Contents

1. Chapter
Overview of software Engineering & the Software Development Process
1.1. Definition of Software and Characteristics of Software

Computer software is the product that software professionals build and then support over the long term.

Software is a set of instructions to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software.

"Ideas and technological discoveries are the driving engines of economic growth."

-The Wall Street Journal

"In modern society, the role of engineering is to provide systems and products that enhance the material aspects of human life, thus making life easier, safer, move secure, and more enjoyable."

- Richard Fairley & Mary Will Shire

Software Engineering is defined as a discipline that addresses the following aspects of the software and its development. The aspects are:

1. Economic : Cost, Benefits, and Returns on Investment (ROI).
2. Design : Ease of development and ensuring delivery of customer requirements.
3. Maintenance : Ease of effecting changes and modifications.
4. Implementation : Ease of installation, Demonstration, and implementation of software by the customer and users.

It is an engineering discipline which is systemic, scientific, and methodical and uses standards, models and algorithms in design & development.

The IEEE (Institution of Electrical and Electronics Engineers) defines Software Engineering as the application of a systematic, disciplined, quantifiable approach to the development, operations and maintenance of software.

Today, software takes on a dual role. It is both a product and a vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, by a network of computers that are accessible by local hardware. Software is an information transformer - producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

As a vehicle, for delivering the product, software acts as the basis for the control of computer (Operating System), the communication of information (networks), and the creation and control of other programs (Software tools and environments).

"Computers make it easy to do a lot of things, but most of the things they make it easier to do don't need to be done"

- Andy Rooney

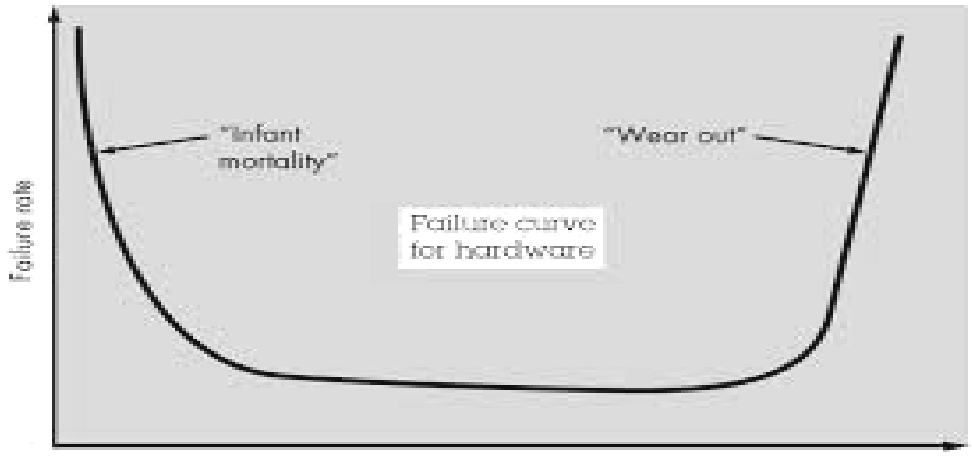
Characteristics of Software:

Software is written to handle an Input – Process – Output system to achieve predetermined goals. Software is logical rather than a physical system element. Therefore software has characteristic that are different than that of hardware.

1. Software is developed or engineered; it is not manufactured in the classical sense.
2. Software doesn't "wear out" like hardware and it is not degradable over a period.

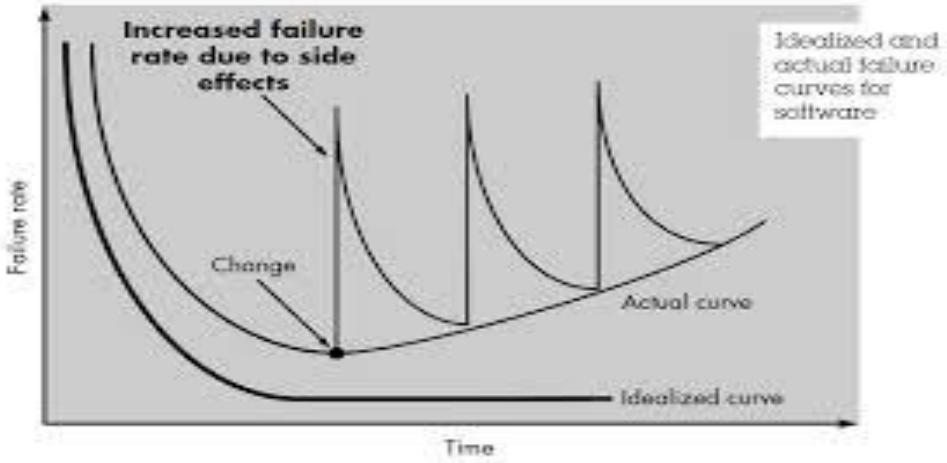
The following figure depicts failure rate as a function of time for hardware. The relationship often called as "bathtub curve", indicates that, hardware exhibits relatively high failure rates early in its life.

Failure curve for Hardware:



Time

Failure curve for Software:



Software is not susceptible to the environmental maladies that cause hardware to wear out. Hence, the failure rate curve for software should take the form of "idealized curve" as shown in the above figure. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. Hence the software doesn't wear out, but it does deteriorate.

3. Although the industry is moving toward component – based construction, most software continues to be custom built.
4. A software component should be designed and implemented so that it can be reused in many different programs.

The classical and conventional definition of software is that it is a set of instructions, which, when executed through a computing device, produces the desired result by the execution of functions and processes.

Software is a set of instructions to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the

software. It also includes a set of documents, such as the software manual, meant for users to understand the software system. Today's software comprises the source code, Executable, Design Documents, Operations and System Manuals and installation and Implementation Manuals.

Software is described by its capabilities. The capabilities relate to the functions it executes, the features it provides and the facilities it offers.

1.2. Types/ Categories of Software:

Today, seven broad categories of computer software present continuing challenges for software engineers.

1. **System Software:**

   System Software is a collection of programs written to serve other programs. Some system software (e.g.- compliers, editors, and file management utilities) processes complex, but determinate information structures. Other system applications (e.g.- operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

2. **Application Software:**

   Application Software consists of standalone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilities business operations or management / technical decision-making.

3. **Engineering / Scientific Software:**

   Formerly characterized by "number crunching" algorithms, engineering and scientific software applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. Computer-aided design, system simulation, and other interactive applications have begun to take on real–time and even system software characteristics.

4. **Embedded Software:**

   Embedded Software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself. Embedded software can perform limited and esoteric functions (e.g. keypad control for a microwave oven) or provide significant function and control capability (e.g. digital functions in an automobile such as fuel control, dashboard displays, braking systems, etc.)

5. **Product–line Software:**

   Designed to provide a specific capability for use by many different customers, product–line software can focus on a limited & esoteric market place (e.g. – inventory control products) or address mass consumer markets (e.g. – word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications.)

6. **Web – applications:**

   "WebApps", span a wide array of applications. WebApps are evolving into sophisticated computing environments that not only provide standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

7. Artificial Intelligence Software:

AI Software makes use of non–numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Due to changing nature of software, i.e. rapid growth of technology, the challenge for software engineers will be –

a) To develop systems and application software that will allow small devices, personal computers, and enterprise system to communicate across vast networks to meet rapid growth of wireless networking.
b) To architect simple (e.g.- personal financial planning) and sophisticated applications that provide benefit to targeted end-user markets worldwide to meet rapid growth of net sourcing (World Wide Web)
c) To build source code that is self descriptive, but, more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.
d) To build applications that will facilitate mass communication and mass product distribution using concepts that is only now forming.
e) "The computer itself will make a historic transition from something that is used for analytic tasks…. to something that can elicit emotion".

- David Vaskevitch

1.3. Software Engineering- Definition, Need

According to Fritz Bauer, software engineering is establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

"More than a discipline, or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem".

- Scott Whit mire

A more comprehensive definition of IEEE-

Software Engineering:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering to software.

1.4. Relationship between Systems Engineering and Software Engineering

Software Engineering

Software engineering deals with designing and developing software of the highest quality. A software engineer does analyzing, designing, developing and testing software. Software engineers carry out software engineering projects, which usually have a standard software life cycle. For example, the Water Fall Software Life cycle will include an analysis phase, design phase, development phase, testing and verification phase and finally the implementation phase. Analysis phase looks at the problem to be solved or the opportunities to be seized by developing the software. Sometimes, a separate business analyst carries out this phase. However, in small companies, software engineers may do this task. Design phase involves producing the design documents such as UML diagrams and ER diagrams depicting the overall structure of the software to be developed and its components. Development phase

involves programming or coding using a certain programming environment. Testing phase deals with verifying that software is bug free and also satisfies all the customer requirements. Finally, the completed software is implemented at the customer site (sometimes by a separate implementation engineer). In recent years, there has been a rapid growth of other software development methodologies in order to further improve the efficiency of the software engineering process. For example, Agile methods focus on incremental development with very short development cycles. Software Engineering profession is a highly rated job because of its very high salary range.

System Engineering

System Engineering is the sub discipline of engineering which deals with the overall management of engineering projects during their life cycle (focusing more on physical aspects). It deals with logistics, team coordination, automatic machinery control, work processes and similar tools. Most of the times, System Engineering overlaps with the concepts of industrial engineering, control engineering, organizational and project management and even software engineering. System Engineering is identified as an interdisciplinary engineering field due to this reason. System Engineer may carry out system designing, developing requirements, verifying requirements, system testing and other engineering studies.

1.5. Software engineering layers: A Layered Technology Approach

Software engineering is a layered technology. The layers of software engineering as shown in the above diagram are:-

1. **A Quality Focus:**

   Any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, six sigma and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

2. **Process Layer:**

   The foundation for software engineering is the process layer. 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 that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, works products (models, documents, data, reports, forms etc.) are produced, milestones are established, quantity is ensured and change is properly managed.

3. **Methods:**

   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.

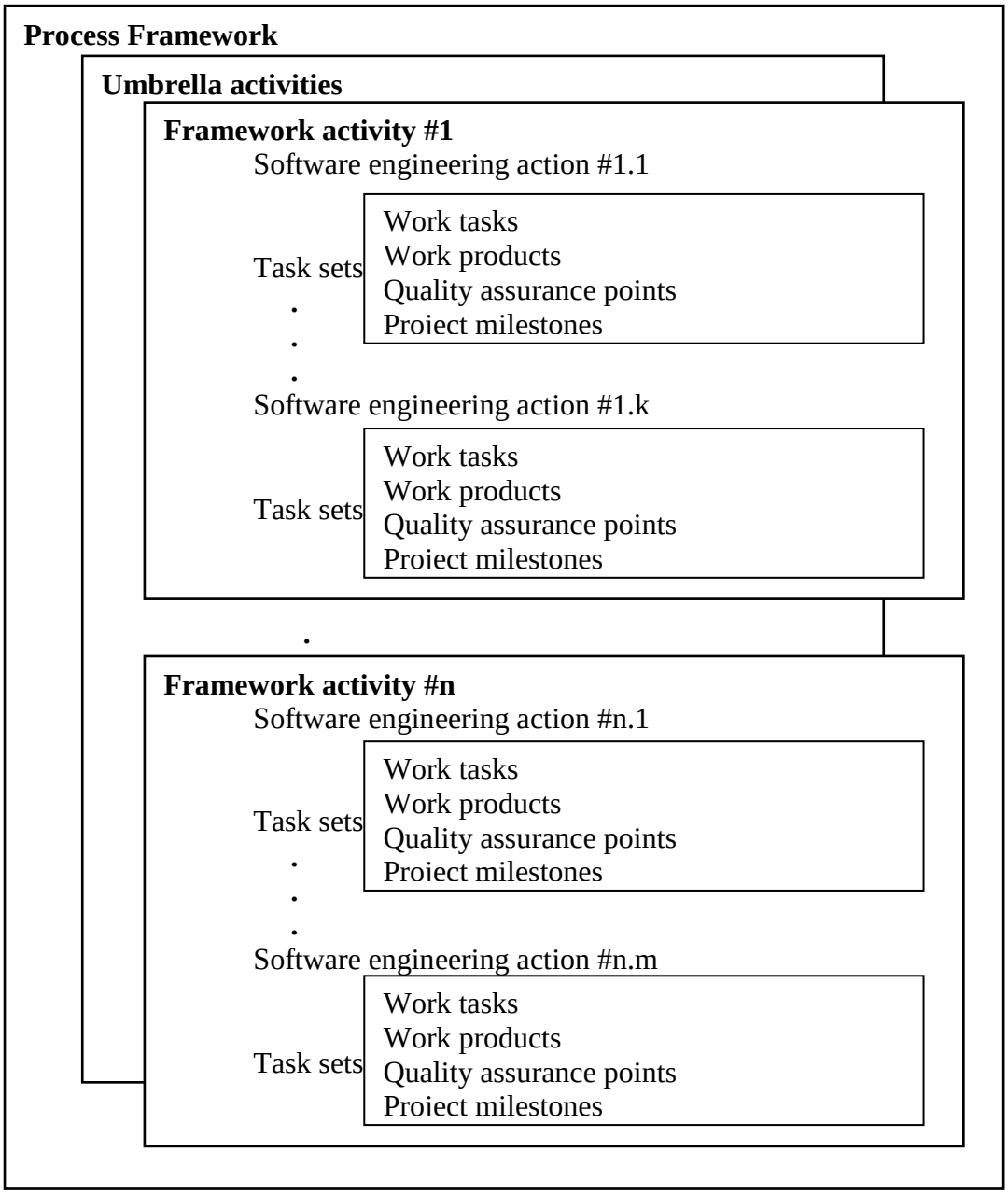
4. **Tools:**

   Software Engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer–aided software engineering is established.

1.6. Software Development Generic Process Framework – Software Process, Software Product, Software Work – Product, Basic Framework Activities, Umbrella Activities

A process framework establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.

Software Process



From the above figure, each framework activity is populated by a set of software engineering actions- a collection of related tasks that produces a major software engineering work product (e.g. design is a SE action). Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

"A process defines who is doing what, when and how to reach a certain goal".

- Ivar Jacobson, Grady Booch and James Rumbaugh

The following generic process framework is applicable to the vast majority of software projects.

1. **Communication :**

   This framework activity involves heavy communication & collaboration with the customer (and the stakeholders) and encompasses requirements gathering and other related activities.

2. **Planning :**

   This activity establishes a plan for the software engineering work that follows. It describes 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.

3. **Modeling :**

   This activity encompasses the creation of models that allow the developer & the customer to better understand software requirements & the design that will achieve those requirements.

4. **Construction :**

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

5. **Deployment :**

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

   "Einstein argued that there must be a simplified explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary."

   - Fred Brooks

Umbrella Activities:

Generic views of SE is complemented by a set of unbrella activities. They are

Software Project tracking and control:

The framework described in the generic view of SE is complemented by a number of umbrella activities, one of which is software project tracking and control. It allows the software team to access progress against the project plan and takes necessary action to maintain schedule. Umbrella activities occur throughout the software process and focus primarily on project management, tracking and control.

Risk Management:

Assess risks that are likely to affect performance and quality of project.

Software quality assurance:

Define and conduct activities to ensure software quality.

Formal Technical Review:

Assess Software Engg. Work products to uncover and remove errors before they are shifted to next level of activity.

Measurement:

Defines and collects process, project and product measures to assist the team in delivering the software that meets customer needs can be used in conjunction with all framework and umbrella activities.

Software configuration Management (SCM):

Manages and effects the changes throughout the software process.

Reusability management:

Defines criteria for work product reuse (including software components) and establishes the mechanism to achieve reusable components.

Work product preparation and production:

Includes activities for creating work product such as models, documents, large,

1.7. Personal And Team Process Models (PSP and TSP) – Concept, Significance with respect to Ongoing Process Improvement, Goals, List of framework activities included

The best software process is one that is close to the people who will be doing the work.

Watts Humphrey argues that it is possible to create a "Personal Software Process" and/or a "Team Software Process". Both require hard work, training and co-ordination, but both are achievable.

"A person who is successful has simply formed the habit of doing things that unsuccessful people will not do".

- Dexter Yager

Personal Software Process(PSP):

Watts Humphrey suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quantity of the work product. In addition, the PSP makes the practitioner responsible for project planning (e.g. estimating and scheduling) and empowers the practitioner to control the quantity of all software work products that are developed.

The PSP process model defines five framework activities: Planning, high-level design review, development and Postmortem.

1. **Planning:**

   This activity isolates requirements and based on these, develops both size and resource estimates. Development tasks are identified & a project schedule is created.

2. **High – Level Design:**

   External Specifications for each component to be constructed are developed and a component design is created.

3. **High – Level Design Review:**

   Formal verification methods are applied to uncover errors in the design.

4. **Development :**

   The component level design is refined and reviewed. Code is generated, reviewed, complied, and tested. Metrics are maintained for all important tasks and work results.

5. **Postmortem :**

   Using the measures & metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

Team Software Process(TSP):

The goal of TSP is to build a "Self-directed" project team that organizes itself to produce high-quantity software. Humphrey defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of 3 to about 20 engineers.
- Show managers how to coach & motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives. To form a self-directed team, we must collaborate well internally and communicate well externally.

"Finding good players is easy. Getting them to play a team is another story".

- Casey Stengel

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality.

Process Technology:

The generic process models must be adapted for use by a software project team. To accomplish this, process technology tools have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress and manage technical quantity.

Process technology tools allow a software organization to build an automated model of the common process framework, task sets and umbrella activities. The model, normally represented as a network can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once, an acceptable process has been created, other process technology tools can be used to allocate, monitor and even control all software engineering tasks defined as part of the process model, to develop a checklist of work tasks to be performed, work products to be produced and quantity assurance activities to be conducted, to coordinate the use of other computer-aided software engineering tools that are appropriate for a particular work task.

In some cases, the process technology tools incorporate standard project management tasks such as estimating, scheduling, tracking and control.

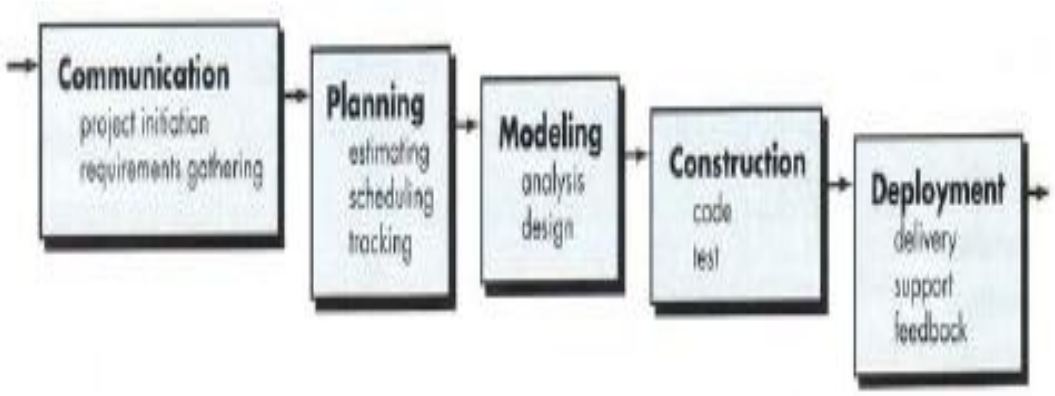
1.8. Prescriptive Process Models:

Irrespective of which level of CMM the organization has, the software engineer has five choices for selection of software process models.

They are –

1. Waterfall Model
2. Incremental Model
3. RAD Model
4. Prototype Model
5. Spiral Model

1. The Waterfall Model:



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

The waterfall model is a traditional method, 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 on-going support of the completed software.

This is one of the initial models. As the figure implies stages are cascaded and shall be developed one after the other. In other words one stage should be completed before the other begins. Hence, when all the requirements are elicited by the customer, analyzed for completeness and consistency, documented as per requirements, the development and design activities commence.

This model presents a high level view and suggests to the developer the sequence of events they should expect to encounter. This model is used to prescribe software development activities in variety of contexts. It is the basis for software deliverables. Associated with each activity are milestones and outcomes, for managers to monitor.

One of the main needs of this model is the user's explicit prescription of complete requirements at the start of development. For developers it is useful to layout what they need to do at the initial stages. Its simplicity makes it easy to explain to customers who may not be aware of software development process. It makes explicit with intermediate products to begin at every stage of development.

One of the biggest limitation is it does not reflect the way code is really developed.

Problem is well understood but software is developed with great deal of iteration.

Often this is a solution to a problem which was not solved earlier and hence software developers shall have extensive experience to develop such application; as neither the user nor the developers are aware of the key factors affecting the desired outcome and the time needed. Hence at times the software development process may remain uncontrolled.

Today software work is fast paced and subject to a never-ending stream of changes in features, functions and information content. Waterfall model is inappropriate for such work. This model is useful in situation were the requirements are fixed and work proceeds to completion in a linear manner.

Among the problems that are sometimes encountered when the waterfall model is applied are

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so directly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The Waterfall Model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program will not be available until late in the project time-span. A major blunder, if undetected until the working program is received, can be disastrous.

The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

2. The Incremental Model:

The incremental model combines elements of the waterfall model applied in an iterative fashion. The incremental model delivers a series of releases, called increments, that provides progressively more functionality for the customer at each increment is delivered. In each increment, additional functions and features are added after confirming the utility of earlier increments.

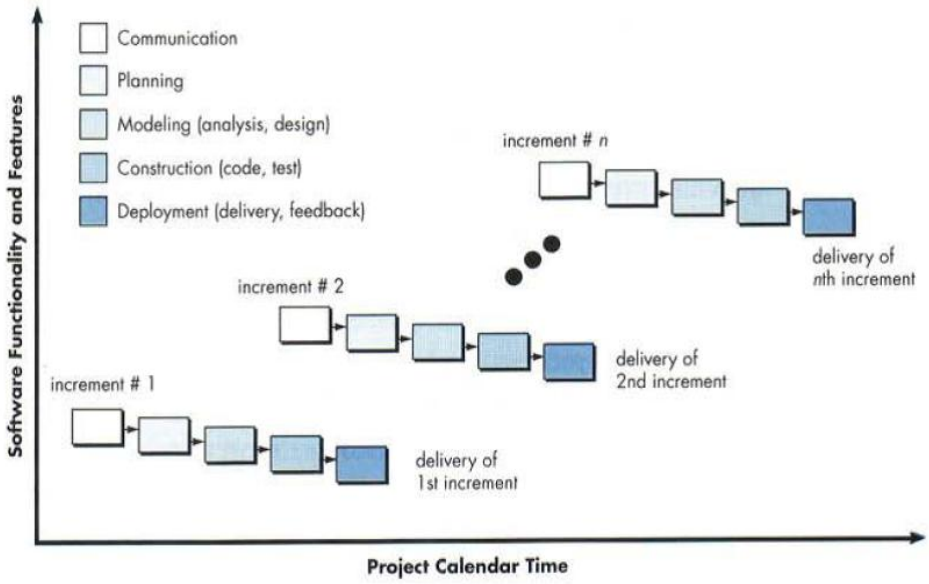
In the early years of development users were willing to wait for software projects to be ready. Today's business does not tolerate long delays. Software helps to distinguish products in the market place and customers are always looking for new quality and functions. One of the ways to reduce time is the phased development. The system is developed such that it can be delivered in parts enabling the users to have few functions while the rest are being developed. Thus development and usage will happen in parallel.

In incremental development the system is partitioned into subsystems or increments. The releases are defined in the beginning with initial function and them adding functionalities with subsequent releases. Incremented development slowly builds up to full functionality with subsequent releases.

This model combines the elements of waterfall model in an iterative fashion. The model applies linear sequences in a staggered manner as the calendar time progresses. In this model first increment is the core product or primary function. The core product implemented undergoes detailed evaluation by the user which becomes advantages for future increments. The feedback also addresses future modifications which are included in the next increments for additional features and functionality. He process is repeated till delivery of each increment till the final product is delivered.

This is useful when the software team is smaller in size. Additional increments can be planned and managed to address technical risks. This has the advantage of prompt system delivery to users without hassle.

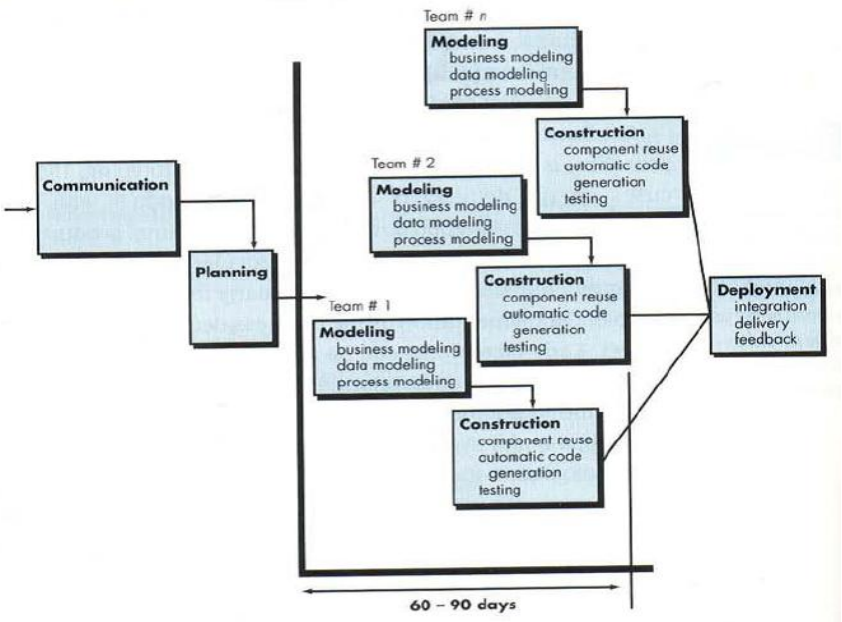
In case of availability of new hardware are delayed and early increments which could be executed on existing systems for partial functionality to prevent inordinate delays.



From this diagram, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "Increments" of the software.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

3. The RAD Model:



Rapid application Development (RAD) is a modern software process model that emphasizes a short development cycle. The RAD Model is a "high-speed" adaptation of the waterfall model, in which rapid development is achieved by using a component based construction approach. If requirements are well understood and project scope is considered, the RAD process enables a development team to create a "Fully Functional System" within a very short period of time (e.g. 60 to 90 days).

One of the distinct features of RAD model is the possibility of cross life cycle activities which will be assigned to teams, teams #1 to team #n leading to each module getting developed almost simultaneously.

This approach is very useful if the business application requirements are modularized as function to be completed by individual teams and finally to integrate into a complete system. As such compared to waterfall model the team will be of larger size to function with proper coordination.

RAD model distributes the analysis and construction phases into a series of short iterative development cycles. The activities of each phase per team are Business modeling, Data modeling and process modeling.

This model is useful for projects with possibility of modularization. RAD may fail if modularization is difficult. This model should be used if domain experts are available with relevant business knowledge.

Communication works to understand the business problem and the information characteristics that the software must accommodate. Planning is essential because multiple software teams' work is parallel on different system functions. Modeling encompasses three major phases – business modeling, data modeling and process modeling- and establishes design representations that serve as the basis for RAD's construction activity. Construction emphasizes the use of pre-existing software components and the application of automatic code generation. Finally, deployment establishes a basis for subsequent iterations, if required.

Advantages:

1. Changing requirements can be accommodated and progress can be measured.
2. Powerful RAD tools can reduce development time.
3. Productivity with small team in short development time and quick reviews, risk control increases reusability of components, better quality.
4. Risk of new approach only modularized systems are recommended through RAD.
5. Suitable for scalable component based systems.

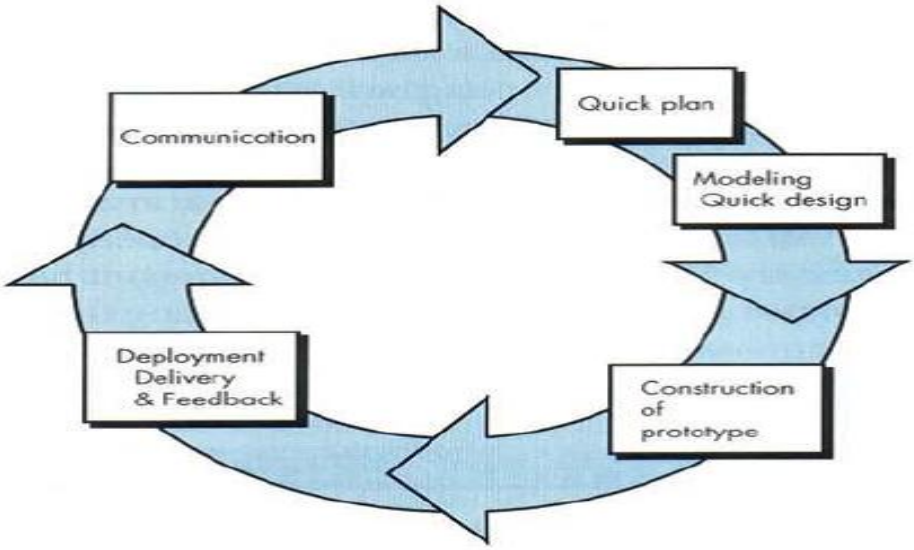
Limitations:

1. RAD model success depends on strong technical team expertise and skills.
2. Highly skilled developers needed with modeling skills.
3. User involvement throughout life cycle. If developers & customers are not committed to the rapid fire activities necessary to complete the System in a much-abbreviated time frame, RAD projects will fail.
4. May not be appropriate for very large scale systems where the technical risks are high.

The difference between RAD and Incremental model (INM) is that, in RAD the requirement of the software system is well defined & agreed by all, namely users, customers and stakeholders; whereas in INM, requirements need to be evolved incrementally to ensure its correctness and to assure quality. Each increment goes through the core processes from analysis to testing before it is delivered to the customer.

4. The Prototype Model:

The prototyping paradigm begins with communication as shown in the diagram below.



The software development process can help to control by including activities and sub processes to enhance understanding. Prototyping is a sub process or a partially developed product that enable customers and developers to examine aspects of a proposed system and decide if it is suitable or appropriate for the finished product.

Developers may build a system to implement a small portion of some of the key requirements to ensure that the requirements are consistent, feasible and practical. In case of changes, revisions are made at the requirements stage by prototyping parts of the design.

Design prototyping helps the developers assess alternative strategies and decide which best suits for the project. There may be radically different designs to get best performances. Often user interface is built and tested as a prototype for users to understand the new system and developers to get the idea of user's reaction/response to the system.

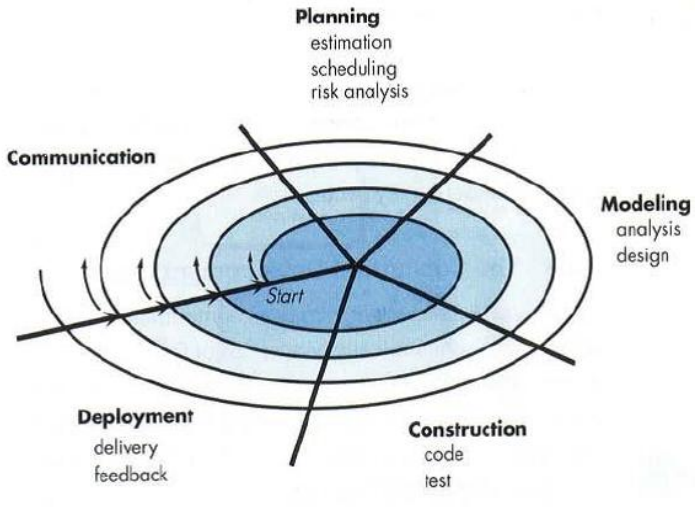
In Business needs, requirements change very often making earlier methods unrealistic and redundant. Short market deadlines make it difficult to complete comprehensive software products.

The evolutionary models are iterative and help the developers to complete short version within the given deadlines.

Ideally prototype serves as a mechanism to identify software requirements for working prototypes. The developer attempts to make use of existing program fragments and applies tools such as report generators which enable working programs to be generated quickly.

The software engineer & customer meet and define the overall objectives for the software, identify whatever requirements are known and outline areas where further definition is mandatory. Prototyping iteration is planned quickly and modeling (in the form of quick design) occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/end-user (e.g. human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed & then evaluated by the customer/user. Feedback is used to refine requirements for the software.

5. The Spiral Model:



Boehm (1988) viewed the software development process in light of risks involved, Spiral model could combine development activities with risk management to minimize and control the risk impact.

This is again an evolutionary model which couples iterative nature of prototyping with controlled and systematic aspects of the waterfall model. It also provides scope for RAD for increasingly complete software.

The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

From the figure given above, a spiral model is divided into a set of framework activities defined by the software engineering team. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. Anchor point milestones – a combination of work products and conditions that are attained along the path of the spiral – are noted for each evolutionary pass.

Each pass through the planning region results in adjustments to the project plan. Cost & schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

The initial circuit around the spiral can be for the concept development and with multiple iterations. The spiral traverses outward for new product development spiral development remains operative for the life span of software. This may be a realistic approach for large scale software development. As the process progresses both users and developers better understand the system. However the system, demands risks, identification and monitoring to prevent hurdles.

The spiral model can be adopted to apply throughout the life cycle of an application, from concept development to maintenance.

1.9. Agile Software Development:

Agile programming is an approach to project management, typically used in software development. It helps teams react to the instability of building software through incremental, iterative work cycles, known as sprints.

Features of the Agile Software Development Approach

The name "agile software process", first originated in Japan. The Japanese faced competitive pressures, and many of their companies, like their American counterparts, promoted cycle-time reduction as the most important characteristic of software process improvement efforts

Modularity

Modularity is a key element of any good process. Modularity allows a process to be broken into components called activities. A software development process prescribes a set of activities capable of transforming the vision of the software system into reality.

Activities are used in the agile software process like a good tool. They are to be wielded by software craftsman who know the proper circumstances for their use. They are not utilized to create a production-line atmosphere for manufacturing software.

Iterative

Agile software processes acknowledge that we get things wrong before we get them right. Therefore, they focus on short cycles. Within each cycle, a certain set of activities is completed. These cycles will be started and completed in a matter of weeks. However, a single cycle (called iteration) will probably not be enough to get the element 100% correct.

Time-Bound

Iterations become the perfect unit for planning the software development project. We can set time limits (between one and six weeks is normal) on each iteration and schedule them accordingly. Chances are, we will not (unless the process contains very few activities) schedule all of the activities of our process in a single iteration. Instead, we will only attempt those activities necessary to achieve the goals set out at the beginning of the iteration. Functionality may be reduced or activities may be rescheduled if they cannot be completed within the allotted time period.

Parsimony

Agile Process is more than a traditional software development process with some time constraints. Attempting to create impossible deadlines under a process not suited for rapid delivery puts the onus on the software developers. This leads to burnout and poor quality Instead, agile software processes focus on parsimony. That is, they require a minimal number of activities necessary to mitigate risks and achieve their goals.

Adaptive

During an iteration, new risks may be exposed which require some activities that were not planned. The agile process adapts the process to attack these new found risks. If the goal cannot be achieved using the activities planned during the iteration, new activities can be added to allow the goal to be reached. Similarly, activities may be discarded if the risks turn out to be ungrounded.

Incremental

An agile process does not try to build the entire system at once. Instead, it partitions the nontrivial system into increments which may be developed in parallel, at different times, and at different rates. We unit test each increment independently. When an increment is completed and tested, it is integrated into the system.

Convergent

Convergence states that we are actively attacking all of the risks worth attacking. As a result, the system becomes closer to the reality that we seek with each iteration. As risks are being proactively attacked, the system is being delivered in increments. We are doing everything within our power to ensure success in the most rapid fashion.

People-Oriented

Agile processes favor people over process and technology. They evolve through adaptation in an organic manner. Developers that are empowered raise their productivity, quality, and performance.

Collaborative

Agile processes foster communication among team members. Communication is a vital part of any software development project. When a project is developed in pieces, understanding how the pieces fit together is vital to creating the finished product. There is more to integration than simple communication. Quickly integrating a large project while increments are being developed in parallel requires collaboration.

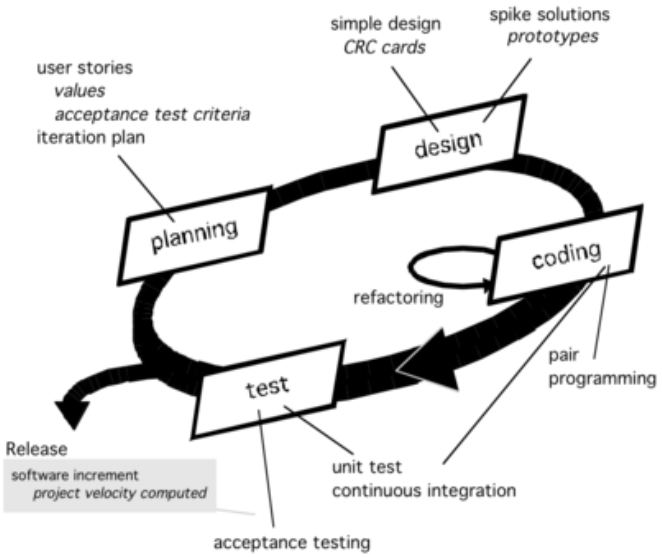
Concept of Extreme Programming

Extreme Programming is an instance of an Agile Software Development method. XP is a method that is optimized for small to medium-sized project teams that fit a certain profile. It promotes rapid feedback and response to continual change. It is based upon the four values of simplicity, communication, feedback, and courage and is consistent with the values of agile software development.

Characteristics of an XP Project

Extreme Programming or XP is a development process that can be used by small to medium-sized teams to develop high quality software within a predictable schedule and budget and with a minimum of overhead. Since XP relies heavily on direct and frequent communication between the team members, the team should be co-located. An ideal project for using XP would be one that has most of the following characteristics:

- A small to medium-sized team (fewer than 20 people on the complete team)
- Co-located, preferably in a single area with a large common space
- A committed, full-time, on-site customer or customer representative



The Extreme Programming Process

Goals

Extreme Programming Explained describes Extreme Programming as a software-development discipline that organizes people to produce higher-quality software more productively.

XP attempts to reduce the cost of changes in requirements by having multiple short development cycles, rather than a long one. In this doctrine, changes are a natural, inescapable and desirable aspect of software-development projects, and should be planned for, instead of attempting to define a stable set of requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Activities

XP describes four basic activities that are performed within the software development process: coding, testing, listening, and designing. Each of those activities is described below.

Coding

The advocates of XP argue that the only truly important product of the system development process is code – software instructions that a computer can interpret. Without code, there is no working product.

Coding can also be used to figure out the most suitable solution. Coding can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem, or finding it hard to explain the solution to fellow programmers, might code it in a simplified manner and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.

Testing

Main article: Test-driven development

Extreme programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws.

Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature.

Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements.

System-wide integration testing was encouraged, initially, as a daily end-of-day activity, for early detection of incompatible interfaces, to reconnect before the separate sections diverged widely from coherent functionality. However, system-wide integration testing has been reduced, to weekly, or less often, depending on the stability of the overall interfaces in the system.

Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Communication between the customer and programmer is further addressed in the Planning Game.

Designing

From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

Question Bank

1. Define Software, Software Engineering, and Process Technology. 4 Marks
2. State four characteristics of software. 4 Marks
3. Explain and differentiate between hardware and software. 4 Marks
4. State and explain with examples seven broad categories of software. (Changing nature of software). 4 Marks
5. Explain Software engineering, System Engineering and their relationship. 4 Marks
6. Explain Software Engineering as layered technology approach. 4 Marks
7. Using schematic diagram explain software process framework. 4 Marks
8. State and explain in brief generic process framework activities. 4 Marks
9. Explain PSP. 4 Marks
10. Explain TSP. 4 Marks
11. State and Explain Waterfall process model with their advantages and limitations. 4 Marks
12. State and Explain RAD process model with their advantages and limitations. 4 Marks
13. State and Explain Incremental process model with their advantages and limitations. 4 Marks
14. State and Explain Prototype process model with their advantages and limitations. 4 Marks
15. State and Explain Spiral process model with their advantages and limitations. 4 Marks
16. Define four attributes of good software. 4 Marks
17. State features of Agile Software development. 4 Marks
18. Explain concept of Extreme Programming (XP). 4 Marks

2. Chapter 2
Software Engineering Practices And Software Requirements Engineering
2.1. Software Engineering Practices – Definition, Importance, Essence

Software Engineering Practice:

- Software engineering deals with processes to ensure delivery of the software through management control of development process and production of requirement analysis models, data models process models, information products, reports and software documentation.
- Software Engineering practices consist of collection of concepts, principles, methods and tools that a software engineer calls upon on a daily basis.
- It equips managers to manage software projects and software engineers to build computer programs.
- Provides necessary technical and management know-how for getting the job done.
- Transforms a haphazard, unfocused approach into something that is more organized, more effective and more likely to achieve success.

Importance of Software Engineering practices:

The software engineering considers various issues like hardware platform, performance, scalability and upgrades.

The Essence of software engineering practices:

The essence includes understanding the problem, planning a solution, carrying out the plan and examining the results for accuracy.

1. Understand the problem (communication and analysis)
   - Who has a stake in the solution to the problem?
   - What are the unknowns (data, function, behavior)?
   - Can the problem be compartmentalized?
   - Can the problem be represented graphically?
2. Plan a solution (planning, modeling and software design)
   - Have you seen similar problems like this before?
   - Has a similar problem been solved and is the solution reusable?
   - Can sub problems be defined and are solutions available for the sub problems?
3. Carry out the plan: The design you've created serves as road map for the system you want to build. (Construction, Code generation)
   - Does the solution conform to the plan? Is source code traceable to the design model?
   - Is each component part of the solution probably correct? Have the design and code been reviewed or have correctness proofs been applied to the algorithm?
4. Examine the result for accuracy (testing and quality assurance)
   - Is it possible to test each component part of the solution?
   - Does the solution produce results that conform to the data, functions and features that are required? Has the software been validated against all stakeholder requirements?

2.2. Core Principles of Software Engineering (Statements & Meaning of each Principle)

1. **The reason it all exists:**

   The software system exists in the organization for providing value to its users with, the availability of hardware and software requirements. Hence all the decisions should be made by keeping this in mind.

2. **Keep it Simple, Stupid (KISS)**

   Software design is not a haphazard process. There are many factors considered in the design effort. The design should be straight forward and as simple as possible. This facilitates having a system which can be easily understood and easy to maintain.

   Simple doesn't mean quick and dirty. In fact, it requires lot of thought and effort to simplify multiple iterations of a complex task. This results in the advantage that the software is less error prone and easily maintainable.

3. **Maintain the vision**

   A clear vision is essential for the success of a software project. If the vision is missing, the project may end up of two or more minds. The team leader has a critical role to play for maintaining the vision and enforce compliance with the help of the team members.

4. **What you produce, others will consume**

   The design and implementation should be done by keeping in mind the user's requirements. The code should permit the system extension. Some other programmers debugging the code should not have any errors and satisfying all the user needs.

5. **Be open to future**

   The system with the long lifetime has more value. The industry standard software systems induce for longer. The system should be ready to accept and adapt to new changes. The systems which are designed by keeping in mind the future needs will be more successful and acceptable to the users.

6. **Plan ahead for reuse**

   Reuse saves time and efforts. The reuse of code and design is one of the advantages of object oriented technologies. The reuse of parts of the code helps in reducing the cost and time evolved, in the new software development.

7. **Think**

   Placing clear and complete thought before action almost always produces better results. With proper thinking, we are most likely to do it right. We also gain knowledge about how to do it right again. It becomes a valuable experience, even if something goes wrong, as there was adequate thought process. Hence when clear thought has gone into the system, value comes out, this provides potential rewards.

2.3. Communication Practices (Concept, Need of communication, Statements and Meaning of each Principle)

Effective communication among the technical peers, customers and other stakeholders, project managers etc. is among the most challenging activities that confront software engineers.

Before customers' requirements can be analyzed, modeled are specified they must be gathered through a communication.

Effective communication is among the most challenging activities that you will confront.

Communication Principles are

1. **Listen:** Try to focus on the speakers words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions.
2. **Prepare before you communicate:** Speed the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon.
3. **Someone should facilitate the activity:** Every communication meeting should have a leader to keep the conversation moving in a productive direction, to mediate any conflict that does occur and to ensure than other principles are followed.
4. **Face-to Face communication is best:** It usually works better when some other representation of the relevant information is present. For eg. A participant may create a drawing or a "strawman" document that serves as a focus for discussion.
5. **Take notes and document decisions:** Things have a way of falling into cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.
6. **Strive for collaboration :** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.
7. **Stay focused; modularize your discussion:** The more likely involved in any communication, the more likely that discussion will bounce from one topic to next. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.
8. **If something is unclear, draw a picture:** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.
9. **A) Once you agree to something, move on**. **B) If you can't agree to something, move on C) If a feature or function is unclear and cannot be clarified at the moment, move on.** : Communication, like any software engineering activity, takes time. Rather than iterating endlessly the people who participates should recognize that many topics require discussion and that "moving on" is sometimes the best the best way to achieve communication agility.
10. **Negotiation** is not a contest or a game. It works best when both parties win: There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

2.4. Planning Practices (Concept, Need of planning, basic activities included, statements and meaning of each principle.)

The planning activity encompasses of a set of management and technical practices that enable the software team to define a road map as it travels towards its strategic goals and tactical objectives. Like most things in life, planning should be conducted in moderation enough to provide useful guidance to the team.

Planning Principles:

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

2. **Involve stakeholders in the planning activity:** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, timelines and other project related issues.
3. **Recognize that the planning is iterative:** When the project work begins it's likely that few things may change. To accommodate these changes the plan must be adjusted, as a consequence. The iterative and incremental may dictate replanning based on the feedback received from users.
4. **Estimate based on what you know:** The purpose of estimation is to provide an indication of the efforts, cost, task duration and skillsets based on the team's current understanding of the work and past experience. If the information is vague or unreliable estimates will be equally unreliable.
5. **Consider the risk as you define the plan:** The team should define the risks of high impact and high probability. It should also provide contingency plan if the risks become a reality. The project plan should be adjusted to accommodate the likelihood of the risks.
6. **Be realistic:** The realistic plan helps in completing the project on time including the inefficiencies and change. Even the best software engineers commit mistakes and then correct them. Such realities should be considered while establishing a project plan.
7. **Adjust granularity as you define the plan:** Granularity refers to the level of details that is introduced as a project plan is developed. It is the representation of the system from macro to micro level. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments. A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date.
8. **Define how do you intend to ensure quality:** The plan should identify how the software team intends to ensure quality. If technical reviews are to be conducted, they should be scheduled.
9. **Describe how you intend to accommodate change:** Even the best planning can be obviated by uncontrolled change. The software team should identify how the changes are to be accommodated as the software engineering work proceeds. If a change is requested, the team may decide on the possibility of implementing the changes or suggest alternatives. The team should also access the impact of change on the development process and the changes in cost.
10. **Track and monitor the plan frequently and make adjustments if required:** Software projects fall behind schedule one day at a time. Therefore, make sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When shippage is encountered, the plan is adjusted accordingly.

The W5HH Principle:

Barry Boehm suggest an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the W5HH principle, which includes a series of questions:

- **W**hy is the system being developed?
- **W**hat will be done?
- **W**hen will it be accomplished?
- **W**ho is responsible for a function?

- **W**here they are organizationally located?
- **H**ow will the job be done technically and managerially?
- **H**ow much of each resource is needed?

"We think that software developers are missing a vital truth: most organizations don't know what they do. They think they know, but they don't know".

2.5. Modeling Principles

Concept of Software Modeling

In software engineering designers create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing, such as machine, we can build a model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, the model must take a different form. The model must be capable of representing the information that the software transforms, the architecture and functions that enable the transformation to occur, the features that the user desires and the behavior of the system as the transformation is taking place. The models must accomplish these objectives at different levels of abstraction. Initially the system is represented by depicting the software from the customer's point of view (Analysis model). The later the system is represented at a more technical level providing concrete specification for the construction of the software (Design model). The Design model represents the characteristics of the software which help the professionals to construct the software effectively.

In software engineer work, two classes of models are created:

Analysis Models

Design Models

Analysis Models represent the customer requirements by depicting the software in three domains

The information domain

The functional domain

The behavioral domain

Analysis models represent customer requirements

Design models provide a concrete specification for the construction of the software. It represents characteristics of the software that help practitioners to construct it effectively.

The architecture

The user interface

And component – level detail

The engineer's first problem in any design situation is to discover what the problem really is".

Analysis Modelling Principles:

Requirement models (also called analysis models) represent customer requirements by depicting the software three different domains: the information domain, the functional domain and the behavioral domain.

1. **The information domain of a problem must be represented and understood**

The information domain encompasses the data that flow into the system from end users, other systems or external devices, the data that flow out the system via the user interface, network interfaces, reports, graphics, and other means and the data stores that collect and organize persistent data objects i.e. data that are maintained permanently.

2. **The functions that the software performs must be defined.**

   Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions affect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction.

3. **The behavior of the software (as a consequence of external events) must be represented)**

   The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

4. **The models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered fashion.**

   Requirements modeling are the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirely.

5. **The analysis task should move from essential information toward implementation detail.**

   Requirements modeling begin by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. Implementation detail indicates how the essence will be implemented.

   Design Modeling Principles: The software design model is similar to the architect's plan or drawing for a house. It begins by representing the totality of the thing to be built and slowly refines the thing to provide guidance for constructing each detail. The design model created for the software provides variety of views of the system.

   "See first that the design is wise and just that ascertained, purpose it resolutely; do not for one repulse forego the purpose that you resolved to effect."

   – William Shakespear

The principles are:

1. **Design should be traceable to the requirements model.**

   The design model should translate the information into architecture; a set of subsystems which implement major functions and a set of component level designs are the realization of the analysis classes.

2. **Always consider the architecture of the system to be built**

   Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system.

3. **Design of data is as important as design of processing functions**

   The data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

4. **Interfaces (both internal and external) must be designed with care**

   The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

5. **User interface design should be tuned to the needs of the end user. However, in every case, it should stress case of use.**

   The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad".

6. **Component-level design should be functionally independent,**

   Functional independence is a measure of the "Single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive-that is, it should focus on one and only one function or sub function.

7. **Components should be loosely coupled to one another and to the external environment.**

   Coupling is achieved in many ways- via a component interface, be messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

8. **Design representations(models) should be easily understandable**

   The purpose of design is to communicate information to practioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

9. **The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity**

   Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

2.6. Construction Practices

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or the end user.

Even the software development process has undergone a radical change over the years.

In the model software engineering work the coding may be:

1. The direct creation of source code using a programming language.
2. Automatic generation of source code using an intermediates design like representation of the components to be built.
3. Automatic generation of executable code using 4GL language

Coding Principles and concept

The principle and concept that guide the coding task are closely aligned programming style, programming language, and programming methods. However, there are a number of fundamental principles that can be stated.

Preparation Principles:

Before you write one line of code, be sure you

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

Coding Principles:

1. As you begin writing code, be sure you:
2. Constrain your algorithms by following structured programming practice.
3. Consider the sue of pair programming
4. Select data structures that will meet the needs of the design
5. Understand the software architecture and create interfaces that are consistent with it.
6. Keep conditional logic as simple as possible.
7. Create nested loops in a way that makes theme easily testable.
8. Select meaningful variable names and follow other local coding standards
9. Write code that is self-documenting
10. Create a visual layout that aids understanding

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

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

Testing principles and concept:

In a classic book on software testing, Glen Mysers states a number of rules that can serve well as testing objectives:

Testing is a process of executing a program with the intent of finding an error.

A good "test-case" is the highest probability of finding an "as-yet undiscovered errors".

A successful test is a one which uncovers an as-yet undiscovered errors

Davis suggests a set of testing principles as follows:

1. **All tests should be traceable to customer requirements.**

   The objective of software testing is to uncover errors. It follows that the most server defects from the users point of view are those that cause the program to fail to meet its requirement.

2. **Tests should be planned long before testing begins**

   Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

3. **The Pareto principle applies to software testing**

   In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

4. **Testing should begin "in a small" and progress toward testing "in the large"**

   The initial testing should be on small individual components. As testing progresses, focus shifts to find errors in integrated clusters of programs ans finally in the entire system.

5. **Exhaustive testing is not possible**

   It may be noted that the number of path permutations for even a moderately sized program is exceptionally large. Hence for this reason it is impossible to execute every combination of the paths during testing.

2.7. Software Deployment

The deployment phase includes 3 actions namely 1. Delivery 2. Support 3. Feedback

1. The delivery cycle provides the customer and the end user with an operational software increment that provides usable functions and features.
2. The support cycle provides documentation, human assistance for all functions and features introduced during all deployment cycles to date.
3. Each feedback cycle provides the software team with useful inputs. The feedback can help in modifications to the functions, features and even the approach for the next increments.

The delivery of the software increment is an important milestone of any software project. A number of key principles should be followed as the team prepares to deliver an increment.

1. **Customer expectations for the software must be managed**

   Before the software delivery the project team should ensure that all the requirements of the users are satisfied.

2. **A complete delivery package should be assembled and tested**

   The system containing all executable software, support data files, tools and support documents should be provided with beta testing at the actual user side.

3. **A support regime must be established before the software is delivered**

   This includes assigning the responsibility to the team members to provide support to the users in case of problem.

4. **Appropriate instructional materials must be provided to end users**

   At the end of construction various documents such as technical manual, operations manual, user training manual, user reference manual should be kept ready. These documents will help in providing proper understanding and assistance to the user.

5. **Buggy software should be fixed first, delivered later.**

   Sometimes under time pressure, the software delivers low-quality increments with a warning to the customer that bugs will be fixed in the next release. Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low quality product caused them. The software reminds them every day.

2.8. Requirements Engineering

Requirement Engineering helps software engineers to better understand the problem they will work to solve. It includes the set of tasks that lead to an understanding of:

1. What will be business impact of the software?
2. What the customer wants exactly?
3. How end user will interact with the system software engineering and other project stakeholders all participate.

1. Inception

Inception means beginning. It is always problematic to the developer that from where to start.

The customer and developer meet and they decide overall scope and nature of the problem.

The aim is

1. To have the basic understanding of problem
2. To know the people who will use the software
3. To know exact nature of problem.

2. Elicitation

Elicitation means to draw out the truth or reply from anybody. In relation with requirement engineering, elicitation is a task that helps the customer to define what is required. To know the objectives of the system to be developed is a critical job.

a. Problem of Scope:

The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

b. Problem of understanding

- Sometimes both customer as well as developer has poor understanding of
- What is needed?
- Capabilities and limitations of the computing environment.

c. Problems of volatility

Volatility means change from one state to another. The customer's requirement may change time to time.

3. Elaboration

Elaboration means to work out in detail. The information obtained during inception and elicitation is expanded and modified during elaboration. Requirement engineering activity focuses on developing the technical model of the software that will include:

1. Functions
2. Features
3. Constraints

This is an analysis modeling action. It focuses on "How end users will interact with system".

4. Negotiation

It means discussion on financial and other commercial issues.

In this step customer, user and stakeholder discuss to decode:

- To rank the requirements
- To decide priorities
- To decide risks
- To finalize the project cost
- Impact of above on cost and delivery

5. Specification

The specification is the final work product produced by requirement engineer. The specification may take different forms: A written document, a set of graphical model, a collection of scenarios, a prototype, Mathematical model. It serves as the foundation for subsequent software engineering activities. It describes the function, performance of a computer-based-system, constraints that will govern its development.

6. Validation

Products are assessed for quality during validation period. Inconsistencies, omission, errors are detected and corrected. Work products conform to the standards established for the process, the project and the product. Clarification related to conflicting requirements, unrealistic expectations, etc.

7. Requirements management

Helps the project team identify, control and track requirements and changes to requirements at any time of the project proceeds.

Source traceability table – stakeholders, recognize multiple viewpoint

Dependency traceability table – work towards collaboration asking for questions

Subsystem traceability table

Interface traceability table.

2.9. SRS (Software Requirements Specifications)

A Software requirements specification (SRS), a requirements specification for a software system, is a description of the behavior of a system to be developed and may include a set of use cases that describe interactions the users will have with the software. In addition it also contains non-functional requirements. Non-functional requirements impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints)

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do. Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

The software requirements specification document enlists enough and necessary requirements that are required for the project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed or being developed. This is achieved and refined with detailed and continuous communications with the project team and customer till the completion of the software.

Software Requirements Specification Template

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents
Revision History

1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Project Scope
1.5 References

2. Overall Description
2.1 Product Perspective
2.2 Product Features
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies

3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)

4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces

5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Question Bank

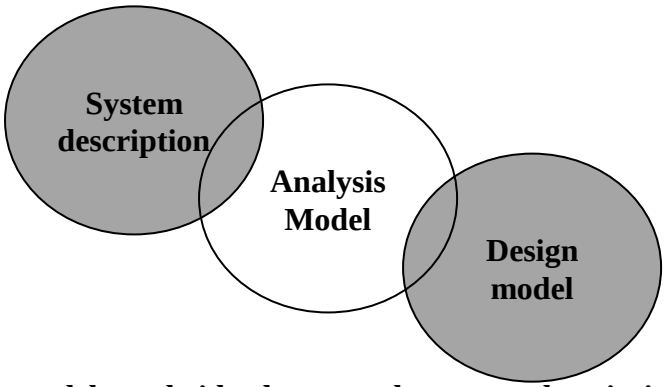
1.	Define SE practices, its importance.	3 Marks
2.	Staff briefly essence of SE Practices.	4 Marks
3.	State and describe Seven core Principles of software Engineering	4 Marks
4.	State and explain the communication Principles	8 Marks
5.	State and explain eight planning Principle	8 Marks
6.	Briefly explain Barry Boehm's W5HH Principle.	4 Marks
7.	Explain five Analysis modeling Principles.	4 Marks
8.	Explain eight design modeling Principles.	8 Marks
9.	Write a note on Construction practices	3 Marks
10.	Explain preparation, Coding & Validation Principles.	8 Marks
11.	State five set of S/W testing Principles.	4 Marks
12.	Explain S/W Deployment phases and state five principles.	8 Marks
13.	State and explain Seven RE tasks.	8 Marks
14.	Define SRS and give its contents.	8 Marks

3. Chapter 3
Analysis and Design Modeling

3.1. Analysis Modeling

The analysis model and requirements specification provide a means for assessing quality once the software is built.

Requirements analysis results in the specification of software's operational characteristics.



The analysis model as a bridge between the system description and the design model.

Objectives

Analysis model must achieve three primary objectives:

Describe customer needs

Establish a basis for software design

Define a set of requirements that can be validated once the software is built.

3.2. Analysis Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to overall understanding of software requirements and provide insight into the information, function, and behavior domains of the system.
- Delay consideration of infrastructure and other non-functional models until design.
  - For example, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- Minimize coupling throughout the system.
  - The level of interconnectedness between classes and functions should be reduced to a minimum.
- Be certain that the analysis model provides value to all stakeholders.
- Each constituent has its own use for the model.
  - Keep the model as simple as it can be.
  - Ex: Don't add additional diagrams when they provide no new information.
  - Only modeling elements that have values should be implemented.

3.3. Domain Analysis

Meaning

Software domain analysis is the identification, analysis and specification of common requirements from a specific application domain, typically for reuse in multiple projects within that application domain.

Object-oriented domain analysis is the identification, analysis and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

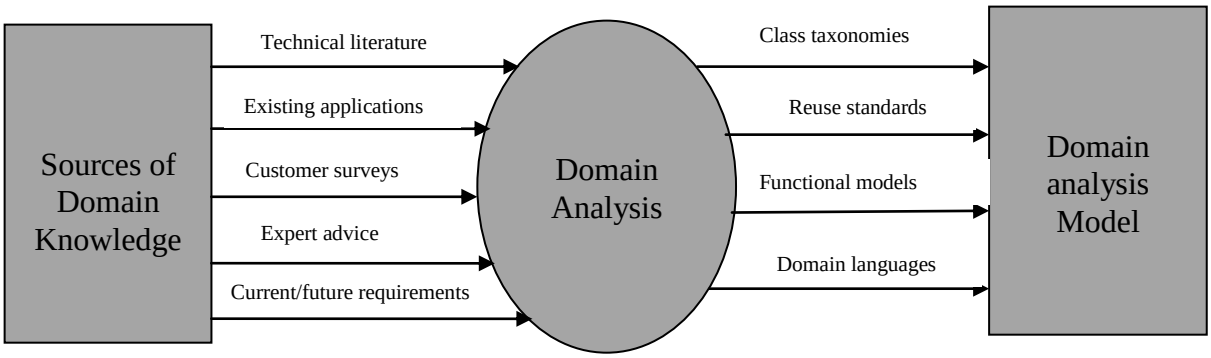
Example of Domain

The specific application domain can range from avionics to banking, from multimedia video game to software embedded within medical devices.

Goal

To find or create analysis classes and/or common functions those are broadly applicable, so that they may be reused.

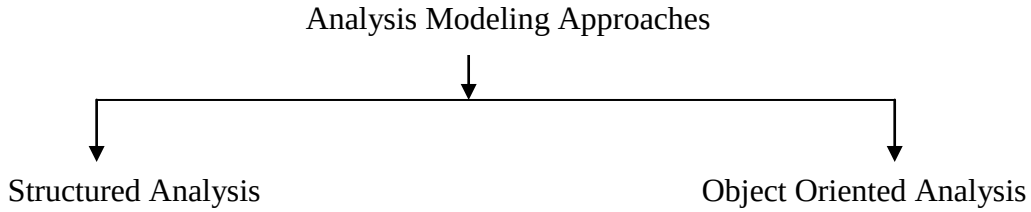
Input and Output of Domain Analysis



Input and Output for Domain Analysis

The role of domain analyst is to discover and define reusable analysis patterns, analysis classes and related information that may be used by many people working on similar but not necessarily the same applications.

Analysis Modeling Approaches

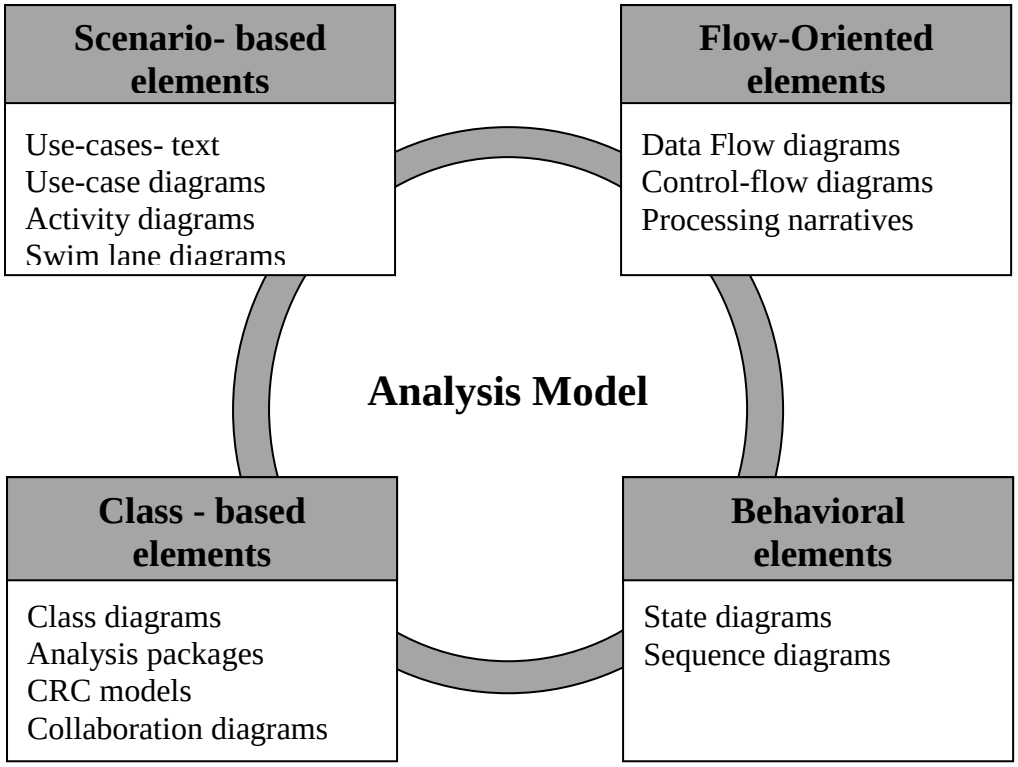


Structured Analysis

Structured Analysis considers data and the process that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate the data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

Object- oriented Analysis

Object-oriented Analysis focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.



Elements of the analysis model

The intent is to define all classes, relationship, behavior associated with them, that are relevant to the problem to be solved. To achieve this following task should occur.

Task 1. Basic user requirements must be communicated between user and developer.

Task 2. Classes must be identified (i.e. attributes, methods defined)

Task 3. A class hierarchy is defined

Task 4. Object- object relationships (object connection) should be represented.

Task 5. Object behavior must be modeled.

Task 6.Task-1 to Task-5 is reapplied iteratively till model is complete.

3.4. Building the Analysis Model

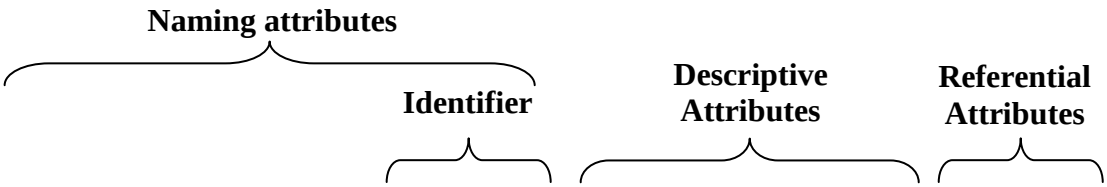
- **Data Modeling Concepts**

  Analysis modeling often begins with data modeling. The software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

- **Data Objects**

  A Data object is a representation of any composite information that must be understood by the software. A data object can be an external entity, a thing, an occurrence of event, a role, a unit, a place, a structure etc. A person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. In a data object i.e.

encapsulated data only – there is no reference within a data object to operations that act on the data. The data object can be represented in a table as given below.



Make	Model	ID#	Body type	Color	Owner

The body of the table represents specific instances of the data object.

- **Data Attributes**
  1. These define the properties of data object and take on one of three different characteristics
  2. Name an instance of the data object
  3. Describe the instance
  4. Make reference to another instance in another table
  5. Attributes may be car, id-number, body type, colour etc.

- **Data relationships**

  Data objects are connected with each other in different ways. The connection established between two objects because they are related. The relationships define relevant connection between two objects.

Connection Relationship

- **Cardinality and Modality with example**
- **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’ ie 1:1 or 1:N or M:N

  It also defines the max no. of objects that can participate in a relationship

- **Modality**

  Cardinality does not however indicate whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

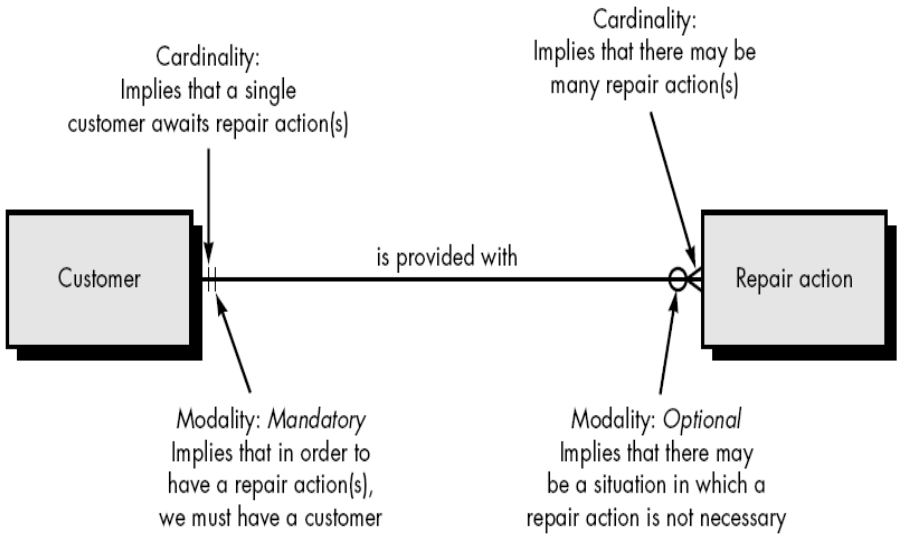
  The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.

  The modality is 1 if an occurrence of the relationship is mandatory.

Example

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.

Following figure illustrates the relationship, cardinality and modality between the data objects customer and repair action.



- **Flow Oriented Modeling- DFD**

  A **Data Flow Diagram** (DFD) is a graphical representation that depicts the information flow and the processes used for transformation as the data moves from input to output.

  Use

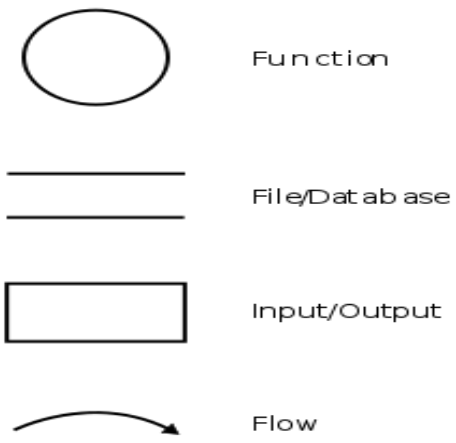
  The data flow diagram may be used to represent a system or software at any level of abstraction.

  DFD provides a mechanism for functional modeling as well as information flow modeling.

  A DFD shows what kinds of data will be input to and output from the system, where the data will come from and go to, and where the data will be stored.

  It does not provide information about the timing of processes, or information about whether processes will operate in sequence or in parallel (which is shown in a flowchart).

  Standard Notations



- A circle (bubble) represents a process or transformation which is applied to data (or control).
- The double line represents a data store - information that is used by the software.
- An arrow represents one or more data items (data objects). All arrows on a data flow diagram should be labeled.

Rules followed for preparing a Data Flow Diagram.

The level 0 data flow diagram (Context Diagram) should depict the software/system as a single bubble. For any application before drawing the detailed DFD, context diagram should be drawn.

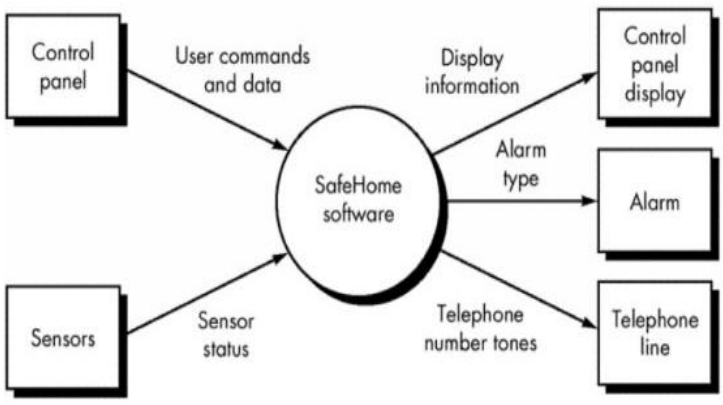
1. Primary input and output should be carefully noted.
2. All arrows and bubbles should be labeled with meaningful names.
3. Information flow continuity must be maintained from level-to-level.
4. One bubble at a time should be refined.
5. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level.

Safe Home Application

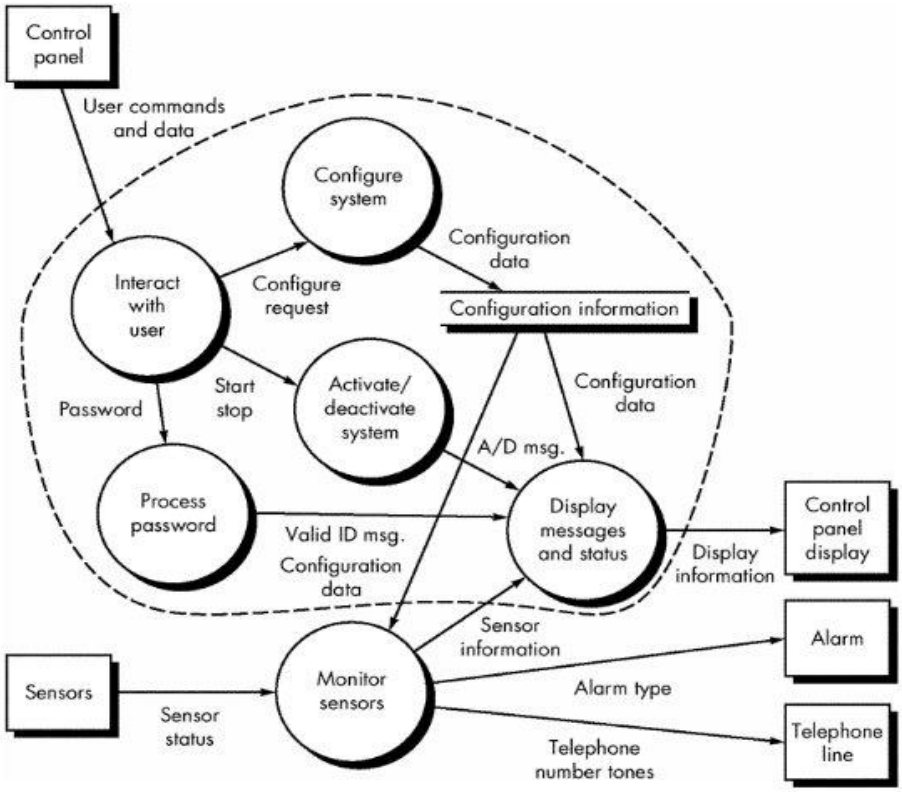
SafeHome software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the SafeHome control panel.

During installation, the SafeHome control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone numbers are input for dialing when a sensor event occurs.

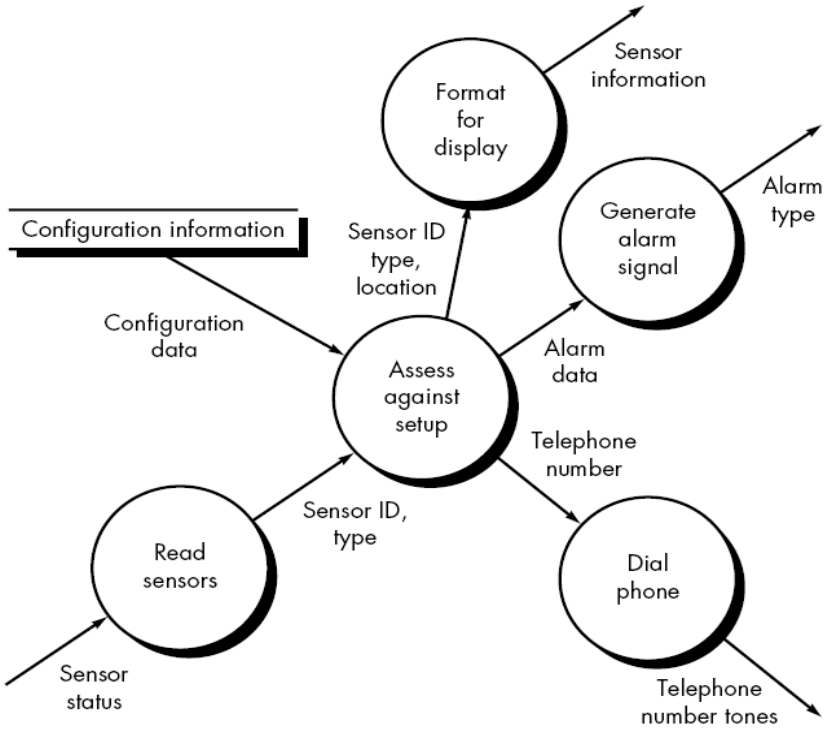
When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained. All interaction with SafeHome is managed by a user interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display.



Level 0 DFD for the SafeHome security function



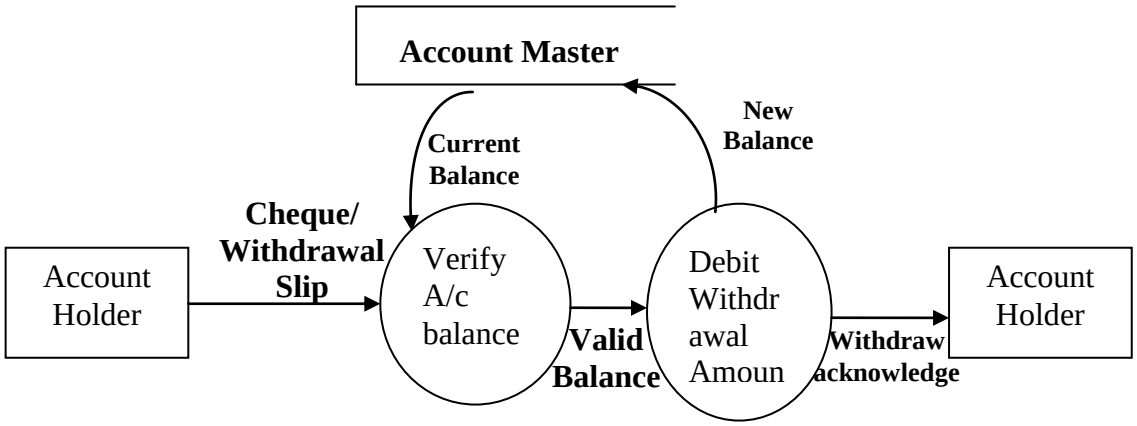
Level 1 DFD for the SafeHome security function



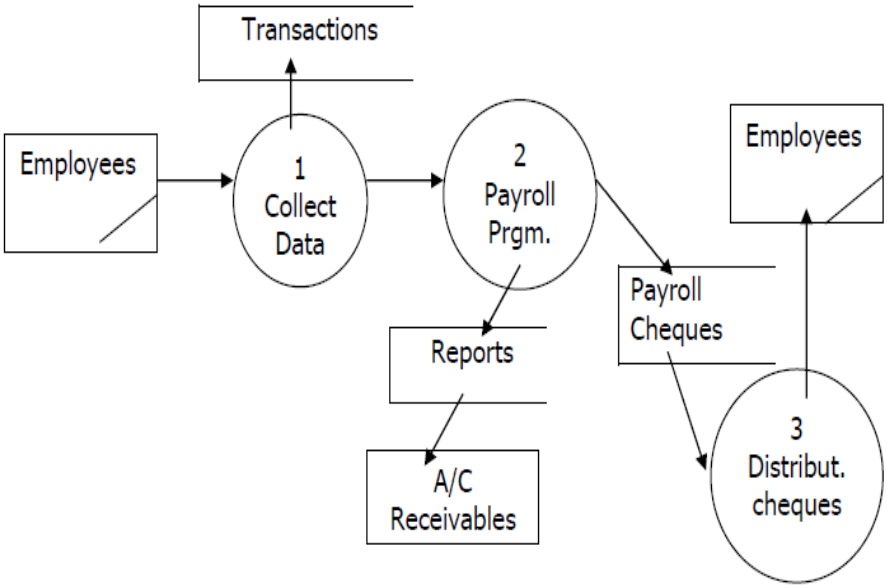
Level 2 DFD that refines the monitor sensors transform

Examples:

DFD for Bank Account



DFD for Payroll System



Data Dictionary (DD)

Meaning

Data dictionary is a repository of data. It stores all the details about data stores and the pointers to original data objects. It is mainly used as tool in the category of Structured Analysis. It is used as a central database for the description of all data objects. Once entries in this dictionary are defined, entity-relationship diagrams can be created and object hierarchies can be developed. Data flow diagramming features allow easy creation of this graphical model and provide features for the creation PSPECs and CSPECs.

The analysis model encompasses representations of data objects, function, and control. In each representation data objects and/or control items play a role. Therefore, it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary.

Data dictionary is a set of **meta-data** which contains the definition and representation of data elements.

Use

It gives a single point of reference of data repository of an organization. Data dictionary lists all data elements but does not say anything about relationships between data elements.

A database contains information about entities of interest to users in an organization.

When created, the database itself becomes an "entity" about which information must be kept for various data administration purposes.

Contents incorporated

Today, the data dictionary is always implemented as part of a Computer Aided Software Engineering ("CASE") structured analysis and design tool. Although the format of dictionaries vary from tool to tool, most contain the following information:

- Name-the primary name of the data or control item, the data store or an external entity.
- Alias- other names used for the first entry.
- Where- used/how-used-a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.
- Content description : a notation for representing content.
- Supplement information-other information about data types, preset values (if known), restrictions or limitations, and so forth.

Data dictionary (or system catalog) is database about the database.DD can be updated, queried much as a "regular" database. DBMS often maintains the DD.

Advantages of Data Dictionary

- Provides a summary of the structure of the database
- Helps DBA manage the database
- Informs users of database scope
- Facilitates communication between users and database administration
- Allows the DBMS to manage the creation, access, and modification of tables and their components

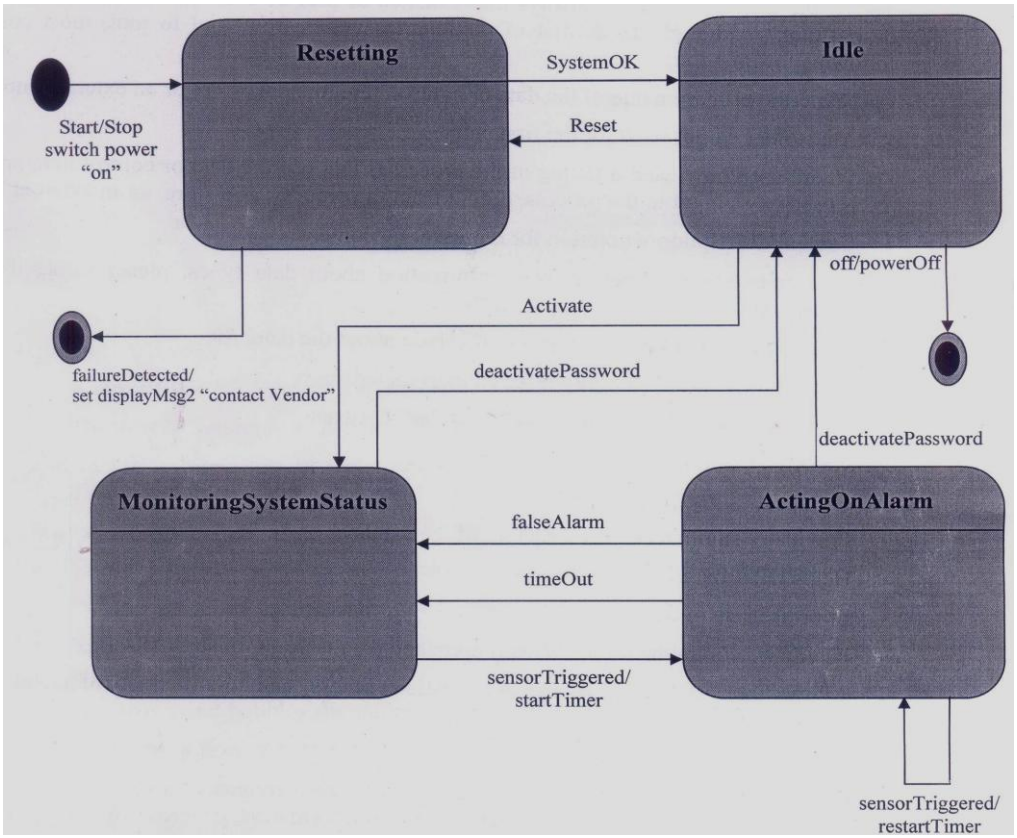
Creating a Control Flow Model

DFD is essential for meaningful insight into software requirements. Large-class applications are driven by events rather than data. They produce control information, process information with heavy concern for time and performance. Such applications require the use of control flow modeling in addition to data flow modeling.

1. List all sensors that are read by the software.
2. List all interrupt conditions
3. List all switches that are activated by actuators.
4. List all data conditions
5. Review all 'control items' as possible for control flow input/output
6. Describe the behavior of a system by identifying its states, identify each state is reached and define the transition between states.
7. Focus on possible omissions which is a very common error in specifying control.

Creating Control Specification (CSPEC)

A control specification (CSPEC) represents the behavior of the system in two different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table – a combinatorial specification of behavior. The following diagram represents CSPEC for the Safe Home Application.



The above diagram depicts a preliminary state diagram for the level 1 control flow model for SafeHome. The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, you can determine the behavior of the system and more important, ascertain whether there are "holes" in the specified behavior.

The state diagram indicates that the transition from the **Idle state** can occur if the system is reset, activated, or powered off. If the system is activated a transition to the **Monitoring System Status state** occurs, display messages are changed and the process Monitor and Control System is invoked. Two transition occurs out of the Monitoring System Status state.. When the system is deactivated, a transition occurs back to the Idle state. When a sensor is triggered into the **Acting On Alarm state.** All transitions and the content of all states are considered during the review.

Creating Process Specification (PSPEC)

This process 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 Unified Modeling Language (UML) activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a mini-SPEC that can serve as a guide to the design of the software component that will implement the process.

- **Scenario – Based Modeling**

Developing Use Cases

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 who use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people play as the system operates. Thus an actor is anything that communicates with the system or product and that is external to the system. Every actor has one or more goals while using the system. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities that play one role in the context of the use case.

Eg. 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 for interaction: programming mode, test mode, monitoring mode and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learnt about the system.

Once actors have been identified, Use Cases can be developed. Rules for developing Use Cases

Who is the primary actor, the secondary actors?

What are the actor's goals?

What preconditions should exist before the story begins?

What main tasks or functions are performed by the actor?

What exceptions might be considered as the story is described?

What variations in the actor's interaction are possible?

What system information will be actor acquire, produce, or change?

Will the actor have to inform the system about changes in the external environment?

What information does the actor desire from the system?

Does the actor wish to be informed about unexpected changes?

What is Use Case?

A use case diagram at its simplest is a representation of a user's interaction with the system and depicting the specifications of a use case. A use case diagram can portray the different types of users of a system and the various ways that they interact with the system. This type of diagram is typically used in conjunction with the textual use case and will often be accompanied by other types of diagrams as well.

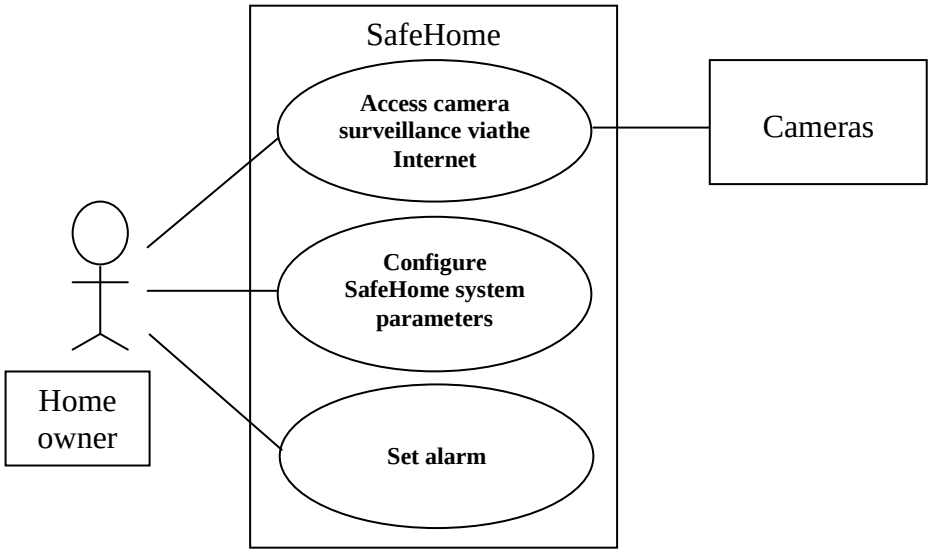
Writing Use Case

1. What should we write about?
2. Inception and elicitation provide us the information we need to begin writing use cases.
3. How much should we write about it?
4. How detailed should we make our description?
5. How should we organize the description?

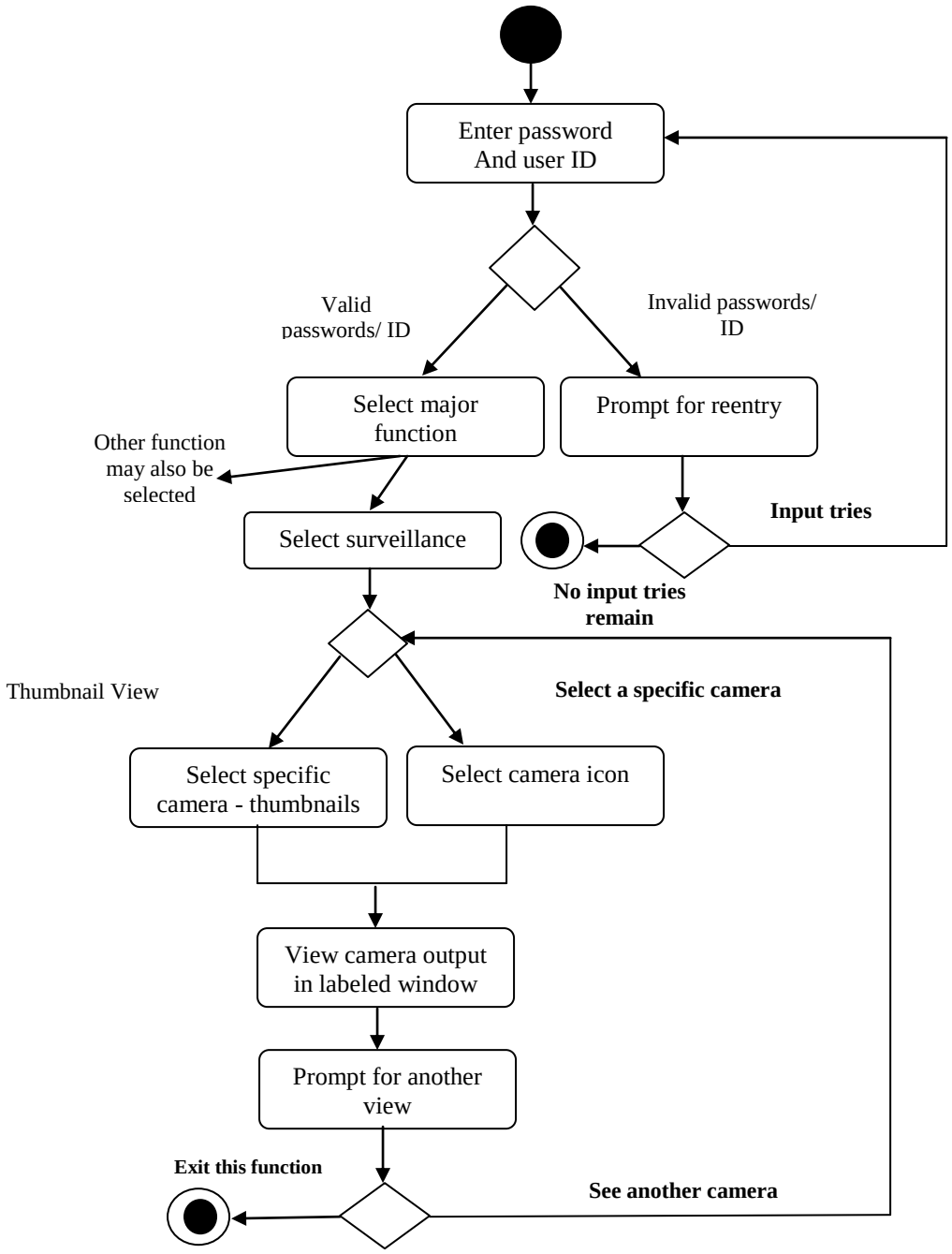
Developing an Activity Diagram

- What are the main tasks or functions that are performed by the actor?
- What system information will the actor acquire, produce or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Preliminary Use-Case Diagram for the SafeHome System



Activity diagram for Access camera surveillance via the Internet- display camera views function



- **Creating a behavioral Model**

  The behavioral model indicates how software will respond to external events or stimuli. To create the model, following are the 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 behavorial model to verify accuracy and consistency.

State Representations

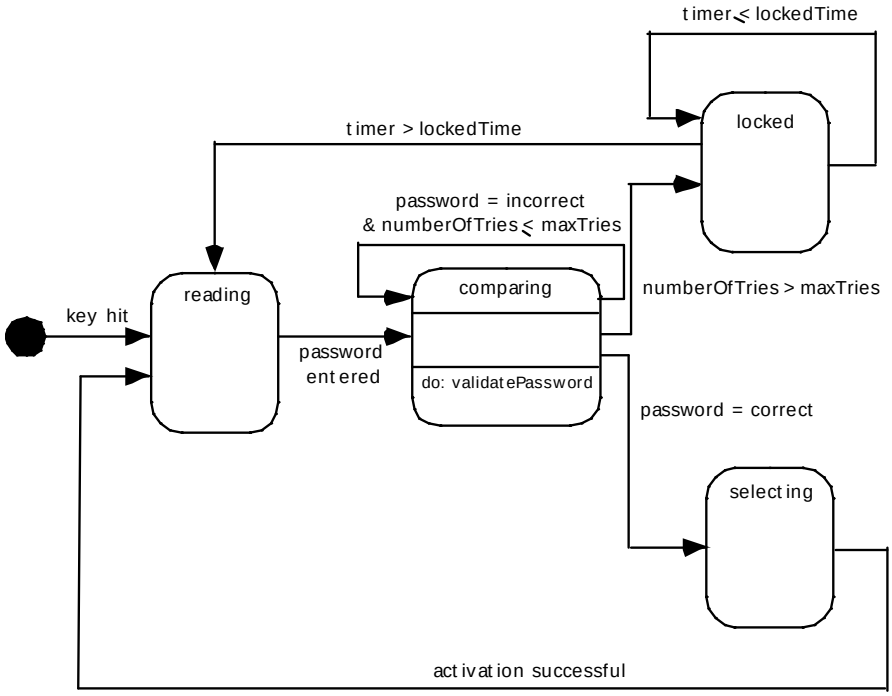
In the context of behavioral modeling, two different characterizations of states must be considered:

1. the state of each class as the system performs its function and
2. the state of the system as observed from the outside as the system performs its function

The state of a class takes on both passive and active characteristics

1. A passive state is simply the current status of an object's all attributes.

The active state of an object indicates the current status of the object as it undergoes a continuing transformation or processing.



The States of a System

- State—a set of observable circumstances that characterizes the behavior of a system at a given time
- State transition—the movement from one state to another
- Event—an occurrence that causes the system to exhibit some predictable form of behavior
- Action—process that occurs as a consequence of making a transition

3.5. Design Modeling

Design Process

Software design is an iterative process through which requirements are translated into a **"blueprint"** for constructing the software. The design is representation at a high level of abstraction – data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

There are three characteristics that serve as a guide for the evaluation of a good design:

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

Design Quality Guidelines

1. A design should exhibit an architecture that a) has been created using recognizable architectural styles or patterns, b) is composed of components that exhibit good design characteristics and c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular, i.e. 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

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:

- **Functionality**: is assessed by evaluating the features 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.
- **Reliability**: is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.
- **Performance**: is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability**: combines the ability to extend the program extensibility, adaptability, serviceability maintainability, testability, compatibility, configurability, etc.

Design Concepts

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time.

Following are the software concepts that span both traditional and object- oriented software development.

Abstraction :

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 we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word open for a door.

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, weight).

Architecture :

Software architecture alludes to the "overall structure of the software and the ways in which the structure provides conceptual integrity for a system."

In its simplest from, architecture is the structure of organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

The goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.

A set of architectural patterns enable a software engineer to reuse design-level concepts.

The architectural design can be represented using one or more of a number of different models.

Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

Patterns :

A design pattern "conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns."

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. Whether the pattern is applicable to the current work,
2. Whether the pattern can be reused, and

3. Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Modularity :

Software architecture and design patterns embody modularity; that is, software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

Monolithic software (large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

It is the compartmentalization of data and function. It is easier to solve a complex problem when you break it into manageable pieces. "Divide-and-conquer"

Don't over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration "Cost".

Information Hiding :

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) 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.

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. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

Functional Independence :

The concept of functional Independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Design software such that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure. Functional independence is a key to good design, and 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. Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world in "lowest possible" way.

Refinement :

It is the elaboration of detail for all abstractions. It is a top down strategy. A program is developed by successfully refining levels of procedural details. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

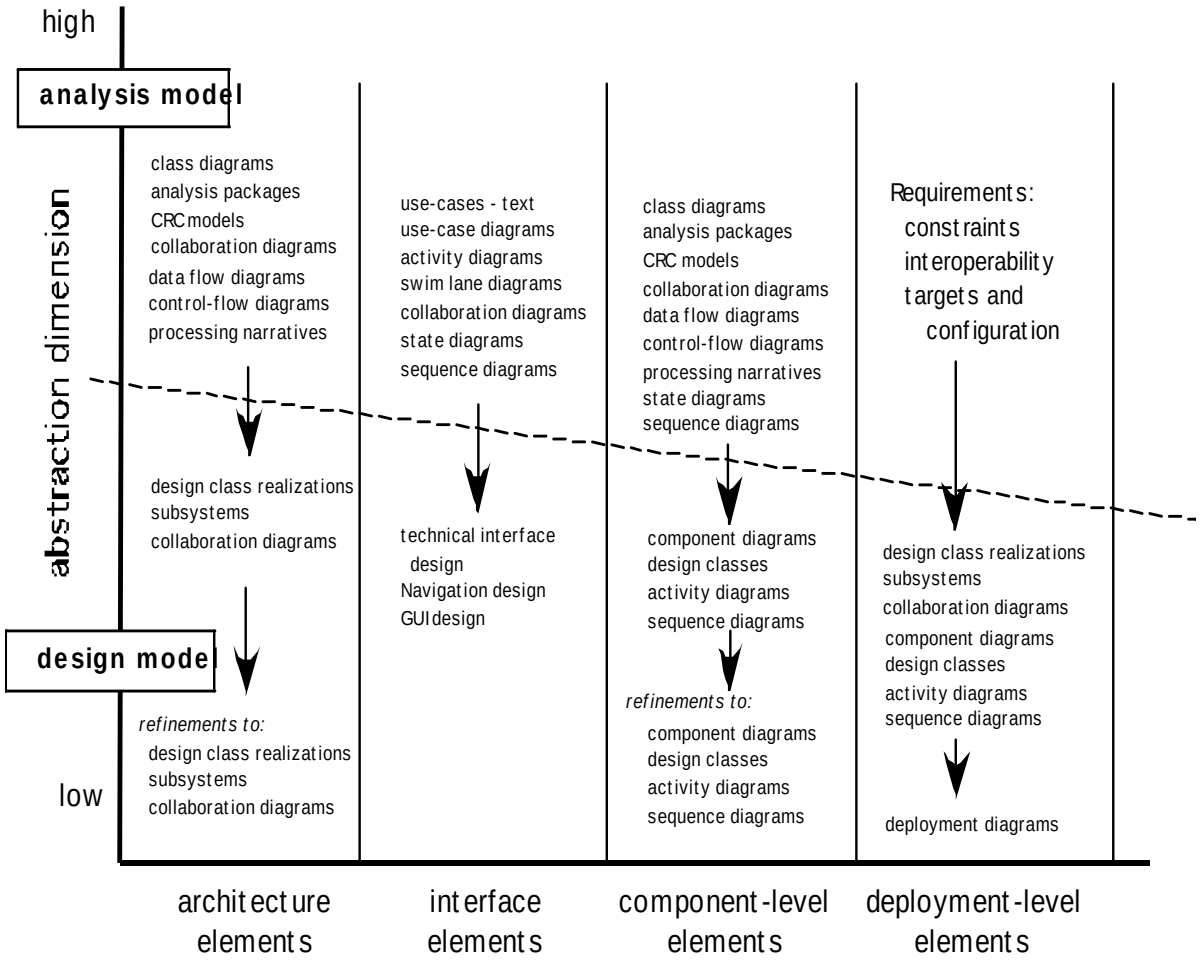
We begin with a statement of function or data that is defined at a high level of abstraction. The statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data. Refinement causes the designer to elaborate on the original statement, providing more and more details

with successive refinement (elaboration). Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refactoring :

It is a reorganization technique that simplifies the design of a component without changing its function or behavior. When software is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed data structures, or any other design failures that can be corrected to yield a better design.

3.6. The Design Model



- **Data design elements**

1. Data model --> data structures
2. Data model --> database architecture

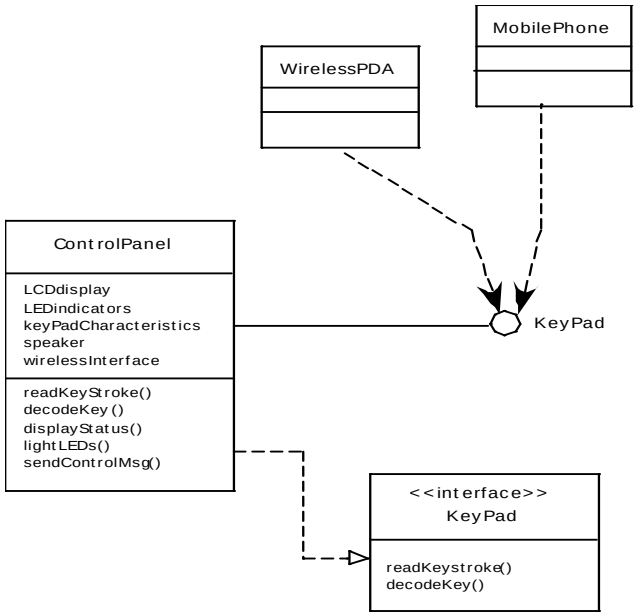
Data design or data architecting creates a model of data which is represented at higher level of abstraction. The data is refined progressively into more implementation specific representation. The data architecture has profound influence on software architecture. At the application level it is treated as database and in business as data warehouse.

Architectural design elements are similar to floor plan of the house. Architectural model is derived from information about the software to be built. Analysis model elements i.e. DFD, analysis classes, availability of architectural patterns and styles.

Interface design elements

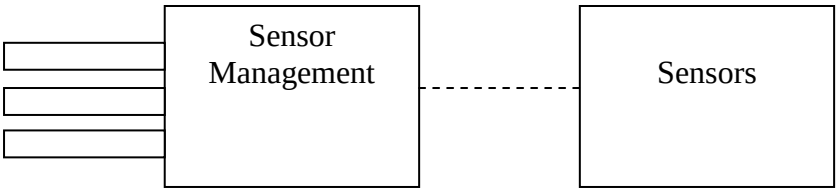
Equivalent to a set of detailed for doors, windows external drawings utilities. This provides details of information few to and from the system among the system components. The three important part of interface design are

- The user interface
- External interfaces to other systems, devices, networks
- Internal interfaces among various components.



Component level design elements

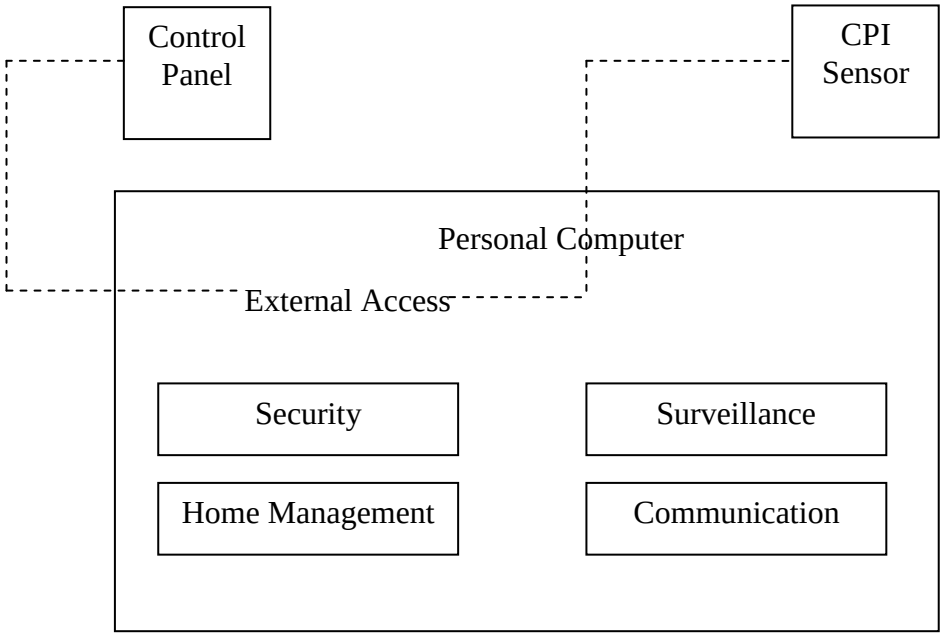
It is equivalent to a set of detailed drawings and specs for each room in a house. It carefully defines every detail of the house. This should fully describe the internal detail of each software component. It also includes all local data objects and algorithmic all processing that occurs within a component and an interface which allows access to all component operations.



UML component diagram for Sensor Management

- **Deployment level design elements**

This indicates how software functionally and subsystem terms will be allocated within the physical computing environment that will support the software. The subsystems housed within each computing element are indicated. The external subsystems are designed to manage all attempts to access the system from outside sources. Each subsystem would be elaborated to indicate the components that implement functions.



UML Deployment Diagram

Question Bank

1. State three objectives of Analysis model — 3 Marks
2. State six analysis rules of thumb — 3 Marks
3. Describe using a diagram inputs and outputs of a domain analysis — 4 Marks
4. Explain elements of the analysis model using a diagram — 4 Marks
5. Define data objects, attributes, relationship, cardinality and modality with example each — 4 Marks
6. Define DFD, give symbols and their meaning — 4 Marks
7. State six rules to be followed while constructing a DFD — 4 Marks
8. Define data dictionary, meaning, use and list advantages of Data Dictionary — 4 Marks
9. State the steps of creating and Control Flow Model — 4 Marks
10. Define CSPEC with an example — 4 Marks
11. Define PSPEC with an example — 4 Marks
12. Define Use Case with an example — 4 Marks
13. Explain activity diagram for Access Camera Survillana in a Safe Home System — 4 Marks
14. Describe the behavioral model with an example of State diagram — 4 Marks
15. Define design process with three characteristics — 4 Marks
16. State eight design quality guidelines — 4 Marks
17. State FURPS design quality attributes — 4 Marks
18. Define Abstraction, Architecture, Pattern, Modularity, Information hiding, Functional Independence, Refinement, Refactoring — 8 Marks
19. Define
    a. Data Design element — 3 Marks
    b. Interface design element — 3 Marks
    c. Component level design elements — 3 Marks
    d. Deployment level design elements — 3 Marks

4. Chapter 4
Software Testing Strategies and Methods
4.1. Software Testing

- **Definition of Software Testing**

  Well planned series of steps that result in successful construction of software.

  Testing is an individualistic process and the types vary depending on the development approaches. **It is defense** against programming errors. To avoid any inherent coding errors, several distinct approaches known as approaches **or** philosophies are used. This is called Strategic approach to software testing.

Good Test:

1. A good test has a high probability of finding an error.
2. The tester must understand the software and how it might fail.
3. A good test is not redundant.
4. Testing time is limited; one test should not serve the same purpose as another test.
5. A good test should be the "**best of the breed".**
6. Tests that have the highest likelihood of uncovering a whole class of errors should be used.
7. A good test should be neither too simple nor too complex.
8. Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors.

Successful Testing Strategies:

Test Plan:

A Test plan is a document detailing a systematic approach for testing a system such as a machine or software. The plan typically contains a detailed understanding of what the eventual workflow will be.

A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from Test Engineers.

Test Case:

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly. The mechanism for determining whether a software program or system has passed or failed such a test is known as a **test oracle**. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as test scripts, particularly when written. Written test cases are usually collected into test suites.

Test Data:

Test Data Generation, an important part of software testing, is the process of creating a set of data for testing the adequacy of new or revised software applications. It may be the actual data that has been taken from previous operations or artificial data created for this purpose.

4.2. Characteristics of Testing Strategies

1. To perform effective testing, a software team should conduct effective **Formal Technical Reviews (FTRs)** using which many errors are eliminated before testing.

2. Testing begins at the component level and works toward the integration of the entire computer based system.
3. Different testing techniques are appropriate at different points of time.
4. Testing is conducted by the developer as well as the individual group.
5. Testing and debugging are different activities which must be accommodated in any testing strategies.

4.3. Software Verification and Validation (V & V)

Verification is a quality control process that is used to evaluate whether or not a product, service, or system complies with regulations, specifications, or conditions imposed at the start of a development phase. Verification can be in development, scale-up, or production. This is often an internal process.

Validation is a Quality assurance process of establishing evidence that provides a high degree of assurance that a product, service, or system accomplishes its intended requirements. This often involves acceptance of fitness for purpose with end users and other product stakeholders.

Set of activities that ensure correct implementation of software function validation refers to different set of activities that has been built is trouble for customer requirements.

Differences between Verification and Validation:

Sr. No.	Verification	Validation
1.	Verification is a static practice of verifying documents, design, code and program	Validation is a dynamic mechanism of validating and testing the actual product
2.	It does not involve executing the code	It always involves executing the code
3.	It is human based checking of documents and files	It is computer based execution of program
4.	Verification uses methods like inspections, reviews, walkthroughs, and desk-checking etc.	Validation uses methods like black box testing, gray box testing, and white box(structural) testing etc.
5.	Verification is to check whether the software conforms to specifications	Validation is to check whether software meets the customer expectations and requirements
6.	It can catch errors that validation cannot catch. It is low level exercise	It can catch errors that verification cannot catch. It is high level exercise
7.	Target is requirements specification, application and software architecture, high level, complete design and database design etc.	Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8.	Verification is done be development team to provide that the software is as per the specifications in the SRS document	Validation is carried out with the involvement of client and testing team.
9.	It, generally, comes first-done before validation.	It generally follows after verification

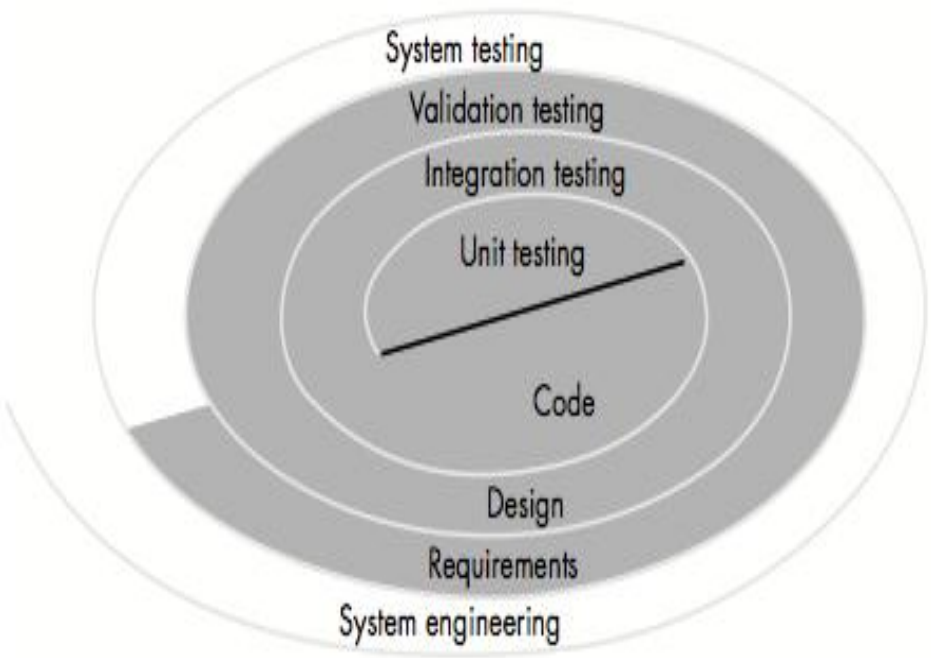
10.	**Question** Are we building the product right?	**Question** Are we building the right product?
11.	**Evaluation Items** Plans, requirements specification, Design Specification, Code and test cases	**Evaluation Items** The actual product/software
12.	**Activities** Reviews, Walkthroughs, Inspections	**Activities** Testing

4.4. Testing Strategies

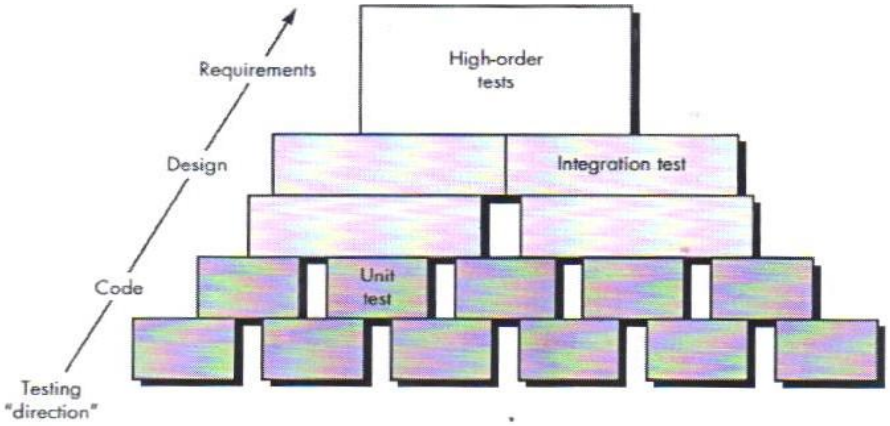
Strategy for Conventional Software Architecture:-

Code unit testing, integration testing, validation testing, system testing

A software process may be viewed as the spiral illustrated in the diagram below. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information the role of software and leads to software requirements analysis, in which the information domain, function, behavior, performance, constraints and validation criteria for software are established.



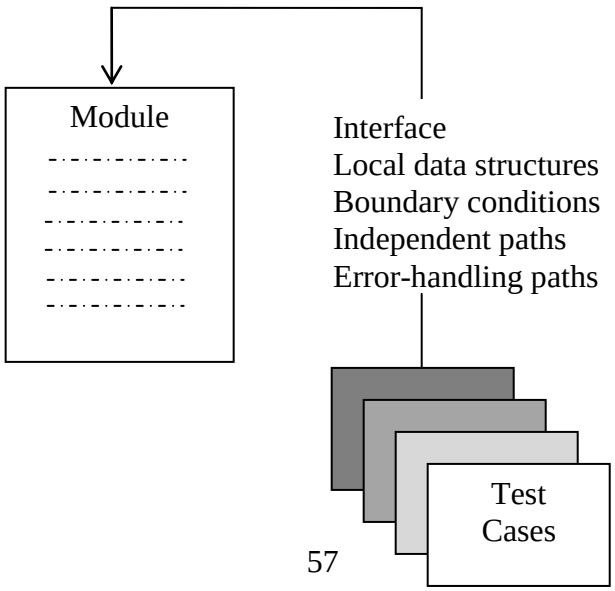
A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit of the software for the implementation in Source Code. The testing progresses by moving outward along the Spiral to integration testing. In this the focus is on design and the construction of the software architecture. Taking another turn outward on the Spiral, validation testing is a part where requirements established as part of requirements modeling which are validated against the software that has been constructed. Finally there is the system testing, where the software and other system elements are tested a whole.



Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. Initially, the tests focus on each component individually, ensuring that they function properly as a unit. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Integration testing addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated, a set of high-order tests is conducted. Validation testing provides final assurance that software meets all informational, functional, behavioral and performance requirements. Software once validated, must be combined with other system elements. System testing verifies that all elements mesh properly and that the overall system functions are achieved.

Testing Strategies for Conventional Software :

- Completeness and Consistency must be assessed as they are built in
- Testing must include error discovery techniques.
- Testing strategies and factors are the unique characteristics of object-oriented software.
- Begins **" testing in the small " → " testing in the large"**
- *Series of regression tests are run to uncover errors due to* communication and collaboration between the classes and addition of new classes.
- Finally entire system as a whole is tested to ensure that errors in requirements are uncovered.

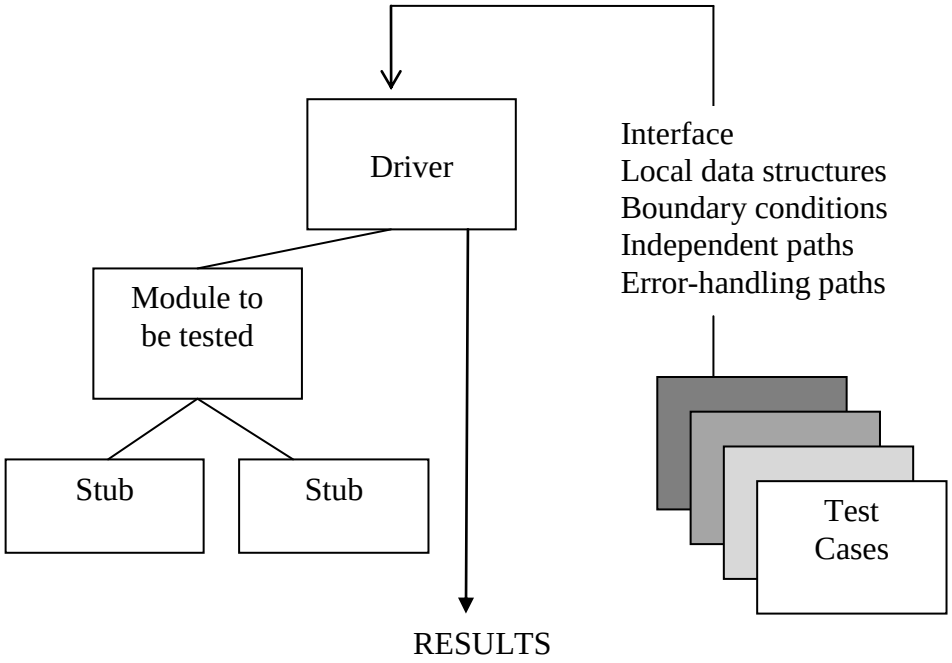


- **Unit Testing**

  Verification efforts of smallest unit of software design. Unit testing is considered as an adjunct to the coding step. Design information should provide guidance for establishing the test cases that are likely to uncover the errors in each of the categories. Each test case should be coupled with a set of expected results. The unit test is white-box oriented, and the step can be conducted in parallel for multiline

  Unit Test Consideration

  The module interface is tested to ensure that information properly flows into and out of the program unit test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.



  Unit Test Procedures

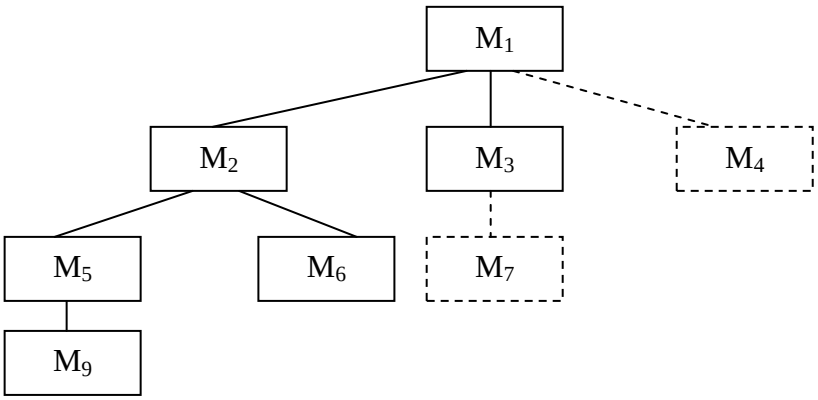
  Unit Testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case designs begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors. Each test case should be coupled with a set of expected results.

- **Integration Testing**

  It is a systematic technique of constructing the software architecture while at the same time conducting test to uncover errors associated with interfacing. The objective is to take tested components and build a program structure that has been dictated by the system.

- **Top down Integration**

  This is an incremental approach to construction of software architecture. Modules are integrated by moving downward through the control hierarchy beginning with the main programme. Modules are then incorporated into the structure with depth-first/breadth first manner.



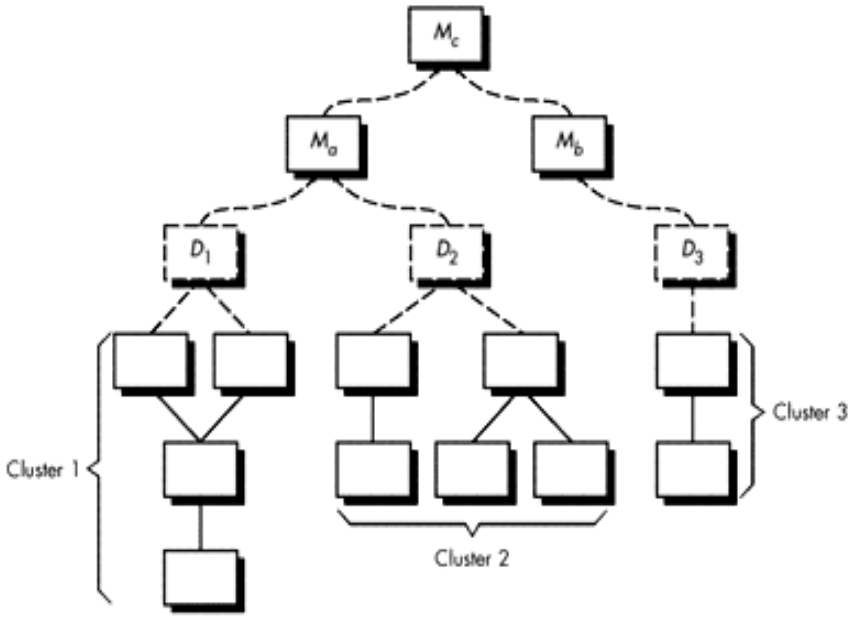
  The depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. Selecting the left-hand path, components M_1, M_2, M_5 would be integrated first. Next, M_8 or M_6 would be integrated. Then, the central and right- hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. Components M_2, M_3 and M_4 would be integrated first. The next control level, M_5, M_6 and so on, follows. The integration process is performed in a series of five steps:

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

- **Bottom-Up integration**

  This begins with the construction a test of small modules. The components are integrated from the bottom-up, the functionally provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

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



In the above figure, components are combined to form clusters 1, 2 and 3. Each of the clusters is tested using a driver. Components in cluster 1 and 2 are subordinate to Ma. Driver D_1 and D_2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both M_a and M_b will ultimately be integrated with component M_c and so forth. As integration moves upward, the need for separate test rivers lessens. If the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

- **Regression Testing**

  Each time when a new module is added as a part of integration testing, the software changes, new data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the re-execution of some subset tests that have already been conducted to ensure that changes have not propagated unintended side effects. In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspects of the software configuration (the program, its documentation, or data that support it) are changes. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

  Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

  The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

  1. A representative sample test that will exercise all software functions.
  2. Additional test that focus on software function that are likely to be affected by the change.
  3. Tests that focus on software components that have been changed.

- **Smoke Testing**

  Smoke testing is an integration testing approach that is commonly used when "shrinkwrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a fragment basis. In essence, the smoke testing approach encompasses the following activities:

  1. Software components that have been translated into code are integrated into a build. They will include files, libraries, reusable modules; engineered components, which are required, implement one or more product functions.
  2. Series of test designed to expose earns that will keep the build from properly performing the function. The intent is to uncover "show stopper" errors.
  3. The build is integrated with other builds and the entire product is smoke tested daily. The approach may be either top down/bottom up.

  The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing, major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly. Smoke testing provides a number of benefits when it is applied on complex, Time critical software engineering projects. They are

  a. Integration risk is minimized
  b. Quality of end product is increased
  c. Error diagnosis and correction are simplified
  d. Progress is easier to assess

4.5. Alpha and Beta Testing

It is generally difficult for a software designer to foresee, how the customer will really use a program instructions for use may be misinterpreted, strange combinations of data may be regularly used, output that seemed clear during testing may be intangible to a user in the field.

If the software developed to be used by many customers, it is impractical to perform formal acceptance test with each one. Most software product designs use a process called Alpha and Beta testing to uncover errors that only the end user seems able to find.

The 'Alpha' test is conducted at the developed site by the end users. The software is used in a natural setting with the developer looking over the shoulder of typical users and recording errors and usage problems. "Alpha" tests are conducted in a controlled environment.

"Beta test" is conducted at end user sites. It is the 'live' application of the software on an environment that cannot be controlled by the developer. The end user records all problems (real or imagined) that are encountered during beta testing and reports this to the developer at regular intervals. Using this feedback software engineers make modifications and then prepare for release of the software product to the entire customer base.

Difference between Alpha and Beta Testing

Alpha Testing	Beta Testing
Alpha testing conducted at Developer Site by End user	Beta Testing is conducted at User site by End user
Alpha testing is conducted in Control Environment as Developer is present	Beta Testing is conducted in Un-control Environment
Less Chances of finding an error as Developer usually guides user	More Chances of finding an error as Developer can use system in any way

It is kind of mock up testing	The system is tested as Real application
Error/Problem may be solved in quick time if possible	The user has to send difficulties to the developer who then corrects it.
Short process	Lengthy Process

4.6. System Testing

- **Concept of System Testing**

  System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

  The software engineer should anticipate potential interfacing problems and 1) design error-handling paths that test all information coming from other elements of the system, 2) conduct a series of tests that simulate bad data or other potential errors at the software interface, 3) record the results of tests to use as "evidence" if finger-pointing does occur and 4) participate in planning and design of system tests to ensure that software is adequately tested.

- **Types of System testing (Recovery, Security, Stress, Performance Testing) with examples**

  a. Recovery Testing

  Many computer-based systems recover from the fault and resume processing within a pre specified time for the system should be "fault" tolerant. The recovery testing for the software to fail in a variety of ways and verifies that the recovery is properly performed. If the recovery is automatic, re-initialization, check pointing mechanism, data recovery and restart are evaluated for correctness. If the recovery needs human intervention, mean-time-to-repairs (MTTR) is evaluated to find whether it is within acceptable limits.

  b. Security Testing

  Security testing attempts to verify that protection mechanisms built into a system protect it from improper penetration. During security testing, the tester plays the role of the individual who desires to penetrate the system. The tester may attempt to acquire passwords through external clerical means, may attack the system with custom software designed to break down any defenses that have been constructed, may overwhelm the system, thereby denying service to others, may purposely cause system errors, hoping to penetrate during recovery, may browse through insecure data, hoping to find the key to system entry.

  c. Stress Testing

  Stress tests are designed to confront programs with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example 1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, 2) input data rates may be increased by an order of magnitude to determine how input functions will respond, 3) test cases that require maximum memory or other resources are executed, 4) test cases that may cause thrashing in a virtual operating system are designed, 5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

  A variation of stress testing is a technique called sensitivity testing. In some situations, a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.

Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

d. Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the runtime performance of software within the context of an integrated system. This occurs throughout all steps in the testing processes. These are often complied with stress testing to assess hardware & software requirements. This is useful to assess resource utilization and to avoid possible system failure. External instrumentation can monitor execution intervals, log events as they occur and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

4.7. Concept of White-Box and Black-Box Testing

- **White- Box Testing**

White Box testing is also called as glass-box testing. It is a test case design philosophy that uses the control structure described as part of component-level design to drive test cases. Using this method, one can derive test cases that 1) guarantee that all independent paths within a module have been exercised at least once, 2) exercise all logical decisions on their true and false sides, 3) execute all loops at their boundaries and within their operational bounds and 4) exercise internal data structures to ensure their validity.

White-box testing is one of the two biggest testing methodologies used today. It has several major advantages:

1. A side effect of having the knowledge of the source code is beneficial to thorough testing.
2. Optimization of code by revealing hidden errors and being able to remove these possible defects.
3. Gives the programmer introspection because developers carefully describe any new implementation.
4. Provides traceability of tests from the source, allowing future changes to the software to be easily captured in changes to the tests.
5. White box tests are easy to automate.
6. White box testing give clear, engineering-based, rules for when to stop testing.

Although white-box testing has great advantages, it is not perfect and contains some disadvantages:

1. White-box testing brings complexity to testing because the tester must have knowledge of the program, including being a programmer. White-box testing requires a programmer with a high-level of knowledge due to the complexity of the level of testing that needs to be done.
2. On some occasions, it is not realistic to be able to test every single existing condition of the application and some conditions will be untested.
3. The tests focus on the software as it exists, and missing functionality may not be discovered.

- **Black-Box Testing**

  It is called behavior testing or performance testing because it focuses on functional requirement of soft.

  1. Black-Box testing attempts to find errors of following category:
  2. Incorrect or Interface errors
  3. Errors in data structure
  4. Initialization & termination error
  5. Behavior & performance error.

Following are some techniques that can be used for designing black box tests.

Equivalence partitioning

Equivalence Partitioning is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

Boundary Value Analysis

Boundary Value Analysis is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/outside of the boundaries as test data.

Cause Effect Graphing

Cause Effect Graphing is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

BLACK BOX TESTING ADVANTAGES

1. Tests are done from a user's point of view and will help in exposing discrepancies in the specification
2. Tester need not know programming languages or how the software has been implemented
3. Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias
4. Test cases can be designed as soon as the specifications are complete

BLACK BOX TESTING DISADVANTAGES

1. Only a small number of possible inputs can be tested and many program paths will be left untested
2. Without clear specifications, which is the situation in many projects, test cases will be difficult to design
3. Tests can be redundant if the software designer/ developer has already run a test case.

Differentiate between Black Box and White box testing

Criteria	Black Box Testing	White Box Testing
Definition	Black Box testing is a software testing method in which the internal structure/design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/design/ implementation of the item being tested is known to the tester.

Levels Applicable to	Mainly applicable to higher levels of testing: Acceptance testing, System testing	Mainly applicable to lower levels of testing: Unit testing, Integration Testing
Responsibility	Generally independent Software Testers	Generally Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design

4.8. Debugging

- **Concept and need of Debugging**

  Debugging occurs as a consequence of successful testing. When a test uncovers error, debugging is an action that results in the removal of the error. It is a process that connects a symptom to a cause is debugging. Debugging process begins with the exertion of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.

  The debugging process will usually have one of two outcomes 1) the cause will be found and corrected 2) the cause will not be found. In second case the person performing debugging may suspect a cause, design a test case to help validate that suspicion and work toward error correction in an iterative fashion.

- **Characteristics of Bugs**

  1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components.
  2. The symptom may disappear when another error is corrected.
  3. The symptom may actually be caused by nonerrors.
  4. The symptom may be caused by human error that is not easily traced.
  5. The symptom may be a result of timing problems, rather than processing problems.
  6. It may be difficult to accurately reproduce input conditions (e.g a real time application in which input ordering is indeterminate).
  7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
  8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

4.9. Debugging Strategies

Debugging approach has one overriding objective to find the cause to find and correct the cause of a software error or detect. The objective is realized by a combination of systematic evaluation, intuition, and luck.

Debugging tactics

The **brute force** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run time traces are invoked, and the program is loaded with output statements.

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where s symptom has been uncovered, the source code is traced backward until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

Cause Elimination is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

Automated debugging: Each of these debugging approaches can be supplemented with debugging tools that provide semi-automated support for the software engineer as debugging strategies are attempted.

Question Bank

1.	Define Software testing and its role in quality assurance	3 Marks
2.	State six attributes of good testing	3 Marks
3.	Differentiate between good test and successful testing	4 Marks
4.	Define and differentiate between Software verification and validation	4 Marks
5.	Define Unit testing and explain its need	4 Marks
6.	Define Integration Testing – To-down approach	4 Marks
7.	Define Integration Testing – Bottom-up Approach	4 Marks
8.	Briefly explain Regression Testing	4 Marks
9.	Explain Smoke testing and its advantages	4 Marks
10.	Explain and Differentiate Alpha and beta Testing	8 Marks
11.	In System testing explain the need for Recovery testing, Security testing, Stress testing and Performance testing	8 Marks
12.	Explain and differentiate between Black Box and White box testing	8 Marks
13.	Define debugging and list eight characteristics of Bugs	8 Marks
14.	Briefly explain debugging strategies	4 Marks

5. Chapter
Software Project Management

5.1. Introduction to Software Project Management & its need

A Project is a temporary sequence of unique, complex and connected activities that have a goal or purpose and must be completed within budget and specific time and according to specifications.

Management is the art of getting things done through and with a team of individuals in formally organized groups. Project Manager is a person responsible for supervising the project development from start to completion.

The software project management includes basic function such as scoping, planning, estimating, scheduling, organizing, directing, coordinating, controlling and closing. The effective software project management focuses on the four P's viz People, Product, Process and Project.

Project management involves the planning, monitoring, and control of the people, process and events that occur as software revolves from a preliminary concept to an operational implementation.

Effective software project management focuses on four P's People, Product, Process and project.

5.2. The Management Spectrum – the 4 P's and their significance

The People:

The "people factor" is so important that the software engineering institute has developed a people management capability maturity model (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability".

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. The organizations that achieve high levels of PM-CMM have higher likelihood of implementing effective software management.

The software processing is populated by stakeholders who can be categorized into one of five constituencies viz Senior managers, Project (technical) managers, Practitioners, Customers, End-users.

The Product :

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constrains should be identified. Without this information, it is impossible to define reasonable and accurate estimates of the cost ,an effective assignment of risk, a realistic breakdown of project tasks, or a manageable product schedule that provides a meaningful indication of progress.

The software developers and customer must meet to define product objectives and scope. Objectives identify the overall goals for the product (from customer's point of view) without considering how the goal will be achieved. Scope identifies the primary data, function and behaviors that characterize the product, and more importantly, attempt to bind these characteristics in a quantitative manner.

We must examine the product and the problem intended to solve at the very beginning of the project. For this reason, the scope of the product must be established and bounded. The first

software project management activity is the determination of software scope. Software project scope must be unambiguous and understandable at the management and technical levels. Problem Decomposition, sometime called partitioning or problem elaboration is an activity that sits at the core of software require analysis.

Decomposition is applied into two major areas:

1. The functionality that must be delivered and
2. The process that will be used to deliver it.

The Process :

A software process provides the framework from which a comprehensive plan for software development can be established. A number of different tasks sets-tasks, milestones, work products, and quality assurance points enable the framework activities to be adapted to the characteristics of the software projects and the requirements of the project team

Project planning begins with the melding of the product and the process. Assuming that the organization has adopted the following set of framework activities:- communications, planning, modeling, construction, deployment and the team members who work on a product function will apply each of the framework activities to it. The process framework establishes a skeleton for project planning. It is adopted by allocating a task set that is appropriate to the project.

In essence, a matrix as shown below is created. Each major function (of word processing software) is listed in the left-hand column. Framework activities are listed in the top row.

The job of project manager is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a sequence of each task.

Process Decomposition commences when the project manager asks," How do we accomplish this framework activity" For example, a small, relatively simple project or a more complex project, which has a broader scope and more significant business impact requires different work tasks for the communication activity.

The Project:

"A project is like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. But most projects worth doing are more like driving a truck off-road, in the mountains, at night." -Cem Kaner, James Bach, and Pattichord Bret

To avoid project failure, a software project manager and the software engineers who build the project must heed a set of common warning signals, understand the critical success factors that lead to good project management, and develop a common sense approach for planning, monitoring and controlling the project. Reel suggests a five-part common sense approach to software projects:

1. Start on the right foot
2. Maintain momentum
3. Track progress
4. Make smart decisions
5. Conduct a postmortem analysis

5.3. Project Scheduling

- **Concept of Project Scheduling**

  Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product function to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks are identified and scheduled.

- **Factors that delay Project Schedule**

  Scheduling for software engineering can be viewed from two perspectives. In first, an end date for release of a computer-based system has already been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of software.

- **Principles of Project Scheduling**

  Basic Principles guide software project scheduling are:

  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. 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.

  Defined milestones

  Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.Each of these principles is applied as the project schedule evolves.

- **Project Scheduling Techniques – Concept of Gantt Chart, PERT, CPM**

  Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

  Program evaluation and review technique (PERT) and the critical path method (CPM) are two projects scheduling methods that can be applied to software development. Both techniques are driven by project planning activities like estimates effort, a decomposition of the product function, the selection of the appropriate process model and task set and decomposition of the tasks that are selected.

  PERT: project Evaluation and Review techniques have developed in 1950s to plan and control large weapons development projects for the US navy. It was a graphic networking technique. The Microsoft project and other PM software packages PERT chart represent another view of project. It does represent intertask relationships more effectively. Tasks and milestones are included in the chart symbols such as circles; squares are used to depict tasks and milestone. Microsoft uses rectangles to represent task. Each task rectangle is divided into sections with task name at the top and task-id/duration in the middle.

  PERT: Program Evaluation and Review Technique

  PERT plan diagrammatically represents the network of tasks required to complete a project. (Task Network). It explicitly establishes sequential dependencies and relationships among the tasks.

  PERT diagram consists of both activities and events.

  Activity → Time and resource consuming efforts required to complete a segment of the total project. These are represented using solid lines with directional arrays.

  Events → Represent the completion of segments/ part of the project represented by circles.

  Activities and events are coded as described to designate their functions in the overall project.

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

  PERT chart and accompanying table defines estimated and actual times, costs, responsible personnel for monitoring and control of project performances.

The total time required to complete the project can be determine by locating the longest path (in terms of time) in the chart. This path is the "critical path".

Both PERT and CPM provide quantitative tools that allow you to 1) determine the critical path – the chain of tasks that determines the duration of the project, 2) establish "most likely" time estimates for individual tasks by applying statistical models and 3) calculate "boundary times" that define a time "window" for a particular task.

Time-Line Charts

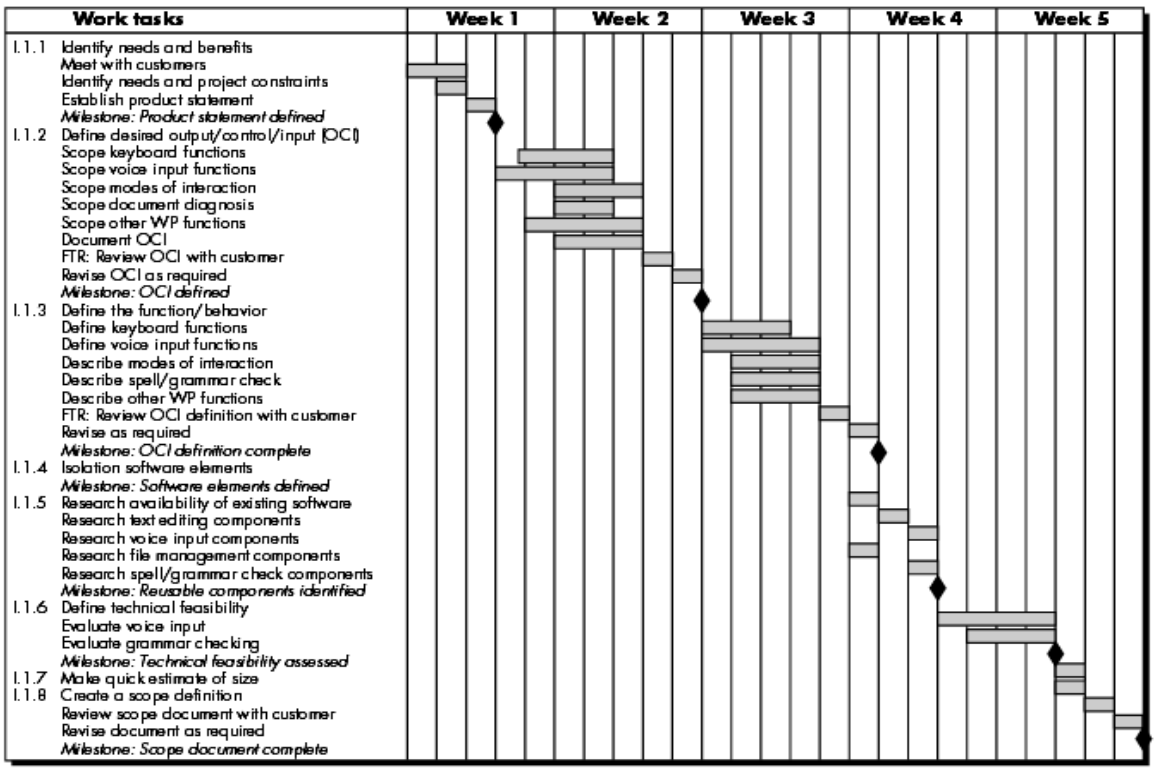
When creating software project schedule, we begin with a set of tasks. If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration and start date are then input for each task, In addition, tasks may be assigned to specific individuals.

As a consequence of this input, a time-line chart, also called a Gantt chart is generated. A time-line chart can be developed for the entire project.

The figure below depicts a part of a software project schedule that emphasizes scoping task for a word-processing (WP) software product. All project tasks are listed in the left-hand 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 diamond indicate milestones.

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce project tables – a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information. Used in conjunction with the time-line chart, project tables enable to track progress.

Time-Line chart - Micro-level Scheduling



Project Table

Critical Path Method (CPM)

A project of any kind involves a number of activities. Some of them are interdependent while others are independent. It is important that project management should effectively plan, schedule, co-ordinate and optimize the activities of the various participants in the project. There are certain activities which are to be completed within the stipulated time. If those critical activities are not completed within the prescribed time line, the completion of the whole project is hampered.

If the project is quite large effective control over all the activities is difficult. To control such projects, Network techniques have been developed.

Advantages of the Critical Path Method:

1. It helps in ascertaining the time schedule.
2. Control becomes easy for management.
3. It helps in preparing a detailed plan of action/operations/ activities
4. It helps in enforcing the plan of actions/operations/activities.
5. It gives a standard method for communicating project plans, schedules and time and cost performances.
6. It identifies the most critical elements.
7. It shows ways to enforce strict supervision over the entire project programme.

5.4. Concept of Task Network

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. The task set must provide enough discipline to achieve high software quality. But, it must not burden the project team with unnecessary work.

In order to develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work.

A task set example

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer believes that potential benefit exists. Concept development projects are approached by applying the following actions:

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.

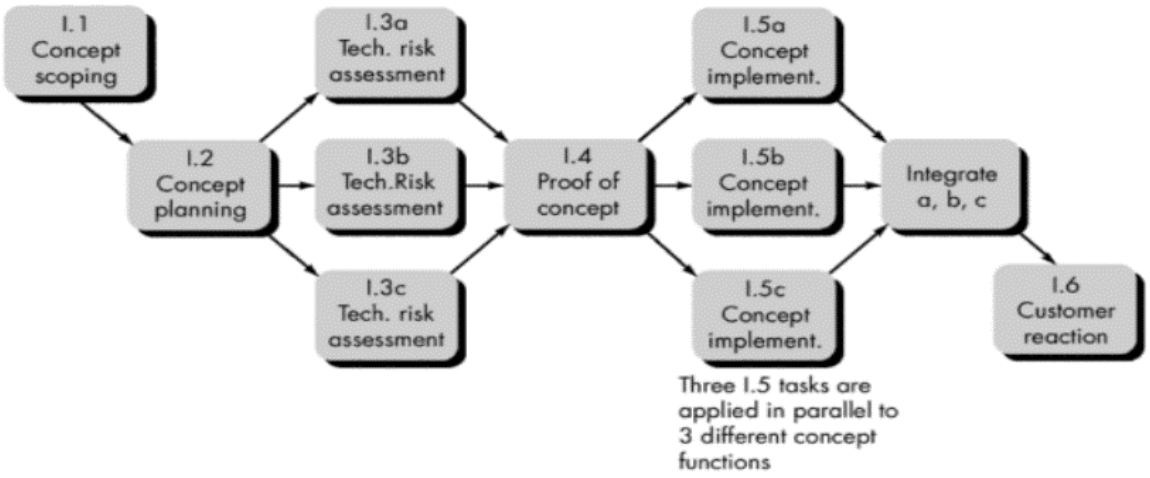
Proof of concept demonstrates the viability of a new technology in the software context.

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.

A task network is also called as 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. In its simplest form, the task network depicts major software engineering actions.

The concurrent nature of software engineering actions leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, it is important to determine dependencies to ensure continuous progress toward completion. In addition, one should be aware of those tasks that lie on the critical path. It means tasks that must be completed on schedule if the project as a whole is to be completed on schedule. It is important to note that the task network is macroscopic.



5.5. Ways of Project Tracking

Project scheduling is very important task. To complete project in decided timing is quite difficult. There might be reality of a technical project that there might be reality of a technical project that there might be hundreds of technical tasks. Some of the tasks may lie in the projects or may be some tasks lie outside the project.

There are certain tasks which fall on the critical path. If those are not considered in schedule, the project may be collapsed. The main job of project manager is to define all tasks involved in project, building a network that shows their independencies and the tasks which are critical within the network.

The major activity carried out in software project scheduling is that, it distributes estimated effects across the planned project period by allocating the effort to particular software engineering tasks.

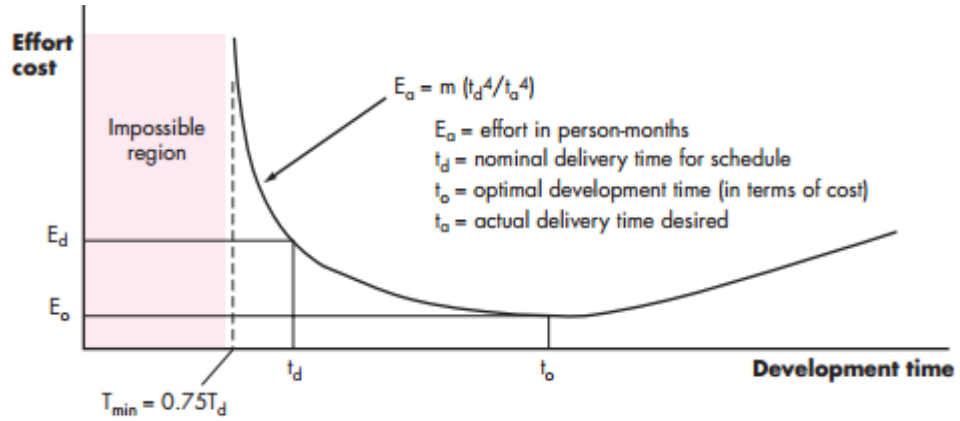
Tracking the schedule

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems
- Evaluating the results of all reviews conducted throughout the software engineering process.

- Comparing the actual start date to the planned start date for each project task listed in the resource table
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon
- Using earned value analysis to assess progress quantitatively

  In reality, all of these tracking techniques are used by experienced project managers.



The relationship between effort and delivery time

The software project scheduling is an activity that distributes effort across the planned project duration by allocating the effort to specific software engineering task.

The project manager prepares a project schedule along with the team members. Every project team member should know their respective tasks and how to fit the same into the overall project schedule.

The project schedule indicates following tasks:

1. The start and stop of each activity
2. When the resource is required?
3. Quantity/Amount of resources.

Generally the schedule evolves overtime. At the early stages of project macroscopic schedule is prepared. After all tasks are identified, the detailed and complete schedule is prepared.

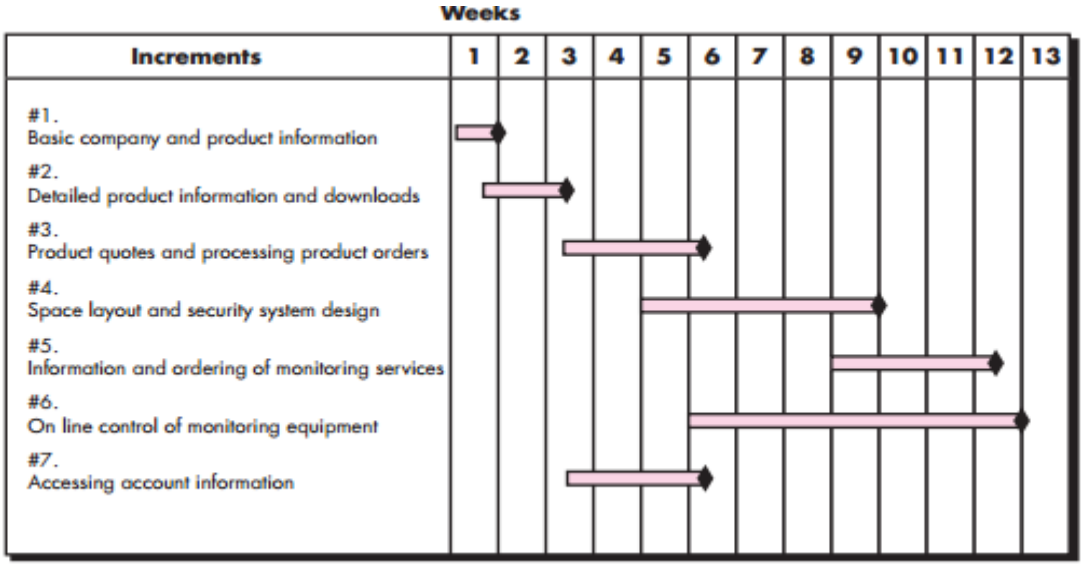
As an example of macroscopic scheduling, consider the SafeHomeAssured.com WebApp. Recalling earlier discussions of SafeHomeAssured.com, seven increments can be identified for the Web-based component of the project:

Increment 1: Basic company and product information

Increment 2: Detailed product information and downloads

Increment 3: Product quotes and processing product orders

Increment 4: Space layout and security system design



Time line for macroscopic project schedule

5.6. Risk Management

What is a software risk?

Risk refers to the uncertainties related to future happenings with the project. A risk is any uncertain event that may or may not happen, which will impact the project.

This means the risks can be predicted and brought within control. However, when the risk becomes a reality, it leads to unwanted consequences and may be losses.

The risk may be proactive or reactive

Reactive Risk Strategy

The reactive risk strategies monitor the project for likely risks. It can be called as the fire-fighting mode or the crisis management mode.

The team gets into action in an attempt to correct the problem rapidly

Proactive Risk Strategy

A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, priorities are ranked and a plan is established to avoid such risks. This is called as a contingency plan which will enable to respond in a controlled and effective manner.

Project Risk

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Technical Risks

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes 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 Risk

Business risks threaten the viability of the software to be built and often risk the project or the product. 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).

5.7. Risk Assessment

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories: generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product- specific risks can be identified only by those with a clear understanding of technology, the people, and the environment that is specific to the software that is to build.

One method for identifying risks is to create a risk item checklist. The checklist can be used for identification and focuses subset or known and predictable risks in the following generic subcategories.

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 built 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.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answer to these questions allows you to estimate the impact of risk.

Risk Analysis

The following questions are to be used for analyzing project risk:

Have top software and customer manager formally committed to support the project?

Are end users enthusiastically committed to the project and the system/product to be built?

Are requirements fully understood by the software engineering team and its customers?

Have customers been involved fully in the definition of requirements?

Do end users have realistic expectations?

Is the project scope stable?

Does the software engineering team have the right mix of skills?

Are project requirements stable?

Does the project team have experience with the technology to be implemented?

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

Do all customer/user constitutes agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively mitigation. Monitoring and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

Risk Components

The 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.

Impact Assessment

Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Risk Projection/Prioritization

Risk projection also called risk estimation, attempts to rate each risk in two ways

1. The likelihood or probability that the risk is real and
2. The consequences of the problems associated with the risk

There are four risk projection steps:

1. Establish a scale that reflects the perceived likelihood of risk
2. Delineate the consequence 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 there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, we can allocate resources where they will have the most impact.

A risk table provides with a simple technique for risk projection.

Sample Risk table prior to sorting

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				

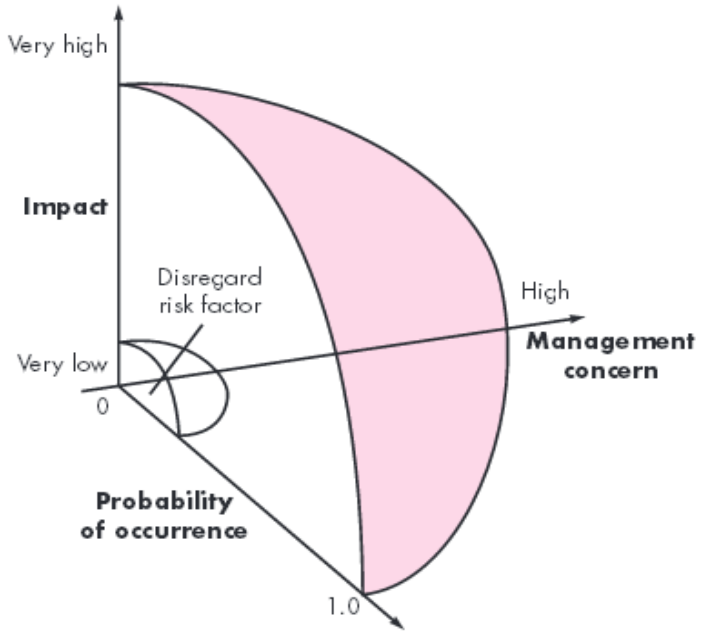
Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

It contains all risks listed in first column of the table. Each risk is categorized in the second column. The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. The impact of each risk is assessed. The categories for each of the four risk components – performance, support, cost and schedule – are averaged to determine an overall impact value.

After the four columns are completed, the table is sorted by probability and by impact. High High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

The cutoff line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are reevaluated to accomplish second-order prioritization.

Risk and Management Concern



risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cutoff line should be managed. The column labeled RMMM contains a pointer into a risk mitigation, monitoring, and management plan or, alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed. Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs.

The overall risk exposure RE is determined using the following relationship

$$RE = P \times C$$

Where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

For example: assume that the software team defines a project risk in the following manner:

Risk identification: Only 70 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: 80 percent (likely).

Risk impact: Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC

and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be

18 X 100 X 14 = $25,200.

Risk exposure: RE = 0.80 X 25,200 ~ $20,200

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

5.8. Risk Control – Need and RMMM Strategy

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, 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

Using the CTC format for the reuse risk, we can write

Given that 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, resulting in the need to custom engineer the remaining 30 percent of components.

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 subconditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

RMMM Strategy:

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

f 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.

For example,

Assume that high staff turnover is noted as a project risk r_1 . Based on past history and management intuition, the likelihood l_1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x_1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule

- To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:
- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts. Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, risk-monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored. In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, one can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents or Wikis," and/or meeting with other team members who will remain on the project.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, you perform a classic cost-benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate risk mitigation, monitoring, and management plan (RMMM). The RMMM plan documents all work performed as part of risk analysis and are used by the project manager as part of the overall project plan.

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate risk mitigation, monitoring, and management plan (RMMM). The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan. Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet (RIS). In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As I have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk.

5.9. Software Configuration Management (SCM)

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms), (2) work products that describe the computer programs (targeted at various stakeholders), and (3) data or content (contained within the program or external to it). 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.

An SCM Scenario

A typical CM operational scenario involves a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product.

At the operational level, the scenario involves various roles and tasks. For the project manager, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

The goals of the configuration manager are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating them (via a Change Control Board that is responsible for approving changes to the software system), and for authorizing changes. The manager creates and disseminates task lists for the engineers and basically creates the project context. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

For the software engineers, the goal is to work effectively. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting work products. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying one another about tasks required and tasks completed. Changes are propagated across each other's work by merging files. Mechanisms exist to ensure that, for components that undergo simultaneous changes, there is some way of resolving conflicts and merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed.

The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

Benefits of SCM

SCM is an umbrella activity that is applied throughout the software process.

SCM is a set of tracking and control activities that are initiated when SE project begin and terminate only when the software is taken out of operation.

SCM helps to improve software quality and on time delivery.

SCM defines the project strategy for change management. When formal SCM is invoked, the change control process produces software change requests, reports and engineering change orders.

SCM helps to track, analyze and control every work product.

Elements of Configuration Management System

- Component elements — a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- Process elements — a collection of actions and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- Construction elements — a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- Human elements — a set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM

SCM Repository - Functions and Features supported

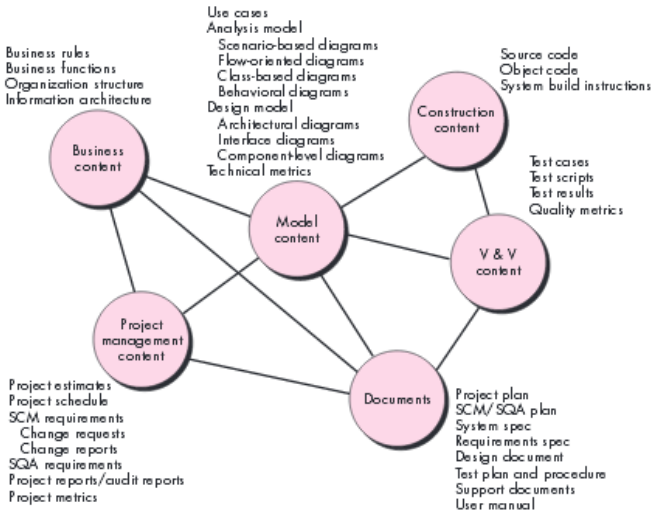
The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM

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. To achieve these capabilities, the repository is defined in terms of a meta-model. The meta-model determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs

General features and Content of repository

The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. A robust 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)

Content of Repository



SCM Features

Versioning.

As a project progresses, many versions (Section 22.3.2) 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 to permit developers to go back to previous versions during testing and debugging. The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects like screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

Dependency tracking and change management.

The repository manages a wide variety of relationships among the data elements stored in it. These 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 and to the generation of deliverables based on it, and it is one of the most important contributions of the repository concept to the improvement of the software process. For example, if a UML class diagram is modified, the repository can detect whether related classes, interface descriptions, and code components also require modification and can bring affected SCIs to the developer's attention.

Requirements tracing.

This special function depends on link management and provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, 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. Information about the source of changes can be entered as attributes of specific objects in the repository. A repository trigger mechanism is helpful for prompting the developer or the tool that is being used to initiate entry of audit information (such as the reason for a change) whenever a design element is modified.

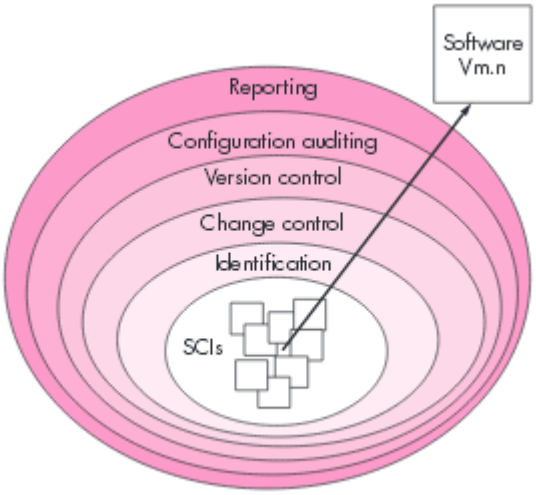
The SCM Process – Change Control and Version Control

The software configuration management process defines a series of tasks that have 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, and (4) to ensure that software quality is maintained as the configuration evolves over time. A process that achieves these objectives need not be bureaucratic or ponderous, but it must be characterized in a manner that enables a software team to develop answers to a set of complex questions:

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How an organization control changes does before and after software is released to a customer?
- Who has responsibility for approving and ranking requested changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead to the definition of five SCM tasks—identification, version control, change control, configuration auditing, and reporting

Layers of SCM process



SCM tasks can viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system. As an SCI moves through a layer, the actions implied by each SCM task may or may not be applicable. For example, when a new SCI is created, it must be identified. However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play). A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know.

Identification of Object 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 is created 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 the code.

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 four major capabilities: (1) a project database (repository) that stores all relevant configuration objects, (2) a version management capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions), (3) a make facility that enables you to collect all relevant configuration objects and construct a specific version of the software. In addition, version control and change control systems often implement an issues tracking (also called bug tracking) capability that 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 change set —a collection of all changes (to some baseline configuration) that are required to create a specific version of the software.

A number of named change sets can be identified for an application or system. This enables you to construct a version of the software by specifying the change sets (by name) that must be

applied to the baseline configuration. To accomplish this, a system modeling approach is applied. The system model contains: (1) a template that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed, (2) construction rules, and (3) verification rules. (4) A number of different automated approaches to version control have been proposed over the last few decades. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

Change Control

Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. 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 on the status and priority of the change. 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, and the criteria for review and audit.

The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system updates the original file once the change has been made. As an alternative the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied. The object(s) is (are) then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.

These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control. Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

The Change Control Process

Question Bank

1. Define Project Management, Manager, Software Project Management, Software Configuration Management (SCM) 4 Marks
2. Explain in brief 4 P's of Software Project Management 4 Marks
3. State eight causes/reasons for the failure of software projects 4 Marks
4. State basic principles of project scheduling. 4 Marks
5. Define Software risk, Project risk, Technical risk, Business risk. 4 Marks
6. Differentiate between Reactive and Proactive risk strategy. 4 Marks
7. Describe relationship between effort and delivery time in scheduling. 4 Marks
8. Using a schematic diagram describe concept of Task Network 4 Marks
9. What is software project tracking and control? Enlist activities. 4 Marks
10. What is risk projection? Explain steps. 4 Marks
11. Explain PERT with an example. 4 Marks
12. Explain Timeline (Gantt) chart. 4 Marks
13. Explain CPM with an example. 4 Marks
14. State recommended guidelines to avoid schedule failure. 4 Marks

15. Explain software Configuration Management (SCM). 4 Marks
16. State and define elements of Configuration Management System. 4 Marks
17. Describe SCM repository. 4 Marks
18. Explain SCM Process change Control and Version Control. 4 Marks
19. State benefits of SCM. 4 Marks

6. Chapter
Software Quality Management

6.1. Basic Quality Concept

- Quality: - Fit for use & meeting customer's requirements
- Process: - A sequence of steps performed for a given purpose
- Quality assurance: All those planned and systematic actions necessary to provide adequate confidence that a product or service will satisfy given requirements for quality.
- Quality control: Includes review & testing
- Quality Management
- Quality: A characteristic attribute of something. As an attribute of an item it refers to measurable characteristics such as length, colures, electrical properties etc. Software is largely an intellectual entity, which is more challenging to characteristics than physical objects.
- User satisfaction: compliant budget, good quality, delivery within scheduled.
- Variation control: inspection, review, and tests.
- Quality assurance: Audit, reporting, effectiveness and completeness of the activities.
- Cost of quality : Prevention Costs, Appraisal Costs Failure Costs

6.2. Software Quality Assurance

Conformance to explicit stated functional and performance requirements, explicitly documented development standards and implicit characteristic that are expected of all professionally developed software.

1. Software requirements are the bases from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If criteria is not followed, lack of quality will be the result.
3. Set of implicit requirements i.e. desire of ease of use, good maintainability. If software conforms to explicit and fails to mean implicit requirements, quality is a suspect.

- **SQA activities**

  Software engineers do the technical works.
  This is an independent group:

  1. SQA plans for project.
  2. Participants in the development of the project's software process description.
  3. Reviews software engineering activities to verify compliances with the defined software process.
  4. Audits designated software work products to verify compliances to define as part of software process.
  5. Ensure deviations in software work are documented; products are documented and handled according to a documented procedure.
  6. Record any non-compliances and report to senior management.

- **Software Reviews**

  It is a filter for software process. A formal technical review is essentials from QA point of views for unnecessary errors and improving software quality.

Cost impact in software defect: - FTR for avoiding errors after software release. A Cost impacts are minimum and errors are deleted easily.

Defect implications examiner: Generations and detector errors in preliminary design stage.

1. **Review**
   - 3-5 people involved
   - Advance preparations
   - Medium durations
   - What is reviewed?
   - Who reviewed?
   - Finding & conclusion

2. **FTR Guidelines**
   - Reviews the product
   - Set agenda & maintain
   - Limits debate & rebuttal
   - Enunciate problem errors
   - Text written notes
   - Limit number of participants and insist on advance preparation.
   - Develop a checklist for each product that is likely to be reviewed.
   - Allocate resources & scheduled time
   - Conduct meaningful training for all reviews
   - Review your early reviews

6.3. Concept of Statistical SQA

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the vital few).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

This relatively simple concept represents an important step toward the creation of an adaptive software process in which changes are made to improve those elements of the process that introduce error.

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 hundreds of different problems are uncovered, all can be tracked to one (or more) of 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)

To apply statistical SQA, the table given below is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, you might implement requirements gathering techniques to improve the quality of customer communication and specifications. To improve EDR, you might acquire tools for data modeling and perform more stringent data design reviews. It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack. Statistical quality assurance techniques for software have been shown to provide substantial quality improvement. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques. The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: Spend your time focusing on things that really matter, but first be sure that you understand what really matters!

Data collection for statistical SQA

	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%

6.4. Quality Evaluation Standards

- **Six Sigma for Software**

  Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

  - **D**efine customer requirements and deliverables and project goals via well-defined methods of customer communication.
  - **M**easure the existing process and its output to determine current quality performance (collect defect metrics).
  - **A**nalyze 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:

  - **I**mprove the process by eliminating the root causes of defects.
  - **C**ontrol 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 (define, measure, analyze, improve, and control**) method.

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

  - **D**efine customer requirements and deliverables and project goals via well-defined methods of customer communication.
  - **M**easure the existing process and its output to determine current quality performance (collect defect metrics).
  - **A**nalyze defect metrics and determine the vital few causes.
  - **D**esign the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
  - **V**erify that the process model will, in fact, avoid defects and meet customer requirements.

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

- **ISO 9000 for Software – Concept and major considerations**

  ISO Standards

  ISO (International Organization for Standardization) is the world's largest developer of standards.

  ISO is a network of the national standards institutes of 148 countries, on the basis of one member per country, with a central secretariat in Geneva, Switzerland, that coordinates the system.

  ISO is a non-government organization: its members are not, as is the case in the United Nations system, delegations of national governments

ISO is able to act, as a bridging organization in which a consensus can be reached on solutions that meet both the requirements of business and the broader needs of society

ISO Standards benefits to society:

For customers, the worldwide compatibility of technology

For governments, International Standards provide the technological and scientific bases underpinning health, safety and environmental legislation.

For trade officials negotiating the emergence of regional and global markets, Internationals Standards create "a level playing field" for all competitors on those markets.

For developing countries, International Standards that represent an international an international consensus on the state of the art constitute an important source of technological know-how

For consumers, conformity of products and services to international Standards provides assurance about their quality, safety and reliability.

For everyone, International Standards can contribute to the quality of life in general by ensuring that the transport, machinery and tools we use are safe.

For the planet it inhabits, International Standards on air, water and soil quality, and on emission of gasses and radiation, can contribute to efforts to preserve the environment.

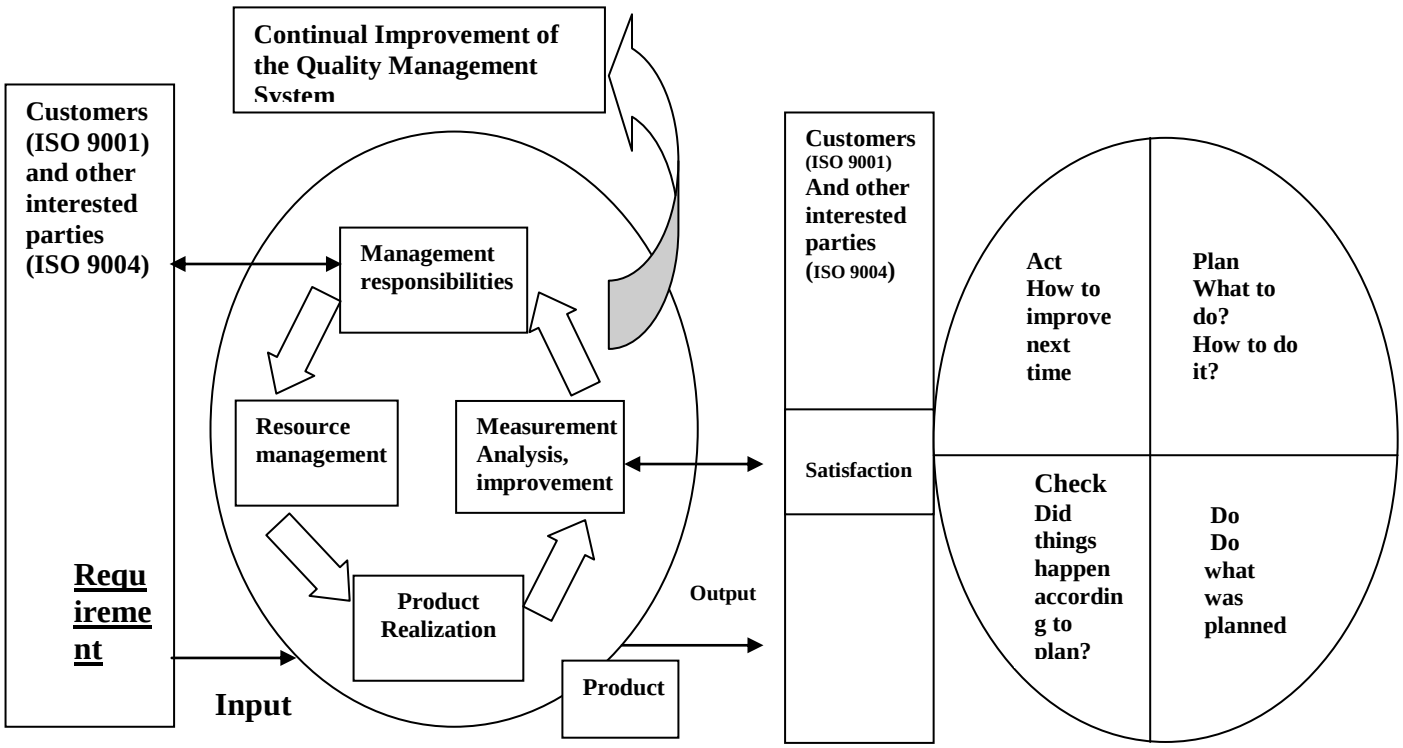
ISO 9000 Standards

ISO 9000: 2000 – Guidelines for selection & use

ISO 9001: 2000 – Quality Assurance Model

ISO 9004: 2000 – Guidelines for process improvements

Model of a process-based Quality Management System



6.5. CMMI – CMMI levels, Process Area Considered

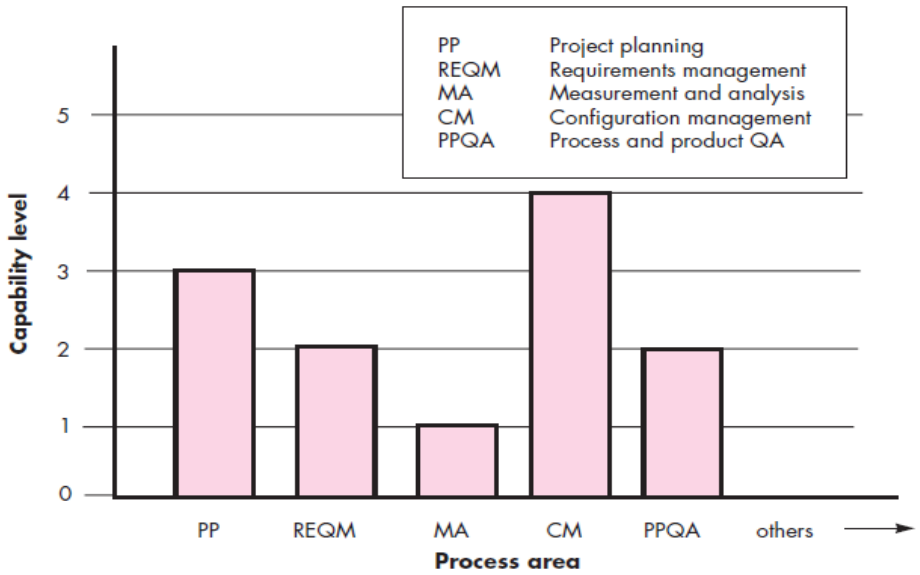
The Capability Maturity Model Integration (CMMI), a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI metamodel describes a process in two dimensions as illustrated in Figure 30.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

Level 0: Incomplete—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

Level 1: Performed—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

CMMI Process Area Capability Profile.



Level 2: Managed—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description".

Level 3: Defined—all capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets".

Level 4: Quantitatively managed—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process".

Level 5: Optimized—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. Specific goals establish the characteristics that must exist if the activities implied by a process area are to be effective. Specific practices refine a goal into a set of process-related activities.

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity level and Process level is shown in the diagram below:

Process Areas required to achieve a Maturity level.

Level	Focus	Process Areas
Optimizing	*Continuous process improvement*	Organizational innovation and deployment Causal analysis and resolution
Quantitatively managed	*Quantitative management*	Organizational process performance Quantitative project management
Defined	*Process standardization*	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Integrated supplier management Risk management Decision analysis and resolution Organizational environment for integration Integrated teaming
Managed	*Basic project management*	Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management
Performed		

6.6. CMMI Vs ISO

Although the SEI's CMM and CMMI are the most widely applied SPI frameworks, a number of alternatives7 have been proposed and are in use. Among the most widely used of these alternatives are:

- SPICE—an international initiative to support ISO's process assessment and life cycle process standards [SPI99].
- ISO/IEC 15504 for (Software) Process Assessment [ISO08].
- Bootstrap—an SPI framework for small and medium-sized organizations that conforms to SPICE [Boo06].
- PSP and TSP—individual and team-specific SPI frameworks ([Hum97], [Hum00]) that focus on process in-the-small, a more rigorous approach to software development coupled with measurement.
- TickIT—an auditing method [Tic05] that assesses an organization's compliance to ISO Standard 9001:2000.

6.7. MsCall's Quality Factors

McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments. McCall and his colleagues provide the following descriptions:

Correctness: The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

Reliability: The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed.

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 learning, operating, preparing input for, and interpreting output of a program.

Maintainability: Effort required locating and fixing an error in a program. [This is a very limited definition.]

Flexibility: Effort required modifying an operational program.

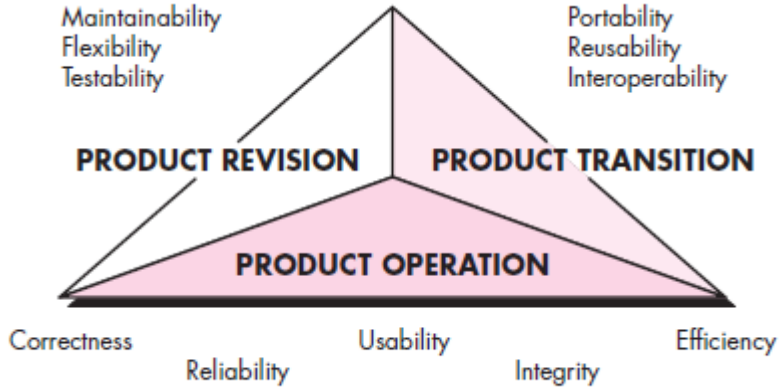
Testability: Effort required testing a program to ensure that it performs its intended function.

Portability: Effort required transferring 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

McCall's software quality factors



It is difficult, and in some cases impossible, to develop direct measures of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality.

Question Bank

1.	Define SQA and list three quality aspects.	4 Marks
2.	Explain Six Sigma. DMADV and DMAIC w.r.t Six Sigma	4 Marks
3.	Define software reliability and software availability.	4 Marks
4.	Explain Eight McCalls quality factors.	4 Marks
5.	Define quality (QC), Quality Assurance (QA)	4 Marks
6.	Write in brief on ISO 9000 software quality standards	4 Marks
7.	Write a detail note on Software Quality Assurance	4 Marks
8.	Write a note on Statistical SQA.	4 Marks
9.	Explain CMMI with its levels of Integration	4 Marks
10.	Compare CMMI and ISO w.r.t Scope, approach and implementation	4 Marks
11.	List quality evaluation standards. Explain any one in detail.	4 Marks