardware reliability theory to the prediction of software reliability.

- Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects.

- In hardware, failures due to physical wear (the effects of temperature, corrosion, shock) are more likely than a design-related failure. the opposite is true for software

- All software failures because of design or implementation problems; but wear does not enter into the picture.

- If we consider a computer-based system, a simple measure of reliability is
    - ✓ **MTBF: *meantime- between-failure***
    - ✓ **MTTF:*mean-time-to-failure***
    - ✓ **MTTR:*mean-time-torepair*,**

- MTBF is a far more useful measure than other quality-related software metrics

- an end user is concerned with failures, not with the total defect count.

- MTBF can be problematic for two reasons:
  (1) it projects a time span betweenfailures, but does not provide us with a projected failure rate, and
  (2) MTBF canbe misinterpreted to mean average life span but this is *not* what it implies.

- An alternative measure of reliability is ***failures-in-time*** **(FIT)—**a statistical measure of how many failures a component will have over one billion hours of operation.
  EX: 1 FIT is equivalent to one failure in every billion hours of operation.

- ***Software availability***a reliability measure the probability that a program is operating according torequirements
  Ex:Availability _ _ 100%

Software Safety

- *Software safety* is a **SQA activity, focuses on the identification and assessment of hazard**s that may affect software negatively and cause an entire system to fail.

- If hazards can be identified early in the software process, that will either eliminate or control potential hazards.

- A **modeling**and **analysis process** is conducted as part of software safety.

- Initially, hazards are identified and categorized by criticality and risk.

- Once these system-level hazards are identified, there is starting to assign severity and probability of occurrence.

- To be effective, software must be analyzed in the context of the entire

- If and only if a set of external environmental conditions is met, the improper position of the mechanical device will cause a disastrous failure.

- Analysis techniques such as **fault tree analysis, real-time logic, and Petri net models can be used to predict** causes of hazards.

- Once hazards are identified and analyzed, safety-related requirements can bespecified for the software.

- the specification can contain a list of undesirableevents and the desired system responses to these events.

- Software safety examines the ways in which failures result in conditions that can lead to a mishap.

- That is, failures are not considered in a vacuum, but are evaluated in thecontext of an entire computer-based system and its environment.

THE ISO 9000 QUALITY STANDARDS

- A quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

- Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting theirspecifi -cations.

- These systems cover a wide variety of activities encompassing a product'sentire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process.

- ISO 9000 describes quality assurance elements in generic terms that can beapplied to any business regardless of the products or services offered.

- To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by thirdpartyauditors for compliance to the standard and for effective operation.

- Upon successful registration, a company is issued a certificate from a registration body represented by the auditors.

- Semiannual surveillance audits ensure continued compliance to the standard.

- The requirements delineated by ISO 9001:2008 address topics such as
  - ✓ management responsibility,
  - ✓ quality system,
  - ✓ contract review,
  - ✓ design control,
  - ✓ document and data control,
  - ✓ product identification and traceability,
  - ✓ process control,
  - ✓ inspection and testing,

- ✓ corrective and preventive action,
- ✓ control of quality records,
- ✓ internal quality audits,
- ✓ training,
- ✓ servicing, and
- ✓ statistical techniques.

- In order for a **software organization to become registered to ISO 9001:2008**, **it must establish policies and procedures to address each of the requirements just noted**and thenbe able to demonstrate that these policies and procedures are being followed.

THE SQA PLAN

- The SQA Plan provides a road map for instituting software quality assurance.

- It developed by the SQA group or by the software team if an SQA group does notexist the plan serves as a template for SQA activities that are instituted for eachsoftware project

- A standard for SQA plans has been published by the IEEE [IEE93].

- The standardrecommends a structure that identifies:
  - ✓ the purpose and scope of theplan,
  - ✓ a description of all software engineering work products (e.g., models,source code) that fall within the SQA,
  - ✓ all applicablestandards and practices that are applied during the software process,
  - ✓ SQAactions and tasks (including reviews and audits) and their placement throughoutthe software process,
  - ✓ the tools and methods that support SQA actionsand tasks,
  - ✓ software configuration management procedures,
  - ✓ methods for assembling, safeguarding, and maintaining all SQA-related records,and
  - ✓ organizational roles and responsibilities relative to product quality

REENGINEERING

Re-engineering

"Re-engineering is the examination and alteration of a system to reconstitute it in a new form."

- This process encompasses a combination of sub-processes like reverse engineering, forward engineering, reconstructing etc

Objectives of Re-engineering:

- To describe a cost-effective option for system evolution.
- To describe the activities involved in the software maintenance process.
- To distinguish between software and data re-engineering and to explain the problems of data re-engineering

SOFTWARE REENGINEERING

- Reengineering is a rebuilding activity,

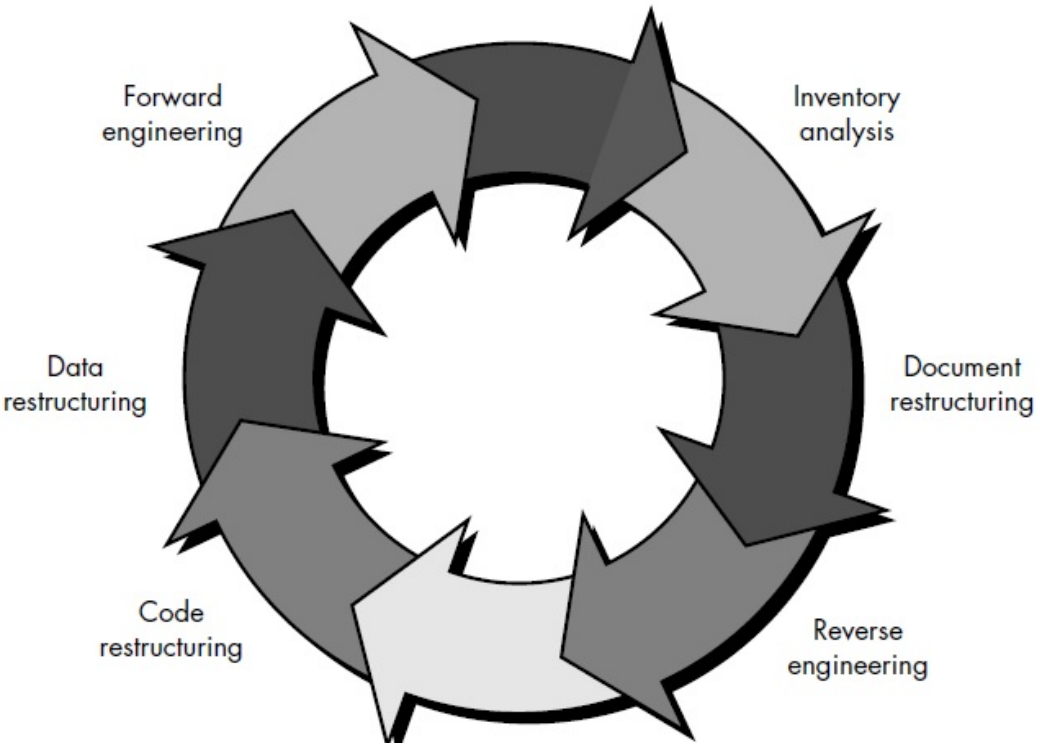


FIG: RE-ENGINEERING

Steps involved in Re-engineering:

- **Inventory analysis. .**
  - ✓ Every software organization should have an inventory of allapplications.
  - ✓ The inventory can be nothing more than a spreadsheet model containinginformation that provides a detailed description of every active application.
  - ✓ the inventory should be revisited on a regular cycle.
  - ✓ The status of applications change as a function of time,

- **Document restructuring**.
  Weak documentation is the trademark of many legacysystems.
  Solution over this
  1. *Creating documentation is far too time consuming.*
     If a program is static- Documentations is limited
     If a program is motstatic life, -Documentations undergo significant changes

  2. *Documentation must be updated, but we have limited resources.*

  3. *The system is business critical and must be fully redocumented.*

- **Reverse engineering.**
  - The term *reverse engineering* has its origins in the hardwareworld.
  - A company disassembles a competitive hardware product in an effort to understandits competitor's design and manufacturing "secrets."
  - These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained.
  - In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimen of the product.

- Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code.
- Reverse engineeringis a process of design recovery.

- Reverse engineering tools extract data, architectural,and procedural design information from an existing program

Code restructuring.

- Some legacy systems have a relatively solid program architecture,
- But individual modules were coded in a way that makes them difficult to understand, test, and maintain.
- In such cases, the code within the suspect modules can be restructured.
- To accomplish this activity, the source code is analyzed using a restructuring tool.
- Violations of structured programming constructs are noted and code is then restructured
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced

Data restructuring

- A program with weak data architecture will be difficult to adapt and enhance.
- Data restructuring is a full-scale reengineering activity.
- In most cases, data restructuring begins with a reverse engineering activity.
- Data objects and attributes are identified, and existing data structures are reviewed for quality.

Forward engineering

- Forward engineering not only recovers design information from existing software
- Also forward engineering uses design information to alter or reconstitute the existing system in an effort toimprove its overall quality.

Reverse Engineering

Reverse engineering can extract design information from source code,

but the abstraction level, the completeness of the documentation,
the degree to which tools and a human analyst work together, and the
directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations
(a low-level abstraction), program and data structure information
(a somewhat higher level of abstraction), object models, data and/or control flow models (a relatively high level of abstraction), and entity relationship models (a high
level of abstraction). As the abstraction level increases, you are provided with information
that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is “integrated” with automated tools to create an effective reverse engineering
process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

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