LECTURE NOTES

ON

SOFTWARE ENGINEERING

III B-Tech I Semester

INFORMATION TECHNOLOGY

CMR TECHNICAL CAMPUS

KANDLAKOYA (V), MEDCHAL

UNIT-I

INTRODUCTION TO SOFTWARE ENGINEERING

Software: Software is

Instructions (computer programs) that provide desired features, function, and performance, when executed
Data structures that enable the programs to adequately manipulate information,
Documents that describe the operation and use of the programs.

Characteristics of Software:

Software is developed or engineered; it is not manufactured in the classical sense.
Software does not "wear out"
Although the industry is moving toward component-based construction, most software continues to be custom built.

Software Engineering:

The systematic, disciplined quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
The study of approaches as in (1)

EVOLVING ROLE OF SOFTWARE:

Software takes 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 by a network of computers.

As a **vehicle**: It is information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as single bit or as complex as a multimedia presentation.

Software delivers the most important product of our time-information.

It transforms personal data
It manages business information to enhance competitiveness
It provides a gateway to worldwide information networks
It provides the means for acquiring information

The role of computer software has undergone significant change over a span of little more than 50 years

Dramatic Improvements in hardware performance
Vast increases in memory and storage capacity
A wide variety of exotic input and output options

1970s and 1980s:

Osborne characterized a "new industrial revolution"
Toffler called the advent of microelectronics part of "the third wave of change" in human history
Naisbitt predicted the transformation from an industrial society to an "information society"
Feigenbaum and McCorduck suggested that information and knowledge would be the focal point for power in the twenty-first century

Stoll argued that the "electronic community" created by networks and software was the key to knowledge interchange throughout the world

1990s began:

Toffier described a "power shift" in which old power structures disintegrate as computers and software lead to a "democratization of knowledge".
Yourdon worried that U.S companies might lose their competitive edge in software related business and predicted "the decline and fall of the American programmer".
Hammer and Champy argued that information technologies were to play a pivotal role in the "reengineering of the corporation".

Mid-1990s:

The pervasiveness of computers and software spawned a rash of books by neo-luddites.

Later 1990s:

Yourdon reevaluated the prospects of the software professional and suggested "the rise and resurrection" of the American programmer.
The impact of the Y2K "time bomb" was at the end of 20th century

2000s progressed:

Johnson discussed the power of "emergence" a phenomenon that explains what happens when interconnections among relatively simple entities result in a system that "self-organizes to form more intelligent, more adaptive behavior".
Yourdon revisited the tragic events of 9/11 to discuss the continuing impact of global terrorism on the IT community
Wolfram presented a treatise on a "new kind of science" that posits a unifying theory based primarily on sophisticated software simulations
Daconta and his colleagues discussed the evolution of "the semantic web".

Today a huge software industry has become a dominant factor in the economies of the industrialized world.

THE CHANGING NATURE OF SOFTWARE:

The 7 broad categories of computer software present continuing challenges for software engineers:

System software
Application software
Engineering/scientific software
Embedded software
Product-line software
Web-applications
Artificial intelligence software.

System software: System software is a collection of programs written to service other programs. The systems software 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
- multiple external interfaces

E.g. compilers, editors and file management utilities.

Application software:

Application software consists of standalone programs that solve a specific business need.
It facilitates business operations or management/technical decision making.
It is used to control business functions in real-time

E.g. point-of-sale transaction processing, real-time manufacturing process control.

Engineering/Scientific software: Engineering and scientific applications range
from astronomy to volcanology
from automotive stress analysis to space shuttle orbital dynamics
from molecular biology to automated manufacturing
E.g. computer aided design, system simulation and other interactive applications.

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.
It can perform limited and esoteric functions or provide significant function and control capability.
E.g. Digital functions in automobile, dashboard displays, braking systems etc.

Product-line software: Designed to provide a specific capability for use by many different customers, product-line software can focus on a limited and esoteric market place or address mass consumer markets
E.g. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications

Web-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.

Artificial intelligence software: AI software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Application within this area includes robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

The following are the **new challenges** on the horizon:

Ubiquitous computing
Netsourcing
Open source
The "new economy"

Ubiquitous computing: The **challenge** for software engineers will be to develop systems and application software that will allow small devices, personal computers and enterprise system to communicate across vast networks.

Net sourcing: The **challenge** for software engineers is to architect simple and sophisticated applications that provide benefit to targeted end-user market worldwide.

Open Source: The **challenge** for software engineers is to build source 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.

The "new economy": The **challenge** for software engineers is to build applications that will facilitate mass communication and mass product distribution.

SOFTWARE MYTHS

Beliefs about software and the process used to build it- can be traced to the earliest days of computing myths have a number of attributes that have made them insidious.

Management myths: Manages with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myth: We already have a book that's full of standards and procedures for building software - Wont that provide my people with everything they need to know?

Reality: The book of standards may very well exist but, is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice?

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

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spend on productive development effort. People can be added but only in a planned and well coordinated manner.

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

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

Customer myths: The customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and ultimately, dissatisfaction with the developer.

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

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is recipe for disaster.

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

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths: Myths that are still believed by software practitioners: during the early days of software, programming was viewed as an art from old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our jobs are done.

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

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

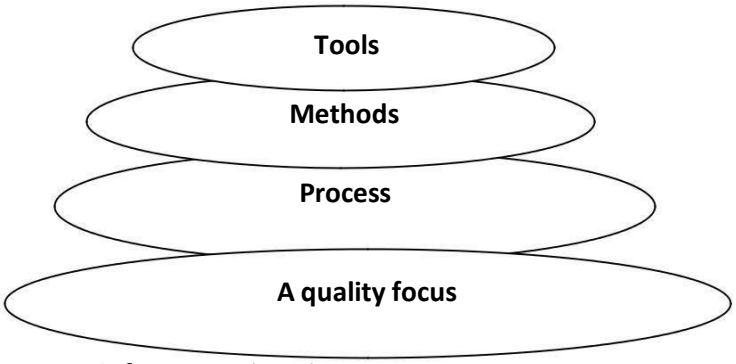
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides guidance for software support.

Myth: software engineering will make us create voluminous and unnecessary documentation and will invariably slows down.

Reality: software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

A GENERIC VIEW OF PROCESS

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:



Software Engineering Layers

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. **The bedrock that supports software engineering is a quality focus.**

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers. **Process defines a framework that must be established for effective delivery of software engineering technology**.

The software forms the basis for management control of software projects and establishes the context in which

technical methods are applied,

work products are produced,

milestones are established,

quality is ensured,

And change is properly managed.

Software engineering methods rely on a set of basic principles that govern area of the technology and include modeling activities.

Methods encompass a broad array of tasks that include
communication,

requirements analysis,

design modeling,

program construction,

Testing and support.

Software engineering tools provide automated or 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.

A PROCESS FRAMEWORK:

Software process must be established for effective delivery of software engineering technology.

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.

The process framework encompasses a **set of umbrella activities** that are applicable across the entire software process.

Each **framework activity** is populated by a set of software engineering actions

Each **software engineering action** is represented by a number of different task sets- each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

In brief

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

A **Process Framework**

establishes the foundation for a complete software process

identifies a small number of **framework activities**

applies to all s/w projects, regardless of size/complexity.

also, set of **umbrella activities**

applicable across entire s/w process.

Each **framework activity** has

set of **s/w engineering actions**.

Each **s/w engineering action** (e.g., design) has

collection of related **tasks (**called **task sets):**

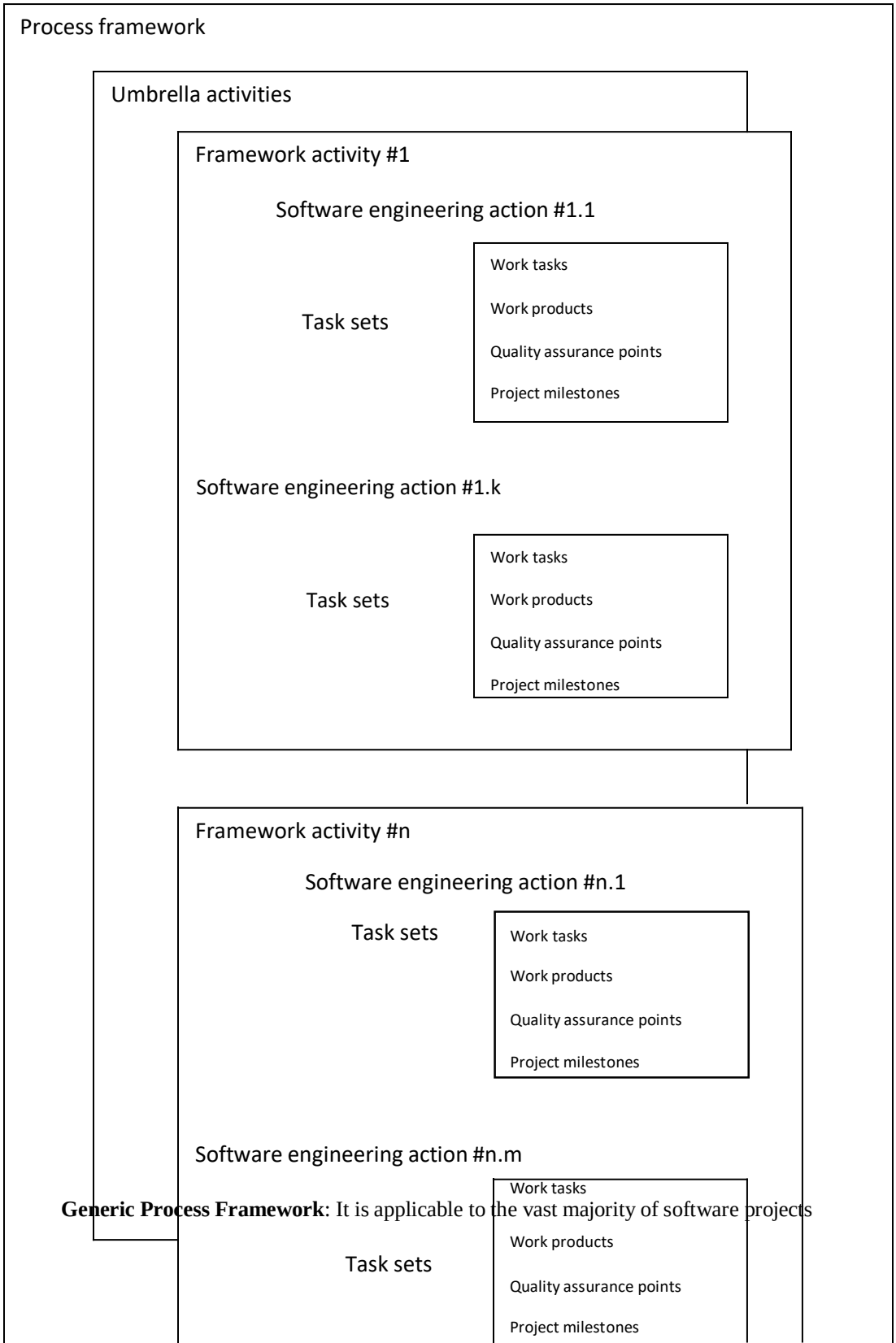
work tasks

work products (deliverables)

quality assurance points

project milestones.

Software process



Generic Process Framework: It is applicable to the vast majority of software projects

Communication activity
Planning activity
Modeling activity
    analysis action
        requirements gathering work task
        elaboration work task
        negotiation work task
        specification work task
        validation work task
    design action
        data design work task
        architectural design work task
        interface design work task
        component-level design work task
Construction activity
Deployment activity

Communication: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

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.

Modeling: This activity encompasses the creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements. The modeling activity is composed of 2 software engineering actions- analysis and design.

Analysis encompasses a set of work tasks.

Design encompasses work tasks that create a design model.

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

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

These 5 generic framework activities can be used during the development of small programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.

The following are the set of **Umbrella Activities.**

Software project tracking and control – allows the software team to assess progress against the project plan and take necessary action to maintain schedule.

Risk Management - assesses risks that may effect the outcome of the project or the quality of the product.

Software Quality Assurance - defines and conducts the activities required to ensure software quality.

Formal Technical Reviews - assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.

Measurement - define and collects process, project and product measures that assist the team in delivering software that needs customer's needs, can be used in conjunction with all other framework and umbrella activities.

Software configuration management - manages the effects of change throughout the software process.

Reusability management - defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

Work Product preparation and production - encompasses the activities required to create work products such as models, document, logs, forms and lists.

Intelligent application of any software process model must recognize that adaption is essential for success but process models do differ fundamentally in:

The overall flow of activities and tasks and the interdependencies among activities and tasks.

The degree through which work tasks are defined within each frame work activity.

The degree through which work products are identified and required.

The manner which quality assurance activities are applied.

The manner in which project tracking and control activities are applied.

The overall degree of the detailed and rigor with which the process is described.

The degree through which the customer and other stakeholders are involved with the project.

The level of autonomy given to the software project team.

The degree to which team organization and roles are prescribed.

THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):

The CMMI represents a process meta-model in two different ways:

As a continuous model

As a staged model.

Each process area is formally assessed against specific goals and practices and is rated according to the following capability levels.

Level 0: Incomplete. The process area is either not performed or does not achieve all goals and objectives defined by CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All 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;

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

Level 4: Quantitatively managed. All 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 level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative 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 practices refine a goal into a set of process-related activities.

The specific goals (SG) and the associated specific practices(SP) defined for project planning are

SG 1 Establish estimates

SP 1.1 Estimate the scope of the project

SP 1.2 Establish estimates of work product and task attributes

SP 1.3 Define project life cycle

SP 1.4 Determine estimates of effort and cost

SG 2 Develop a Project Plan

SP 2.1 Establish the budget and schedule

SP 2.2 Identify project risks

SP 2.3 Plan for data management

SP 2.4 Plan for needed knowledge and skills

SP 2.5 Plan stakeholder involvement

SP 2.6 Establish the project plan

SG 3 Obtain commitment to the plan

SP 3.1 Review plans that affect the project

SP 3.2 Reconcile work and resource levels

SP 3.3 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, **the generic goals (GG) and practices (GP)** for the project planning process area are

GG 1 Achieve specific goals

GP 1.1 Perform base practices

GG 2 Institutionalize a managed process

GP 2.1 Establish and organizational policy

GP 2.2 Plan the process

GP 2.3 Provide resources

GP 2.4 Assign responsibility

GP 2.5 Train people

GP 2.6 Manage configurations

GP 2.7 Identify and involve relevant stakeholders

GP 2.8 Monitor and control the process

GP 2.9 Objectively evaluate adherence

GP 2.10 Review status with higher level management

3 Institutionalize a defined process

GP 3.1 Establish a defined process

GP 3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

GP 4.1 Establish quantitative objectives for the process

GP 4.2 Stabilize sub process performance

GG 5 Institutionalize and optimizing process

GP 5.1 Ensure continuous process improvement

GP 5.2 Correct root causes of problems

PROCESS PATTERNS

The software process can be defined as a collection patterns that define a set of activities, actions, work tasks, work products and/or related behaviors required to develop computer software.

A process pattern provides us with a template- a consistent method for describing an important characteristic of the software process. A pattern might be used to describe a complete process and a task within a framework activity.

Pattern Name: The pattern is given a meaningful name that describes its function within the software process.

Intent: The objective of the pattern is described briefly.

Type: The pattern type is specified. There are three types

> **Task patterns** define a software engineering action or work task that is part of the process and relevant to successful software engineering practice. *Example:* Requirement Gathering
>
> **Stage Patterns** define a framework activity for the process. This pattern incorporates multiple task patterns that are relevant to the stage.
>
> *Example:* Communication
>
> **Phase patterns** define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.
>
> *Example:* Spiral model or prototyping.

Initial Context: The conditions under which the pattern applies are described prior to the initiation of the pattern, we ask

> What organizational or team related activities have already occurred.
>
> What is the entry state for the process
>
> What software engineering information or project information already exists

Problem: The problem to be solved by the pattern is described.

Solution: The implementation of the pattern is described.

This section describes how the initial state of the process is modified as a consequence the initiation of the pattern.

It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern

Resulting Context: The conditions that will result once the pattern has been successfully implemented are described. Upon completion of the pattern we ask

> What organizational or team-related activities must have occurred
>
> What is the exit state for the process
>
> What software engineering information or project information has been developed?

Known Uses: The specific instances in which the pattern is applicable are indicated

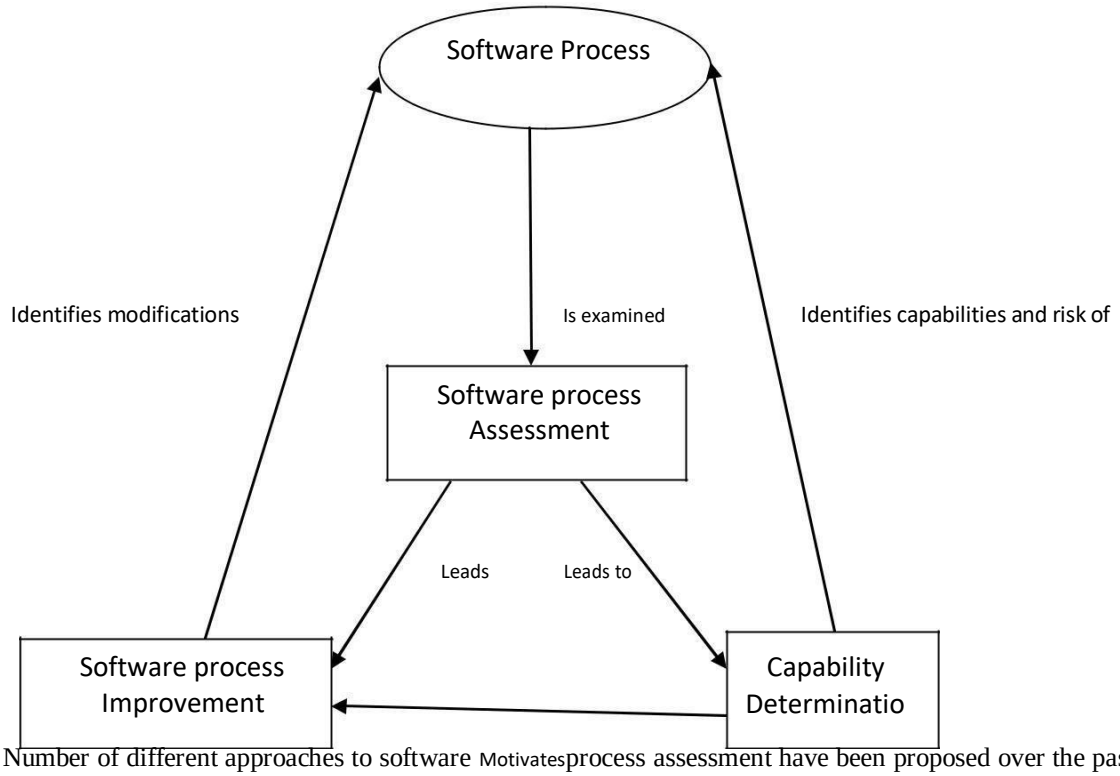
Process patterns provide and effective mechanism for describing any software process.

The patterns enable a software engineering organization to develop a hierarchical process description that begins at a high-level of abstraction.

Once process pattern have been developed, they can be reused for the definition of process variants-that is, a customized process model can be defined by a software team using the pattern as building blocks for the process models.

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. In addition, the process itself should be assessed to be essential to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.



A Number of different approaches to software process assessment have been proposed over the past few decades.

Standards CMMI Assessment Method for Process Improvement (SCAMPI) provides a five step process assessment model that incorporates initiating, diagnosing, establishing, acting & learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM Based Appraisal for Internal Process Improvement (CBA IPI) provides a diagnostic technique for assessing the relative maturity of a software organization, using the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504) standard defines a set of requirements for software process assessments. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

ISO 9001:2000 for Software is a generic standard that applies to any organization that wants to improve the overall quality of the products, system, or services that it provides. Therefore, the standard is directly applicable to software organizations &companies.

PERSONAL AND TEAM PROCESS MODELS:

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

Each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization. Alternatively, the team itself would create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.

Personal software process (PSP)

The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.

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

Planning: This activity isolates requirements and, base on these develops both size and resource estimates. In addition, a defect estimate is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High level design: External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High level design review: Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development: The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important task and work results.

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

PSP stresses the need for each software engineer to identify errors early and, as important, to understand the types of errors that he is likely to make.

PSP represents a disciplined, metrics-based approach to software engineering.

Team software process (TSP): The goal of TSP is to build a "self-directed project team that organizes itself to produce high-quality software. The following are the 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 and 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 defines

- roles and responsibilities for each team member -

tracks quantitative project data

- identifies a team process that is appropriate for the project - a

strategy for implementing the process

- defines local standards that are applicable to the teams software engineering work; -

continually assesses risk and reacts to it

-Tracks, manages, and reports project status.

TSP defines the following framework activities: launch, high-level design, implementation, integration and test, and postmortem.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.

Scripts define specific process activities and other more detailed work functions that are part of the team process.

Each project is "launched" using a sequence of tasks.

The following launch script is recommended

Review project objectives with management and agree on and document team goals

Establish team roles

Define the teams development process

Make a quality plan and set quality targets

Plan for the needed support facilities

PROCESS MODELS

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

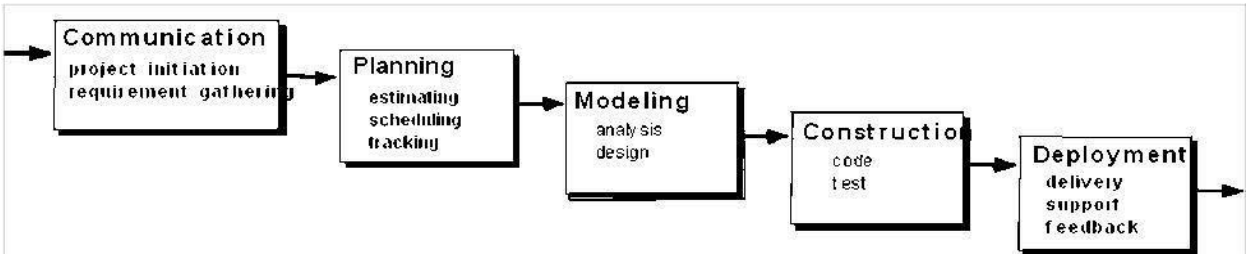
THE WATERFALL MODEL:

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

Context: Used when requirements are reasonably well understood.

Advantage:

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



The **problems** that are sometimes encountered when the waterfall model is applied are:

Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

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 exist at the beginning of many projects.

The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

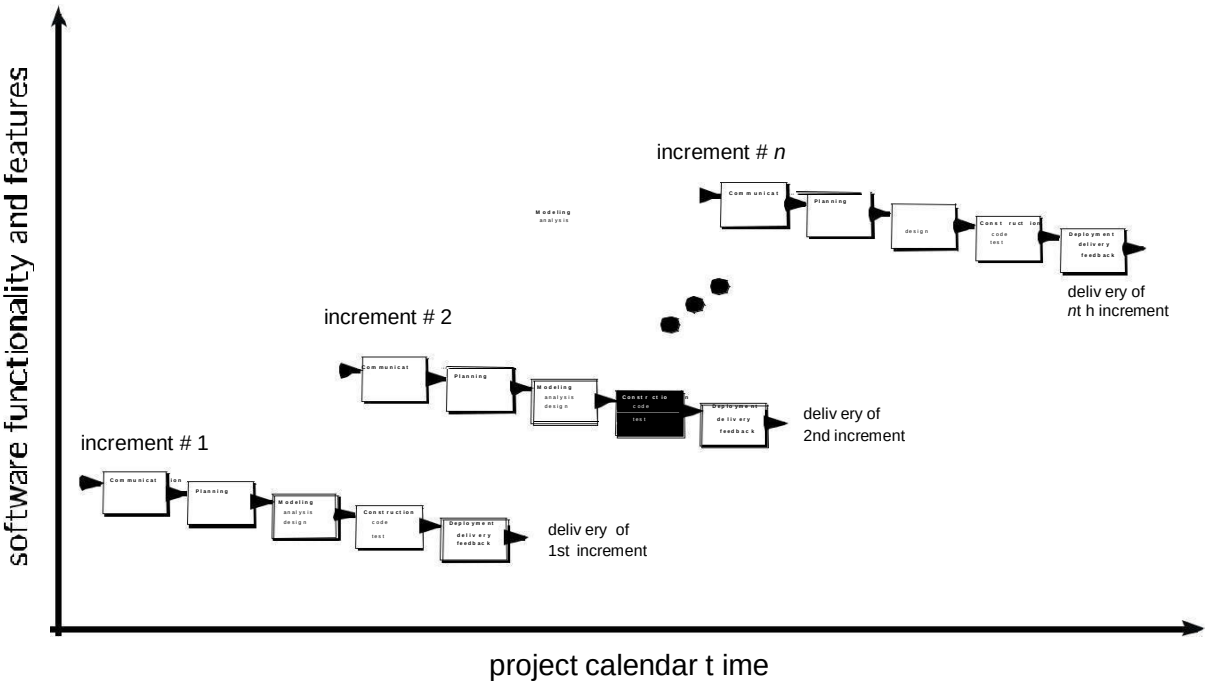
INCREMENTAL PROCESS MODELS:

The incremental model

The RAD model

THE INCREMENTAL MODEL:

Context: Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



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 provide progressively more functionality for the customer as each increment is delivered.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed. The core product is used by the customer. As a result, a plan is developed for the next increment.

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.

This process is repeated following the delivery of each increment, until the complete product is produced.

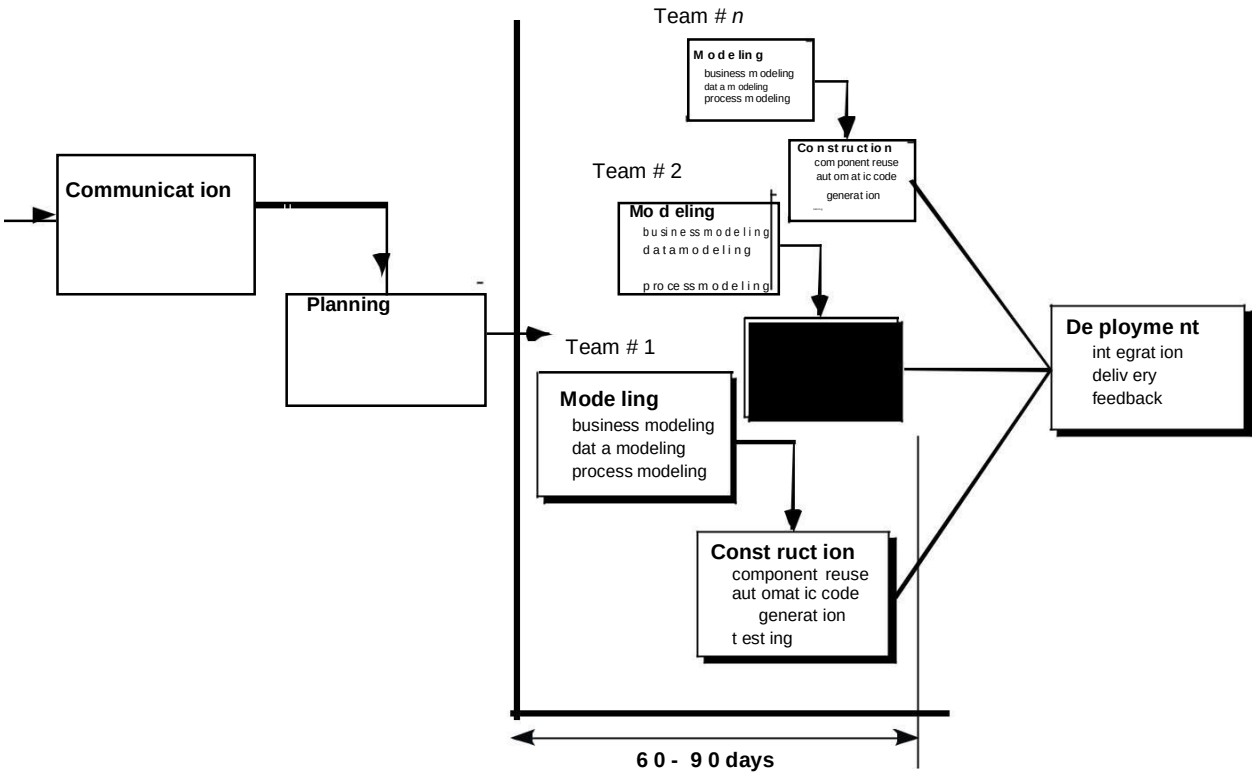
For *example*, word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking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

Difference: The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on delivery of an operational product with each increment

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within a very short time period.



The RAD approach maps into the generic framework activities.

> ***Communication*** works to understand the business problem and the information characteristics that the software must accommodate.
>
> ***Planning*** is essential because multiple software teams works in parallel on different system functions.
>
> ***Modeling*** encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.
> ***Deployment*** establishes a basis for subsequent iterations.

The RAD approach has **drawbacks**:

> For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
>
> If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail

If a system cannot be properly modularized, building the components necessary for RAD will be problematic

If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and

RAD may not be appropriate when technical risks are high.

EVOLUTIONARY PROCESS MODELS:

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

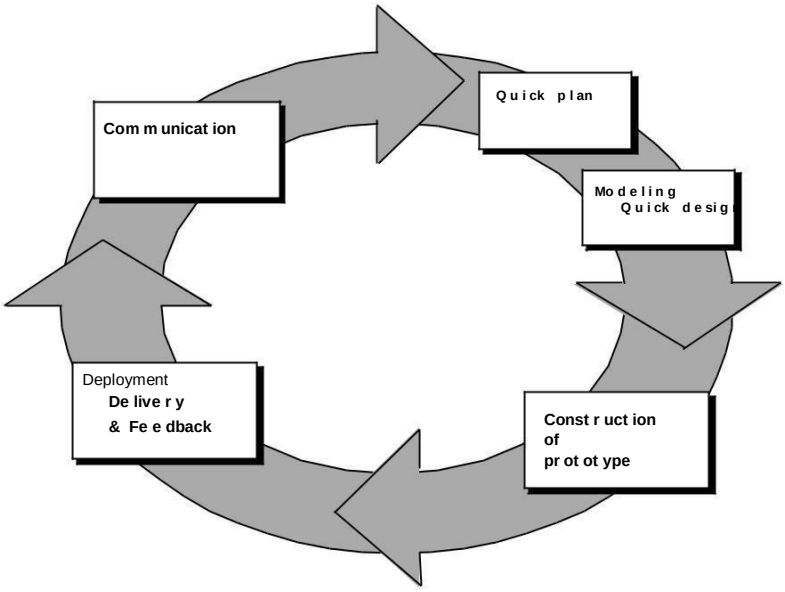
PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and 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 occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.



Context:

If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.

If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

Prototyping can be **problematic** for the following reasons:

The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.

The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

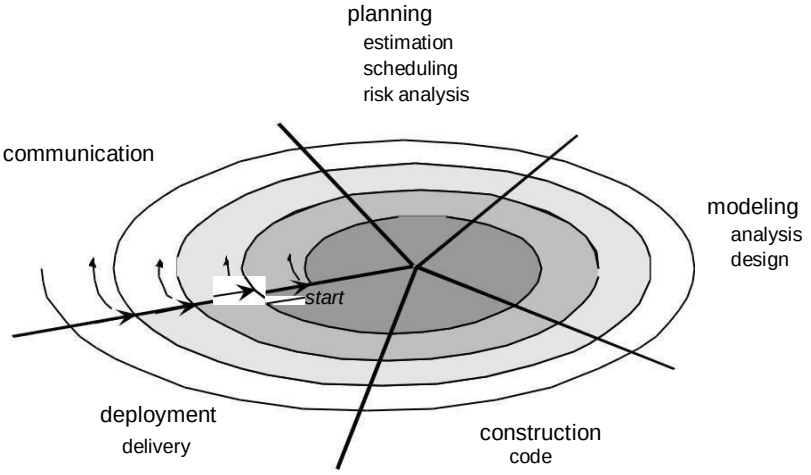
THE SPIRAL MODEL

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

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

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more

complete versions of the engineered system are



produced.

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.

The first circuit around the spiral might result in the development of product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan. Cost and 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.

It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.

Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

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

Advantages:

It provides the potential for rapid development of increasingly more complete versions of the software.

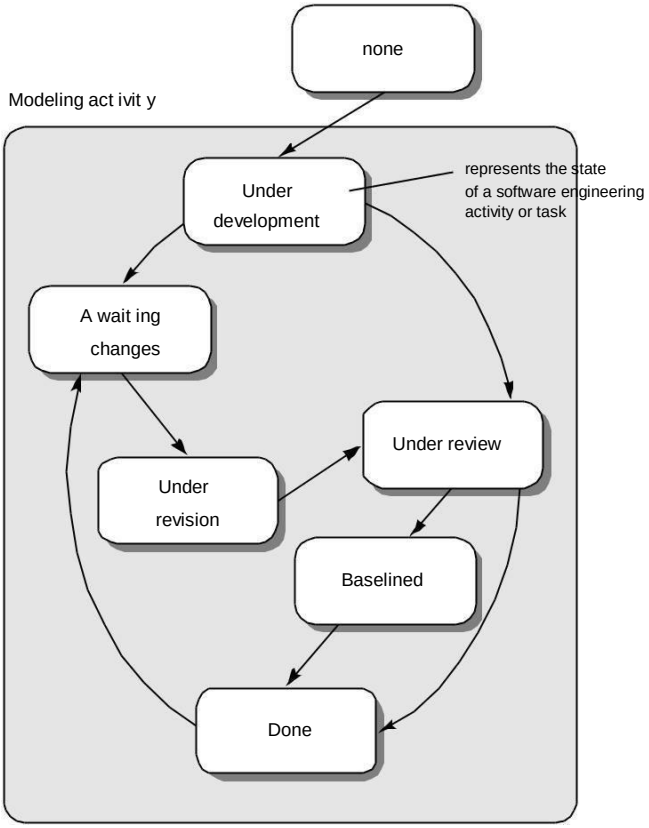
The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product

Draw Backs:

The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

THE CONCURRENT DEVELOPMENT MODEL:

The concurrent development model, sometimes called *concurrent engineering*, can be represented schematically as a series of framework activities, software engineering actions and tasks, and their associated states.



The activity *modeling* may be in anyone of the states noted at any given time. Similarly, other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states.

Any of the activities of a project may be in a particular state at any one time:
- under development
- awaiting changes
- under revision
- under review

In a project the *communication* activity has completed its first iteration and exists in the **awaiting changes** state. The modeling activity which existed in the **none** state while initial communication was completed, now makes a transition into the **under development** state. If, however, the customer indicates

that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.

The event analysis model correction which will trigger the analysis action from the **done** state into the **awaiting changes** state.

Context: The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved.

Advantages:

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.

It defines a network of activities rather than each activity, action, or task on the network exists simultaneously with other activities, action and tasks.

A FINAL COMMENT ON EVOLUTIONARY PROCESSES:

The concerns of evolutionary software processes are:

The first concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.

Second, evolutionary software process do not establish the maximum speed of the evolution. If the evolution occurs too fast, without a period of relaxation, it is certain that the process will fall into chaos.

Third, software processes should be focused on flexibility and extensibility rather than on high quality.

THE UNIFIED PROCESS:

The unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development.

The Unified process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse". If suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

A BRIEF HISTORY:

During the 1980s and into early 1990s, object-oriented (OO) methods and programming languages gained a widespread audience throughout the software engineering community. A wide variety of object-oriented analysis (OOA) and design (OOD) methods were proposed during the same time period.

During the early 1990s James Rumbaugh, Grady Booch, and Ival Jacobsom began working on a "Unified method" that would combine the best features of each of OOD & OOA. The result was UML- a

unified modeling language that contains a robust notation fot the modeling and development of OO systems.

By 1997, UML became an industry standard for object-oriented software development. At the same time, the Rational Corporation and other vendors developed automated tools to support UML methods.

Over the next few years, Jacobson, Rumbugh, and Booch developed the Unified process, a framework for object-oriented software engineering using UML. Today, the Unified process and UML are widely used on OO projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

PHASES OF THE UNIFIED PROCESS:

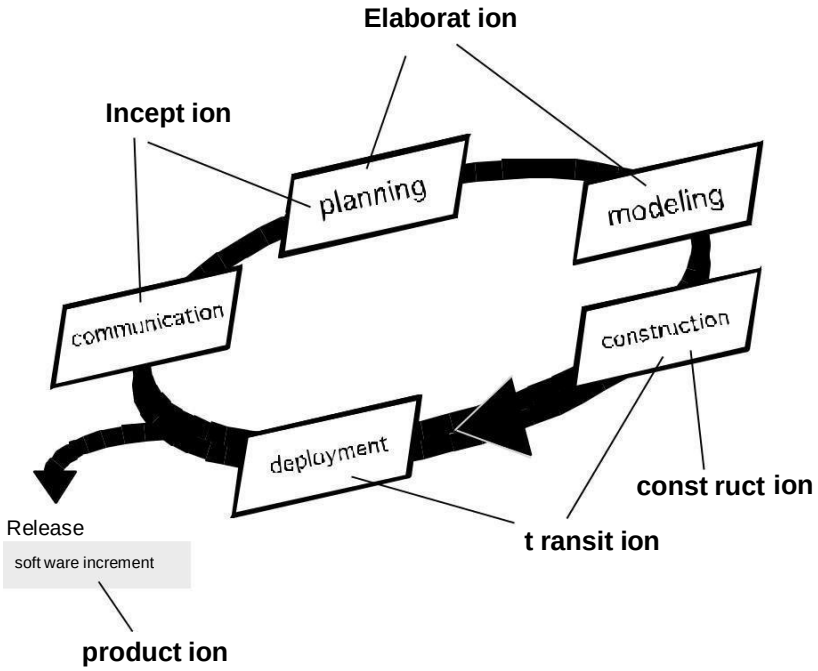
The ***inception*** phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, business requirements for the software are identified, a rough architecture for the system is proposed and a plan for the iterative, incremental nature of the ensuing project is developed.

The ***elaboration*** phase encompasses the customer communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use-cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software- the use-case model, the analysis model, the design model, the implementation model, and the deployment model.

The ***construction*** phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use-case operational for end-users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.

The ***transition*** phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment activity. Software given to end-users for beta testing, and user feedback reports both defects and necessary changes.

The ***production*** phase of the UP coincides with the deployment activity of the generic process. During this phase, the on-going use of the software is monitored, support for the operating environment is provided, and defect reports and requests for changes are submitted and evaluated.



A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set. That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.

UNIFIED PROCESS WORK PRODUCTS:

During the ***inception phase***, the intent is to establish an overall "vision" for the project,

identify a set of business requirements,

make a business case for the software, and

define project and business risks that may represent a threat to success.

The most important work product produced during the inception is the use-case modell-a collection of use-cases that describe how outside actors interact with the system and gain value from it. The use-case model is a collection of software features and functions by describing a set of preconditions, a flow of events and a set of post-conditions for the interaction that is depicted.

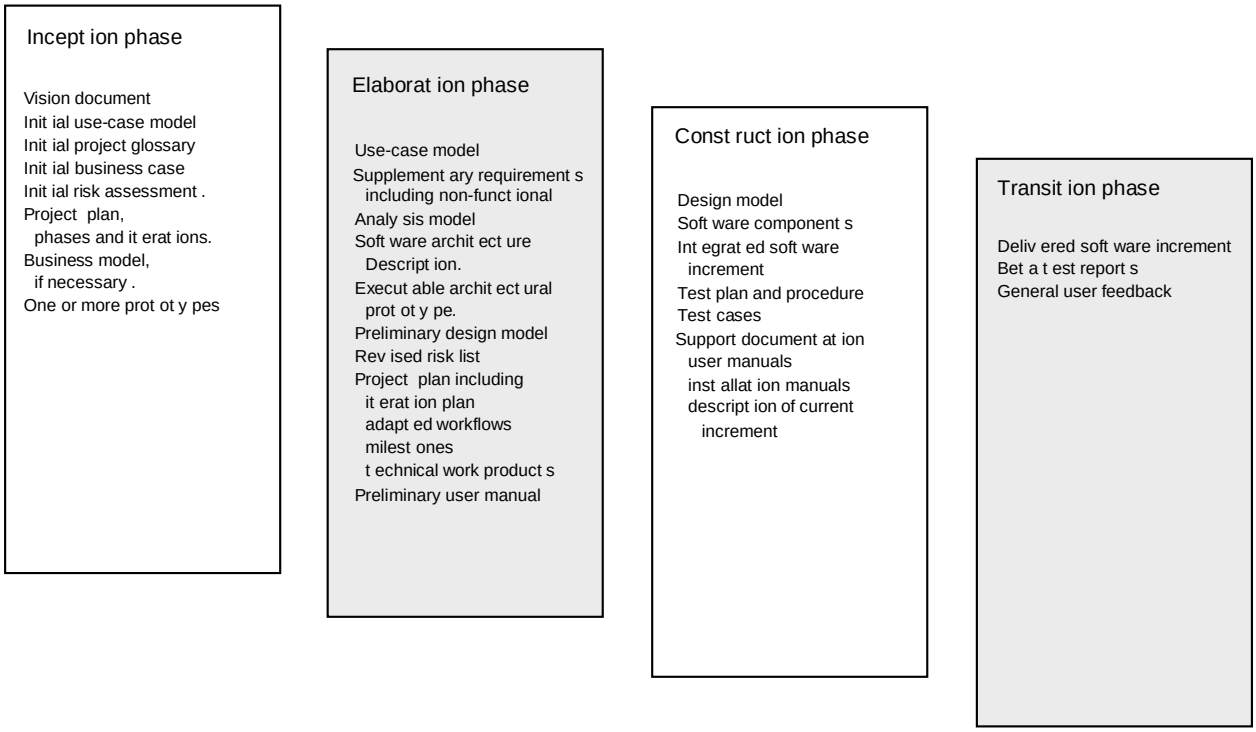
The use-case model is refined and elaborated as each UP phase is conducted and serves as an important input for the creation of subsequent work products. During the inception phase only 10 to 20 percent of the use-case model is completed. After elaboration, between 80 to 90 percent of the model has been created.

The ***elaboration phase*** produces a set of work products that elaborate requirements and produce and architectural description and a preliminary design. The UP analysis model is the work product that is developed as a consequence of this activity. The classes and analysis packages defined as part of the analysis model are refined further into a design model which identifies design classes, subsystems, and the interfaces between subsystems. Both the analysis and design models expand and refine an evolving

representation of software architecture. In addition the elaboration phase revisits risks and the project plan to ensure that each remains valid.

The ***construction phase*** produces an implementation model that translates design classes into software components into the physical computing environment. Finally, a test model describes tests that are used to ensure that use cases are properly reflected in the software that has been constructed.

The ***transition phase*** delivers the software increment and assesses work products that are produced as end-users work with the software. Feedback from beta testing and qualitative requests for change is produced at this time.



UNIT-2

SOFTWARE REQUIREMENTS

Software requirements are necessary

To introduce the concepts of user and system requirements
To describe functional and non-functional requirements
To explain how software requirements may be organised in a requirements document

What is a requirement?

The requirements for the system are the description of the services provided by the system and its operational constraints

It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

This is inevitable as requirements may serve a dual function

May be the basis for a bid for a contract - therefore must be open to interpretation; ○
May be the basis for the contract itself - therefore must be defined in detail;

Both these statements may be called requirements

Requirements engineering:

The process of finding out, analysing documenting and checking these services and constraints is called requirement engineering.

The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Requirements abstraction (Davis):

*If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The **requirements** must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation's needs. Once a contract has been awarded, the contractor must write a **system definition** for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the **requirements document** for the system."*

Types of requirement:

User requirements

Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

System requirements

A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Definitions and specifications:

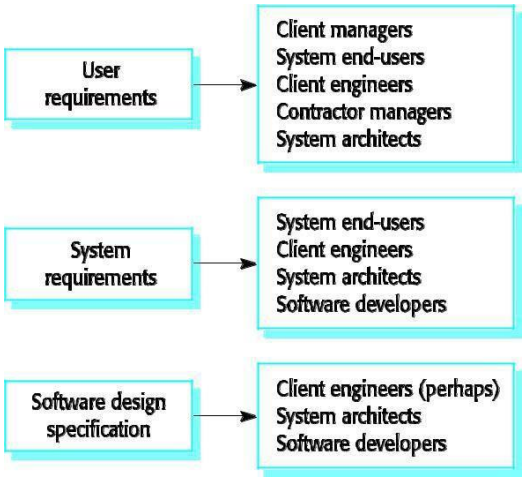
User Requirement Definition:

The software must provide the means of representing and accessing external files created by other tools.

System Requirement specification:

The user should be provided with facilities to define the type of external files.
Each external file type may have an associated tool which may be applied to the file.
Each external file type may be represented as a specific icon on the user's display.
Facilities should be provided for the icon representing an external file type to be defined by the user.
When an user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Requirements readers:



Functional and non-functional requirements:

Functional requirements

Statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations.

Non-functional requirements

Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Domain requirements

Requirements that come from the application domain of the system and that reflect characteristics of that domain.

FUNCTIONAL REQUIREMENTS:

Describe functionality or system services.

Depend on the type of software, expected users and the type of system where the software is used.

Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

The functional requirements for **The LIBSYS system:**

A library system that provides a single interface to a number of databases of articles in different libraries.

Users can search for, download and print these articles for personal study.

Examples of functional requirements

The user shall be able to search either all of the initial set of databases or select a subset from it.

The system shall provide appropriate viewers for the user to read documents in the document store.

Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

Requirements imprecision

Problems arise when requirements are not precisely stated.

Ambiguous requirements may be interpreted in different ways by developers and users.

Consider the term 'appropriate viewers'

- User intention - special purpose viewer for each different document type;
- Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements completeness and consistency:

In principle, requirements should be both complete and consistent.

Complete

They should include descriptions of all facilities required.

Consistent

There should be no conflicts or contradictions in the descriptions of the system facilities. In practice, it is impossible to produce a complete and consistent requirements document.

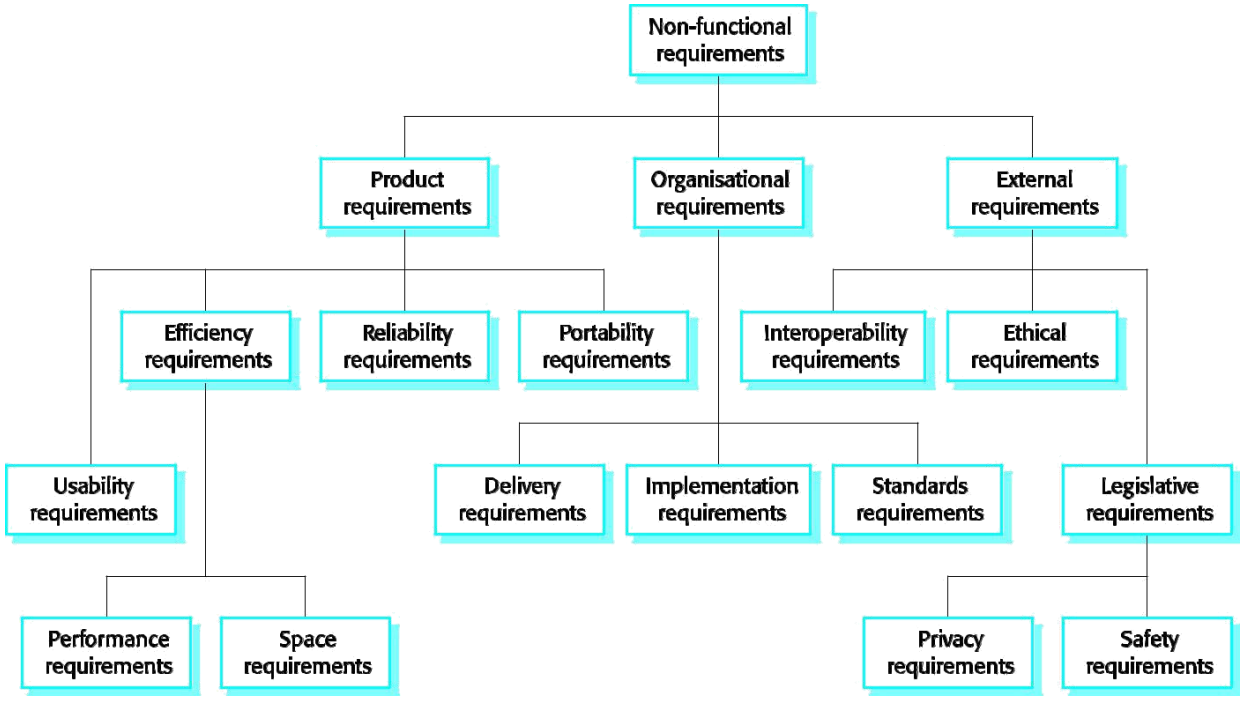
NON-FUNCTIONAL REQUIREMENTS

These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

Process requirements may also be specified mandating a particular CASE system, programming language or development method.

Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional requirement types:



Non-functional requirements :

Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

*Eg:*The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

Organisational requirements

Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

Eg: The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Eg: The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Goals and requirements:

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.

Goal

A general intention of the user such as ease of use.

The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.

Verifiable non-functional requirement

A statement using some measure that can be objectively tested.

Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.

Goals are helpful to developers as they convey the intentions of the system users.

Requirements measures:

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

Requirements interaction:

Conflicts between different non-functional requirements are common in complex systems.

Spacecraft system

To minimise weight, the number of separate chips in the system should be minimised.

To minimise power consumption, lower power chips should be used.

However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?

A common **problem with non-functional requirements** is that they can be difficult to verify. Users or customers often state these requirements as general goals such as ease of use, the ability of the system to

recover from failure or rapid user response. These vague goals cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered.

DOMAIN REQUIREMENTS

Derived from the application domain and describe system characteristics and features that reflect the domain.

Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.

If domain requirements are not satisfied, the system may be unworkable.

Library system domain requirements:

There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.

Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Domain requirements problems

Understandability

Requirements are expressed in the language of the application domain;

This is often not understood by software engineers developing the system.

Implicitness

Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

USER REQUIREMENTS

Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.

User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language

Lack of clarity

Precision is difficult without making the document difficult to read.

Requirements confusion

Functional and non-functional requirements tend to be mixed-up.

Requirements amalgamation

Several different requirements may be expressed together.

Requirement problems

Database requirements includes both conceptual and detailed information

Describes the concept of a financial accounting system that is to be included in LIBSYS;

However, it also includes the detail that managers can configure this system - this is unnecessary at this level.

Grid requirement mixes three different kinds of requirement

Conceptual functional requirement (the need for a grid);

Non-functional requirement (grid units);

Non-functional UI requirement (grid switching).

Structured presentation

Guidelines for writing requirements

Invent a standard format and use it for all requirements.

Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.

Use text highlighting to identify key parts of the requirement.

Avoid the use of computer jargon.

SYSTEM REQUIREMENTS

More detailed specifications of system functions, services and constraints than user requirements.

They are intended to be a basis for designing the system.

They may be incorporated into the system contract.

System requirements may be defined or illustrated using system models

Requirements and design

In principle, requirements should state what the system should do and the design should describe how it does this.

In practice, requirements and design are inseparable

A system architecture may be designed to structure the requirements;

The system may inter-operate with other systems that generate design requirements;

The use of a specific design may be a domain requirement.

Problems with NL(natural language) specification

Ambiguity

The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult.

Over-flexibility

The same thing may be said in a number of different ways in the

specification. Lack of modularisation

NL structures are inadequate to structure system requirements.

Alternatives to NL specification:

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT. Now, use-case descriptions and sequence diagrams are commonly used .
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Structured language specifications

The freedom of the requirements writer is limited by a predefined template for requirements.

All requirements are written in a standard way.

The terminology used in the description may be limited.

The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.

Form-based specifications

Definition of the function or entity.

Description of inputs and where they come from.

Description of outputs and where they go to.

Indication of other entities required.

Pre and post conditions (if appropriate).

The side effects (if any) of the function.

Tabular specification

Used to supplement natural language.

Particularly useful when you have to define a number of possible alternative courses of action.

Graphical models

Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.

Sequence diagrams

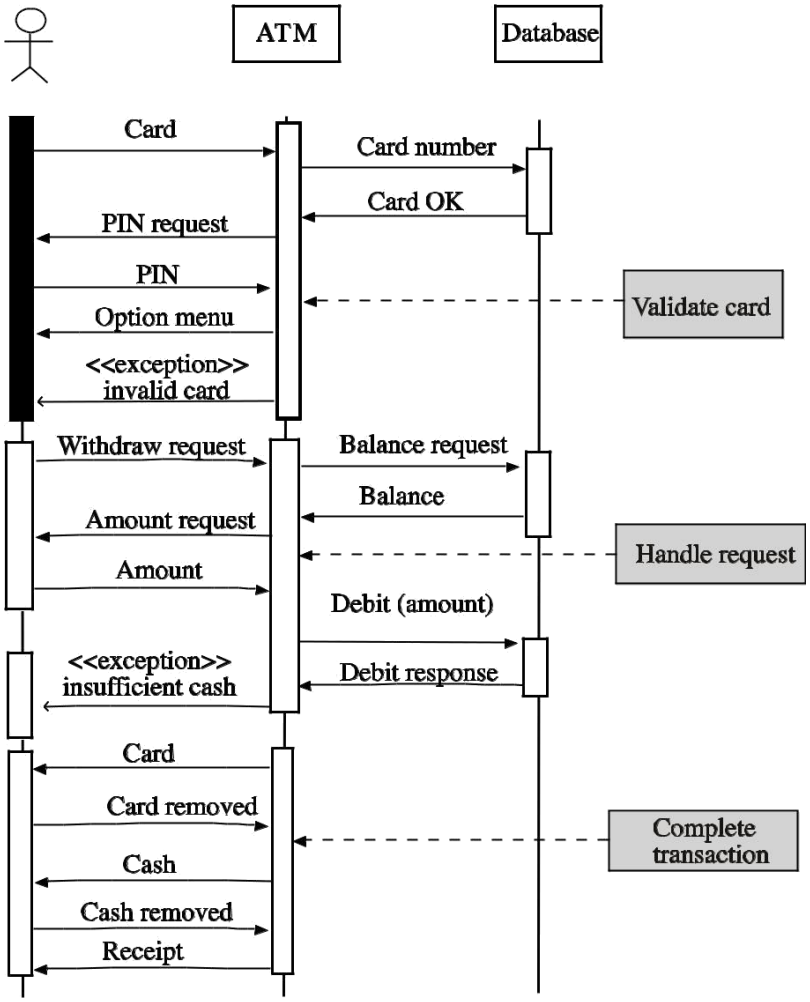
These show the sequence of events that take place during some user interaction with a system.
You read them from top to bottom to see the order of the actions that take place.
Cash withdrawal from an ATM

Validate card;

Handle request;

Complete transaction.

Sequence diagram of ATM withdrawal



System requirement specification using a standard form:

Function
Description
Inputs
Source
Outputs
Destination
Action
Requires
Pre-condition
Post-condition
Side-effects

When a standard form is used for specifying functional requirements, the following information should be included:

Description of the function or entity being specified

Description of its inputs and where these come from
Description of its outputs and where these go to
Indication of what other entities are used
Description of the action to be taken
If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called
Description of the side effects of the operation.

INTERFACE SPECIFICATION

Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.

Three types of interface may have to be defined

Procedural interfaces where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Applicatin Programming Interfaces (APIs)

Data structures that are exchanged **t**hat are passed from one sub-system to another. Graphical data models are the best notations for this type of description

Data representations that have been established for an existing sub-system

Formal notations are an effective technique for interface specification.

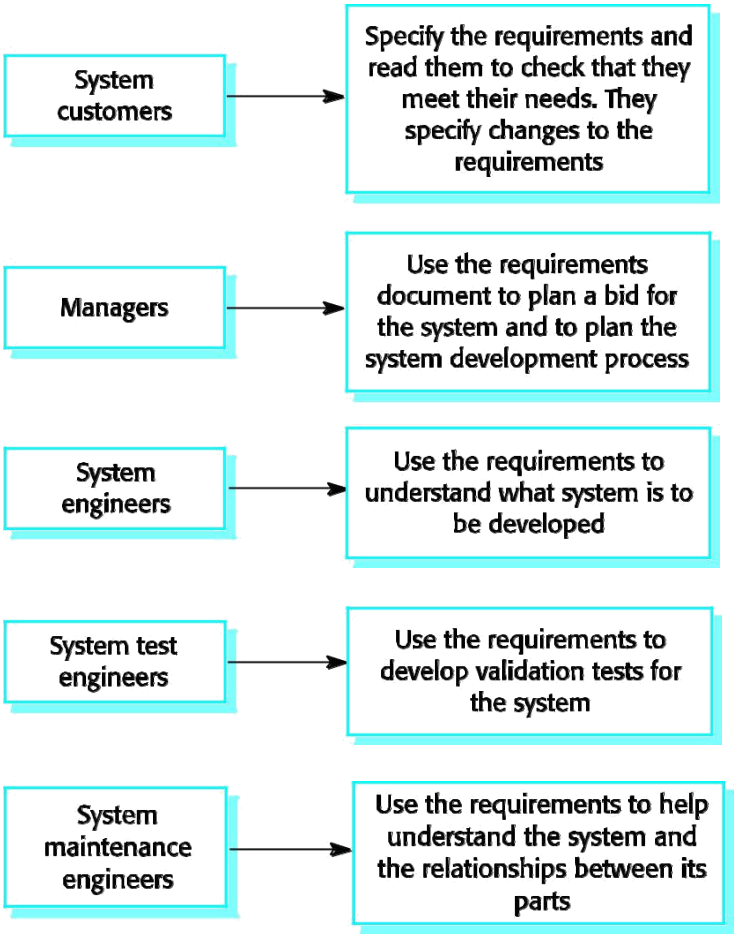
THE SOFTWARE REQUIREMENTS DOCUMENT:

The requirements document is the official statement of what is required of the system developers.

Should include both a definition of user requirements and a specification of the system requirements.

It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Users of a requirements document:



IEEE requirements standard defines a generic structure for a requirements document that must be instantiated for each specific system.

Introduction.
- Purpose of the requirements document
- Scope of the project
- Definitions, acronyms and abbreviations
- References
- Overview of the remainder of the document

General description.

- Product perspective
- Product functions
- User characteristics
- General constraints
- Assumptions and dependencies

Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance,

specify logical database requirements, design constraints, emergent system properties and quality characteristics.

Appendices.

Index.

REQUIREMENTS ENGINEERING PROCESSES

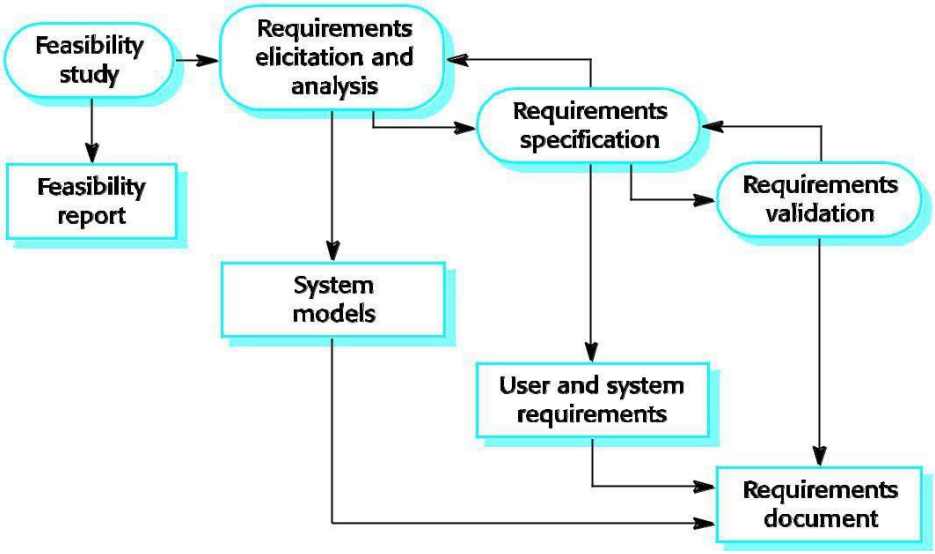
The **goal** of requirements engineering process is to create and maintain a system requirements document.

The overall process includes four high-level requirement engineering sub-processes. These are concerned with

Assessing whether the system is useful to the business(feasibility study)
Discovering requirements(elicitation and analysis)
Converting these requirements into some standard form(specification)
Checking that the requirements actually define the system that the customer wants(validation)

The process of managing the changes in the requirements is called **requirement management**.

The requirements engineering process



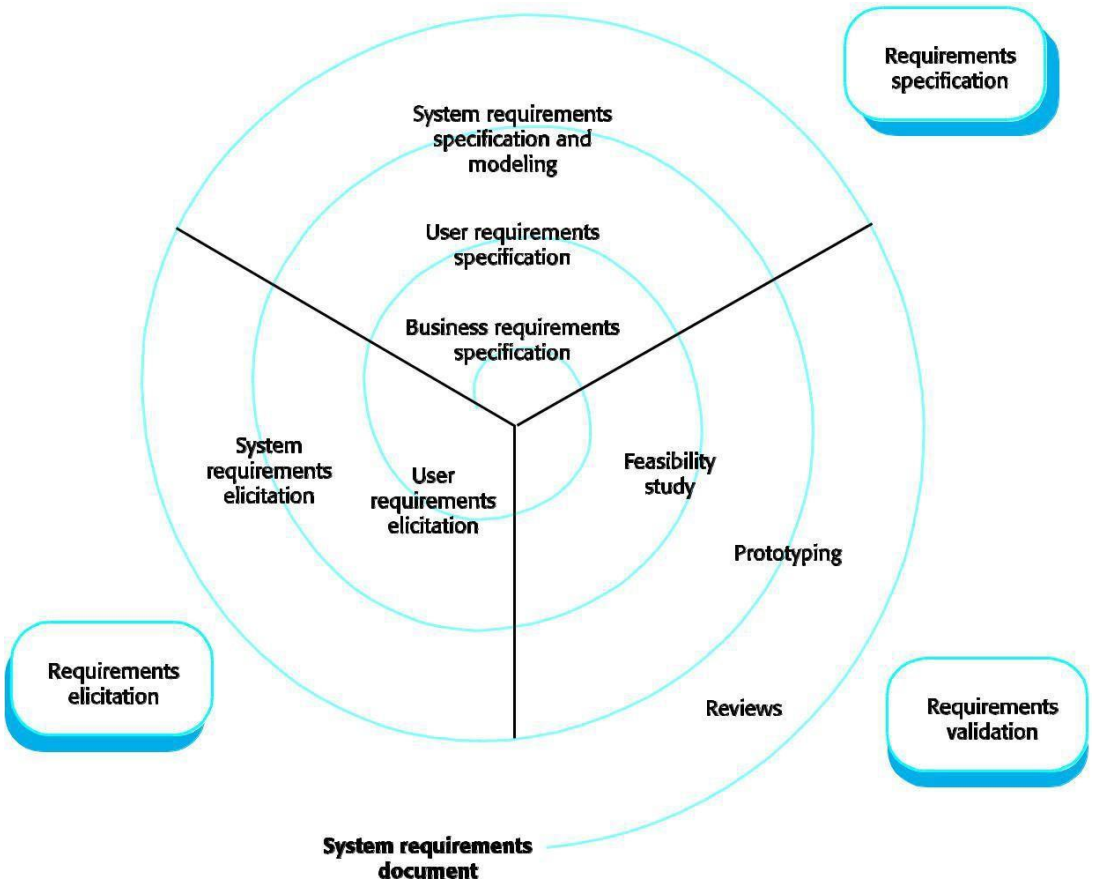
Requirements engineering:

The alternative perspective on the requirements engineering process presents the process as a **three-stage activity** where the activities are organized as an iterative process around a spiral. The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements and the user requirements. Later in the process, in the outer rings of the spiral, more effort will be devoted to system requirements engineering and system modeling.

This spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.

Some people consider requirements engineering to be the process of applying a structured analysis method such as object-oriented analysis. This involves analyzing the system and developing a set of graphical system models, such as use-case models, that then serve as a system specification. The set of models describes the behavior of the system and are annotated with additional information describing, for example, its required performance or reliability.

Spiral model of requirements engineering processes



1) FEASIBILITY STUDIES

A feasibility study decides whether or not the proposed system is worthwhile. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it worth carrying on with the requirements engineering and system development process.

A short focused study that checks

- If the system contributes to organisational objectives;

- If the system can be engineered using current technology and within budget;
- If the system can be integrated with other systems that are used.

Feasibility study implementation:

A feasibility study involves information assessment, information collection and report writing.

Questions for people in the organisation

- What if the system wasn't implemented?
- What are current process problems?
- How will the proposed system help?
- What will be the integration problems?
- Is new technology needed? What skills?
- What facilities must be supported by the proposed system?

In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system. They should try to complete a feasibility study in two or three weeks.

Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

2) REQUIREMENT ELICITATION AND ANALYSIS:

The requirement engineering process is requirements elicitation and analysis.

Sometimes called requirements elicitation or requirements discovery.

Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.

May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Problems of requirements analysis

Stakeholders don't know what they really want.

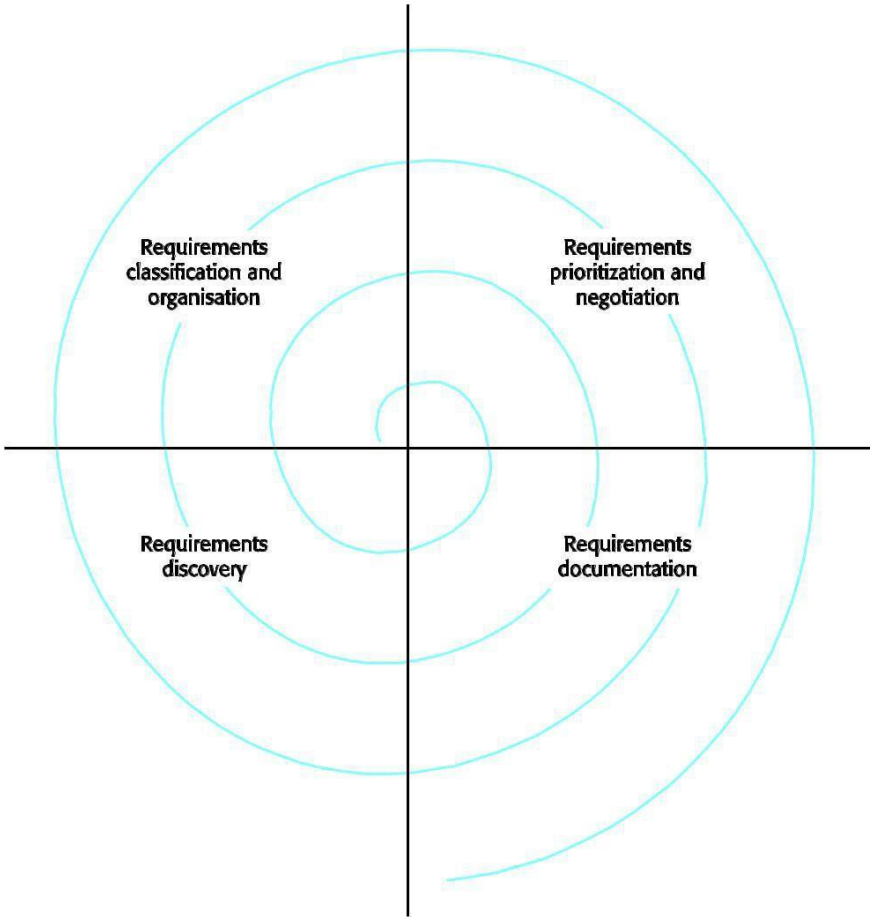
Stakeholders express requirements in their own terms.

Different stakeholders may have conflicting requirements.

Organisational and political factors may influence the system requirements.

The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

The requirements spiral



Process activities

Requirements discovery

- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

Requirements documentation

- Requirements are documented and input into the next round of the spiral.

The process cycle starts with requirements discovery and ends with requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle.

Requirements classification and organization is primarily concerned with identifying overlapping requirements from different stakeholders and grouping related requirements. The most common way of grouping requirements is to use a model of the system architecture to identify subsystems and to associate requirements with each sub-system.

Inevitably, stakeholders have different views on the importance and priority of requirements, and sometimes these view conflict. During the process, you should organize regular stakeholder negotiations so that compromises can be reached.

In the requirement documenting stage, the requirements that have been elicited are documented in such a way that they can be used to help with further requirements discovery.

REQUIREMENTS DISCOVERY:

Requirement discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.

Sources of information include documentation, system stakeholders and the specifications of similar systems.

They interact with stakeholders through interview and observation and may use scenarios and prototypes to help with the requirements discovery.

Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system.

For example, system stakeholder for a bank ATM include

Bank customers

Representatives of other banks

Bank managers

Counter staff

Database administrators

Security managers

Marketing department

Hardware and software maintenance engineers

Banking regulators

Requirements sources(stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoints, where each viewpoint presents a sub-set of the requirements for the system.

Viewpoints:

Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.

This multi-perspective analysis is important as there is no single correct way to analyse system requirements.

Types of viewpoint:

Interactor viewpoints

- People or other systems that interact directly with the system. These viewpoints provide detailed system requirements covering the system features and interfaces. In an ATM, the customer's and the account database are interactor VPs.

Indirect viewpoints

- Stakeholders who do not use the system themselves but who influence the requirements. These viewpoints are more likely to provide higher-level organisation requirements and constraints. In an ATM, management and security staff are indirect viewpoints.

Domain viewpoints

- Domain characteristics and constraints that influence the requirements. These viewpoints normally provide domain constraints that apply to the system. In an ATM, an example would be standards for inter-bank communications.

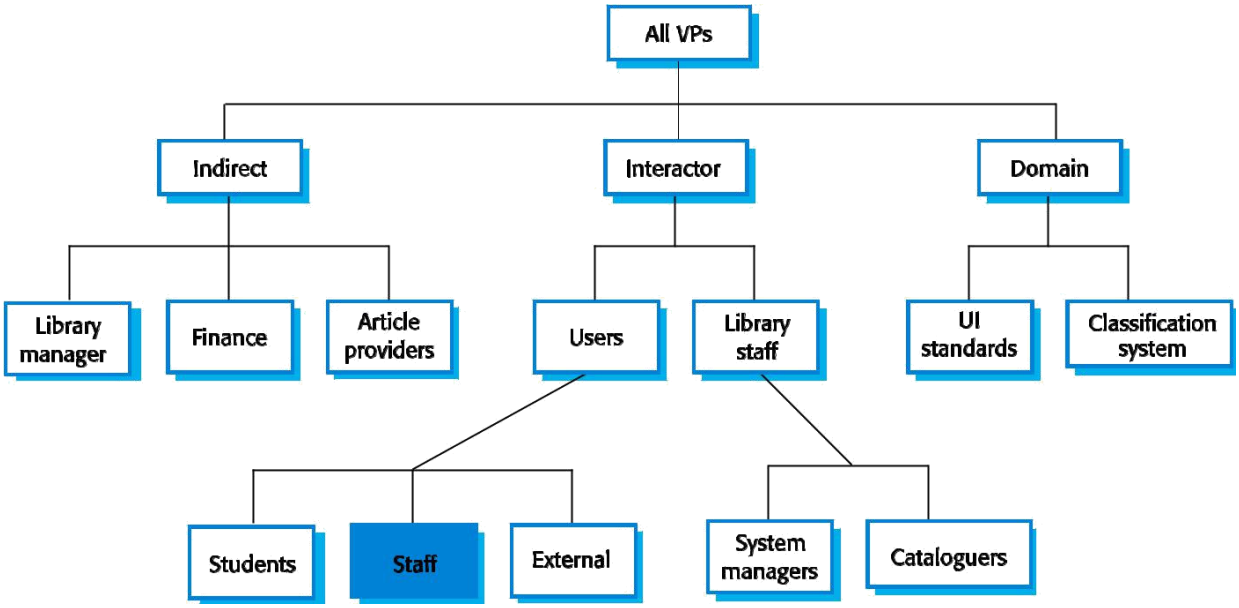
Typically, these viewpoints provide different types of requirements.

Viewpoint identification:

Identify viewpoints using

- Providers and receivers of system services;
- Systems that interact directly with the system being specified;
- Regulations and standards;
- Sources of business and non-functional requirements.
- Engineers who have to develop and maintain the system;
- Marketing and other business viewpoints.

LIBSYS viewpoint hierarchy



Interviewing

In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.

There are two types of interview

> **Closed interviews** where a pre-defined set of questions are answered.
>
> **Open interviews** where there is no pre-defined agenda and a range of issues are explored with stakeholders.

Interviews in practice:

> Normally a mix of closed and open-ended interviewing.
>
> Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
>
> Interviews are not good for understanding domain requirements
>
> - Requirements engineers cannot understand specific domain terminology;
> - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Effective interviewers:

> Interviewers should be open-minded, willing to listen to stakeholders and should not have pre-conceived ideas about the requirements.

They should prompt the interviewee with a question or a proposal and should not simply expect them to respond to a question such as ‘what do you want’.

Scenarios:

Scenarios are real-life examples of how a system can be used.

They should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes.

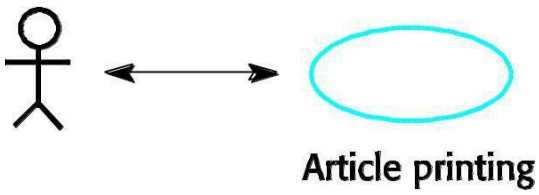
Use cases

Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.

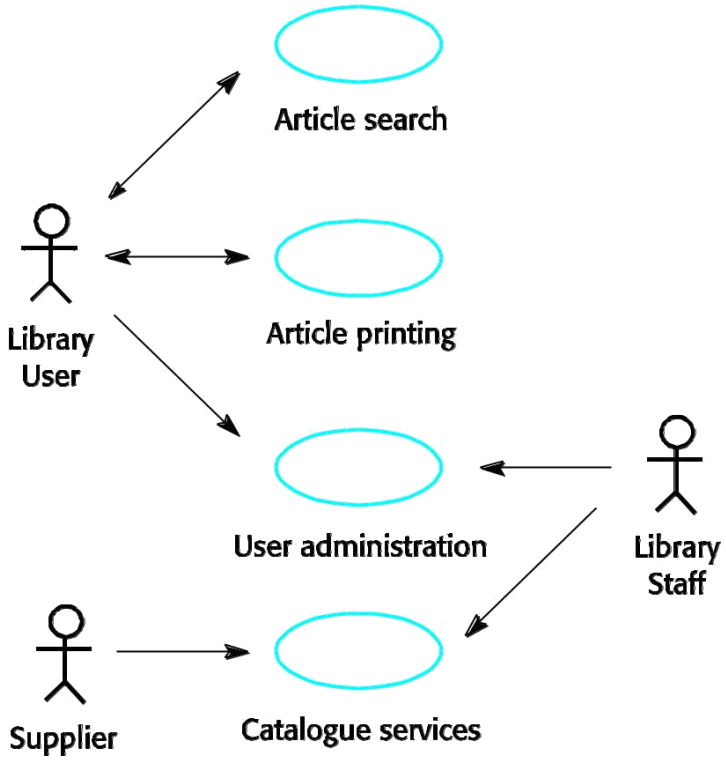
A set of use cases should describe all possible interactions with the system.

Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

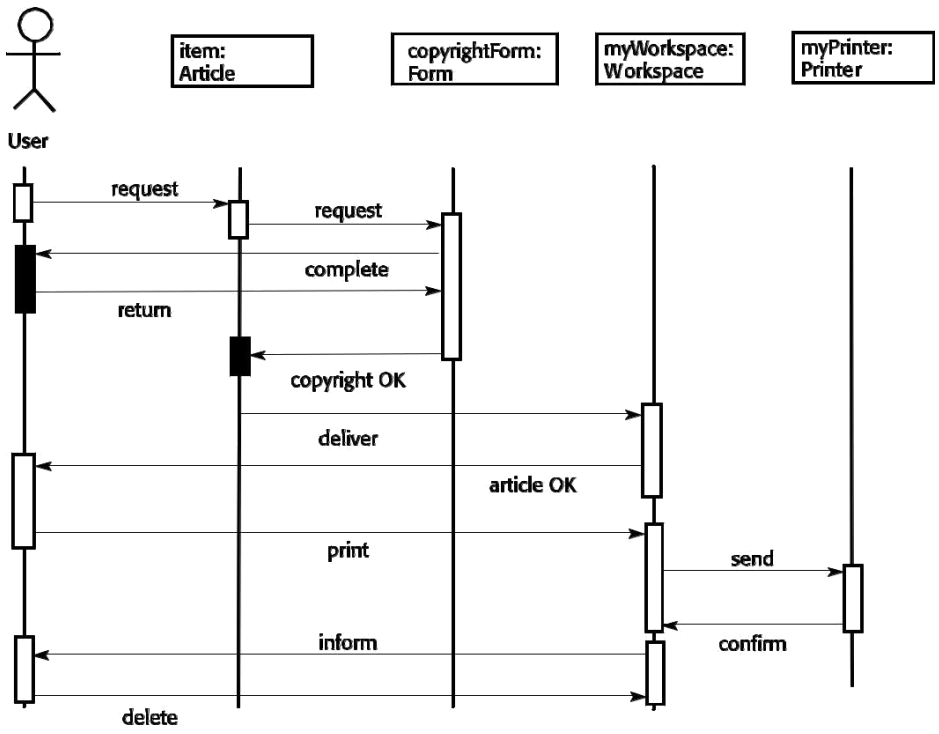
Article printing use-case:



LIBSYS use cases:



Article printing sequence:



Social and organisational factors

Software systems are used in a social and organisational context. This can influence or even dominate the system requirements.

Social and organisational factors are not a single viewpoint but are influences on all viewpoints.

Good analysts must be sensitive to these factors but currently no systematic way to tackle their analysis.

ETHNOGRAPHY:

A social scientists spends a considerable time observing and analysing how people actually work.

People do not have to explain or articulate their work.

Social and organisational factors of importance may be observed.

Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Focused ethnography:

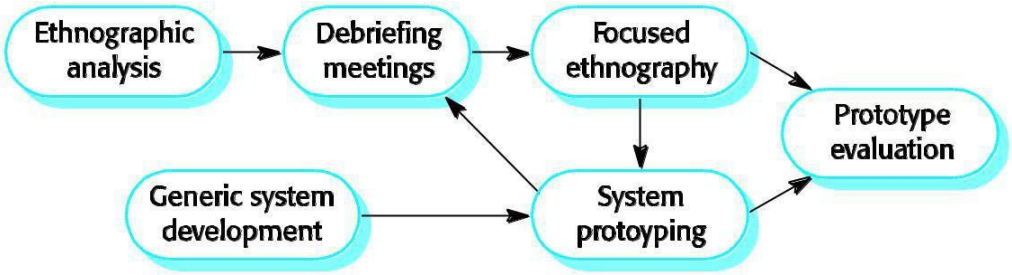
Developed in a project studying the air traffic control process

Combines ethnography with prototyping

Prototype development results in unanswered questions which focus the ethnographic analysis.

The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping



Scope of ethnography:

Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work.

Requirements that are derived from cooperation and awareness of other people's activities.

REQUIREMENTS VALIDATION

Concerned with demonstrating that the requirements define the system that the customer really wants.

Requirements error costs are high so validation is very important

- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking:

Validity: Does the system provide the functions which best support the customer's needs?

Consistency: Are there any requirements conflicts?

Completeness: Are all functions required by the customer included?

Realism: Can the requirements be implemented given available budget and technology

Verifiability: Can the requirements be checked?

Requirements validation techniques

Requirements reviews

- Systematic manual analysis of the requirements.

Prototyping

- Using an executable model of the system to check requirements. Covered in Chapter 17.

Test-case generation

- Developing tests for requirements to check testability.

Requirements reviews:

Regular reviews should be held while the requirements definition is being formulated.

Both client and contractor staff should be involved in reviews.

Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks:

***Verifiability*:** Is the requirement realistically testable?

***Comprehensibility*:** Is the requirement properly understood?

***Traceability*:** Is the origin of the requirement clearly stated?

***Adaptability*:** Can the requirement be changed without a large impact on other requirements?

REQUIREMENTS MANAGEMENT

Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

Requirements are inevitably incomplete and inconsistent

- New requirements emerge during the process as business needs change and a better understanding of the system is developed;
- Different viewpoints have different requirements and these are often contradictory.

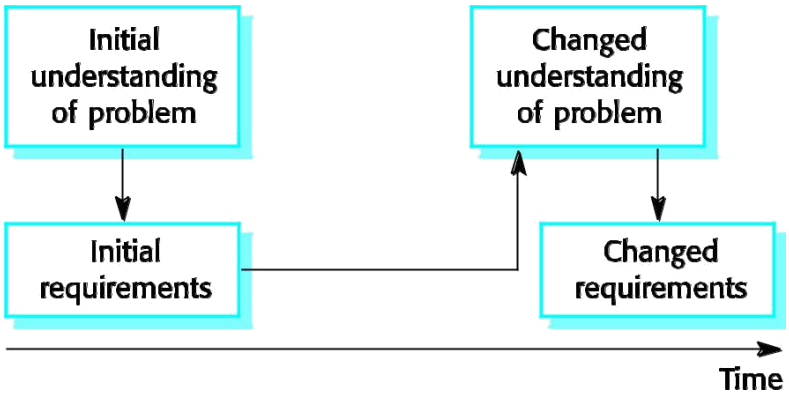
Requirements change

The priority of requirements from different viewpoints changes during the development process.

System customers may specify requirements from a business perspective that conflict with end-user requirements.

The business and technical environment of the system changes during its development.

Requirements evolution:



Enduring and volatile requirements:

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

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

Requirements classification:

Requirement Type	Description
Mutable requirements	Requirements that change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organisations processes and open up new ways of working which generate new system requirements
Compatibility requirements	Requirements that depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

Requirements management planning:

During the requirements engineering process, you have to plan:

- Requirements identification

  How requirements are individually identified;

- A change management process

  The process followed when analysing a requirements change;

- Traceability policies

  The amount of information about requirements relationships that is maintained;

- CASE tool support

  The tool support required to help manage requirements change;

Traceability:

Traceability is concerned with the relationships between requirements, their sources and the system design

Source traceability

- Links from requirements to stakeholders who proposed these requirements;

Requirements traceability

- Links between dependent requirements;

Design traceability - Links from the requirements to the design;

CASE tool support:

Requirements storage

- Requirements should be managed in a secure, managed data store.

Change management

- The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.

Traceability management

- Automated retrieval of the links between requirements.

Requirements change management:

Should apply to all proposed changes to the requirements.

Principal stages

- Problem analysis. Discuss requirements problem and propose change;

– Change analysis and costing. Assess effects of change on other requirements;

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

Change management:

SYSTEM MODELLING

System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Different models present the system from different perspectives

Behavioural perspective showing the behaviour of the system;

Structural perspective showing the system or data architecture.

Model types

Data processing model showing how the data is processed at different stages.

Composition model showing how entities are composed of other entities.

Architectural model showing principal sub-systems.

Classification model showing how entities have common characteristics.

Stimulus/response model showing the system's reaction to events.

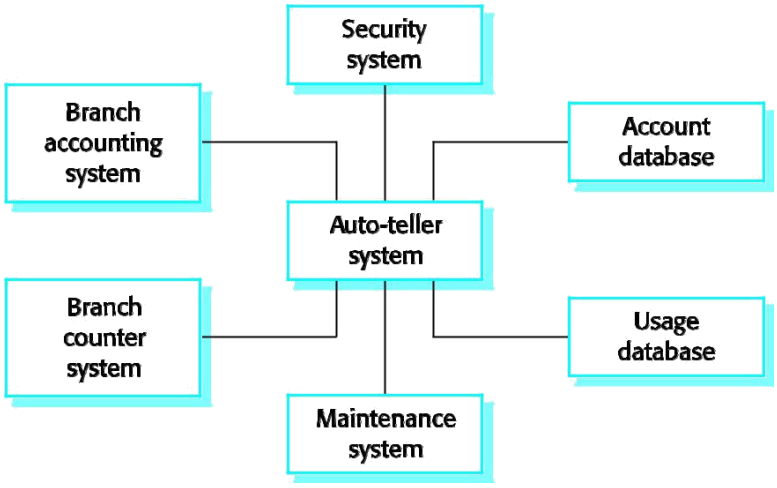
1) CONTEXT MODELS:

Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.

Social and organisational concerns may affect the decision on where to position system boundaries.

Architectural models show the system and its relationship with other systems.

The context of an ATM system:



Process models:

Process models show the overall process and the processes that are supported by the system.

Data flow models may be used to show the processes and the flow of information from one process to another.

2) BEHAVIOURAL MODELS:

Behavioural models are used to describe the overall behaviour of a system.

Two types of behavioural model are:

Data processing models that show how data is processed as it moves through the system;

State machine models that show the systems response to events.

These models show different perspectives so both of them are required to describe the system's behaviour.

Data-processing models:

Data flow diagrams (DFDs) may be used to model the system's data processing.

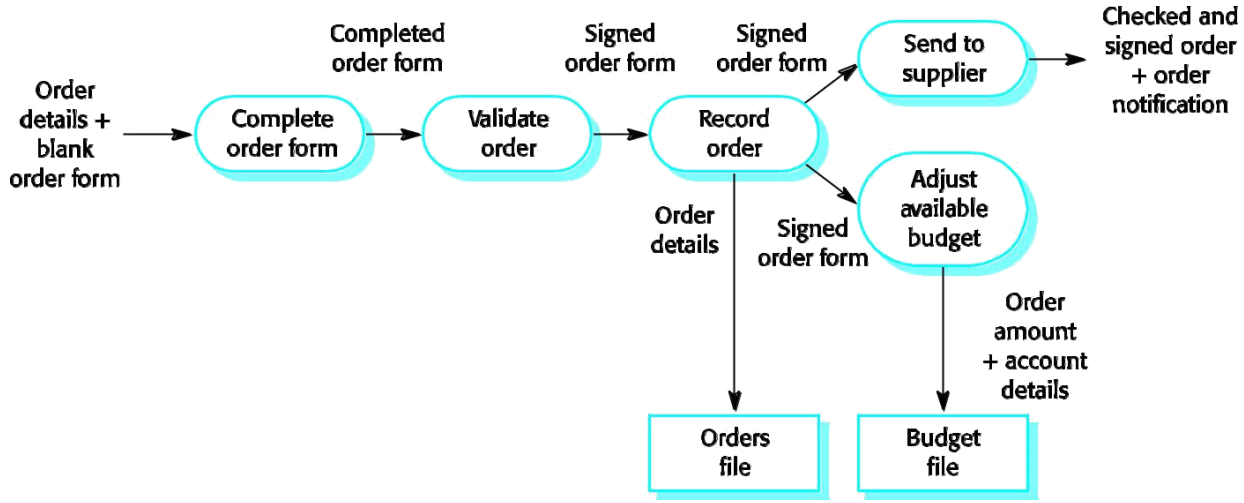
These show the processing steps as data flows through a system.

DFDs are an intrinsic part of many analysis methods.

Simple and intuitive notation that customers can understand.

Show end-to-end processing of data.

Order processing DFD:



Data flow diagrams:

DFDs model the system from a functional perspective.

Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.

Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

State machine models:

These model the behaviour of the system in response to external and internal events.

They show the system's responses to stimuli so are often used for modelling real-time systems.

State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.

Statecharts are an integral part of the UML and are used to represent state machine models.

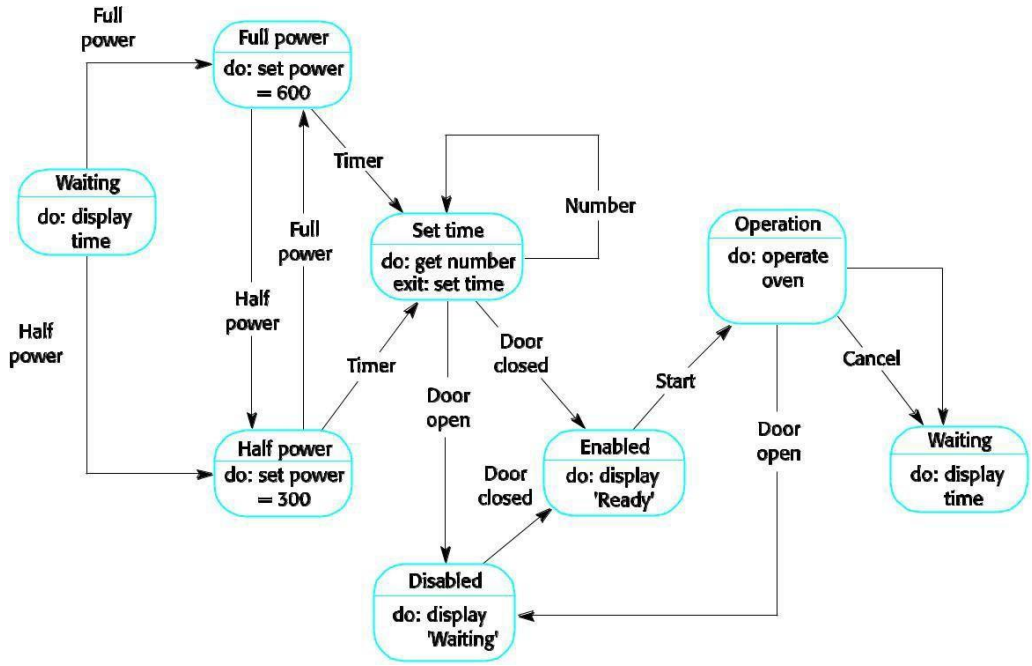
Statecharts:

Allow the decomposition of a model into sub-models (see following slide).

A brief description of the actions is included following the 'do' in each state.

Can be complemented by tables describing the states and the stimuli.

Microwave oven model:



Microwave oven state description:

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows ‘Half power’.
Full power	The oven power is set to 600 watts. The display shows ‘Full power’.
Set time	The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

Microwave oven stimuli:

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

3) SEMANTIC DATA MODELS:

Used to describe the logical structure of data processed by the system.

An entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes

Widely used in database design. Can readily be implemented using relational databases.

No specific notation provided in the UML but objects and associations can be used.

Data dictionaries

Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.

Advantages

Support name management and avoid duplication;

Store of organisational knowledge linking analysis, design and implementation;

Many CASE workbenches support data dictionaries.

4) OBJECT MODELS:

Object models describe the system in terms of object classes and their associations.

An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.

Various object models may be produced

Inheritance models;

Aggregation models;

Interaction models.

Natural ways of reflecting the real-world entities manipulated by the system

More abstract entities are more difficult to model using this approach

Object class identification is recognised as a difficult process requiring a deep understanding of the application domain

Object classes reflecting domain entities are reusable across systems

Inheritance models:

Organise the domain object classes into a hierarchy.

Classes at the top of the hierarchy reflect the common features of all classes.

Object classes inherit their attributes and services from one or more super-classes. these may then be specialised as necessary.

Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

Object models and the UML:

The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.

It has become an effective standard for object-oriented modelling.

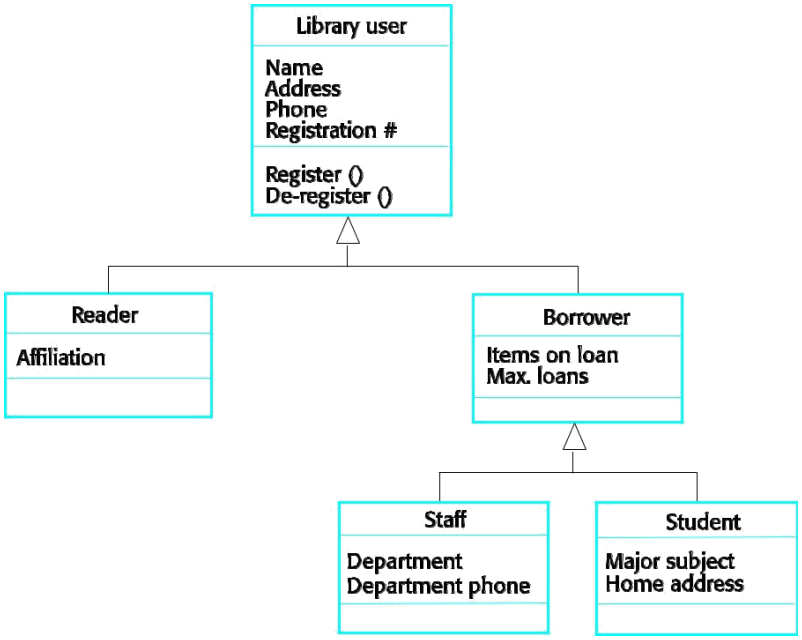
Notation

Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;

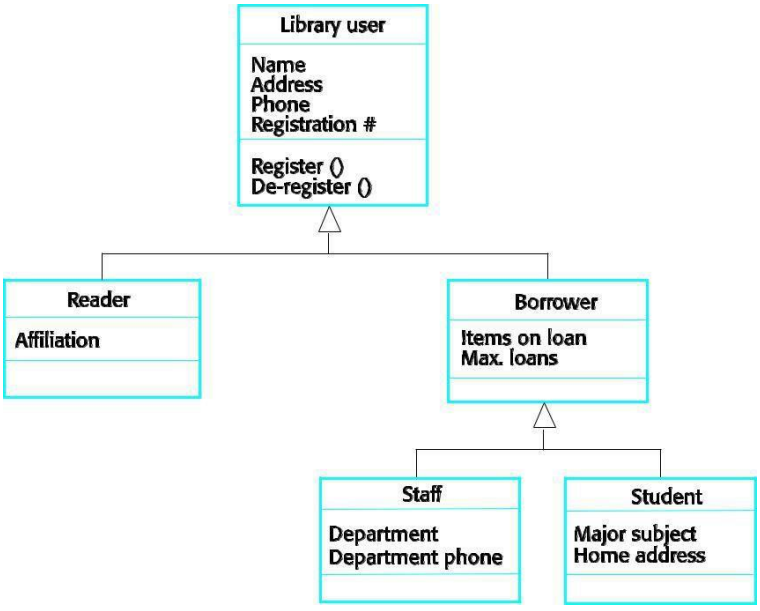
Relationships between object classes (known as associations) are shown as lines linking objects;

Inheritance is referred to as generalisation and is shown ‘upwards’ rather than ‘downwards’ in a hierarchy.

Library class hierarchy:



User class hierarchy:



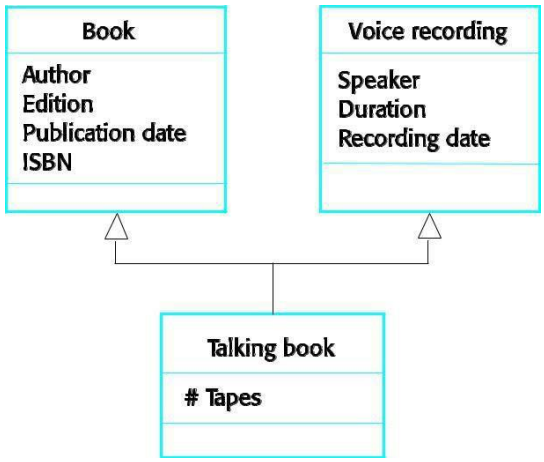
Multiple inheritance:

Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.

This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics.

Multiple inheritance makes class hierarchy reorganisation more complex.

Multiple inheritance

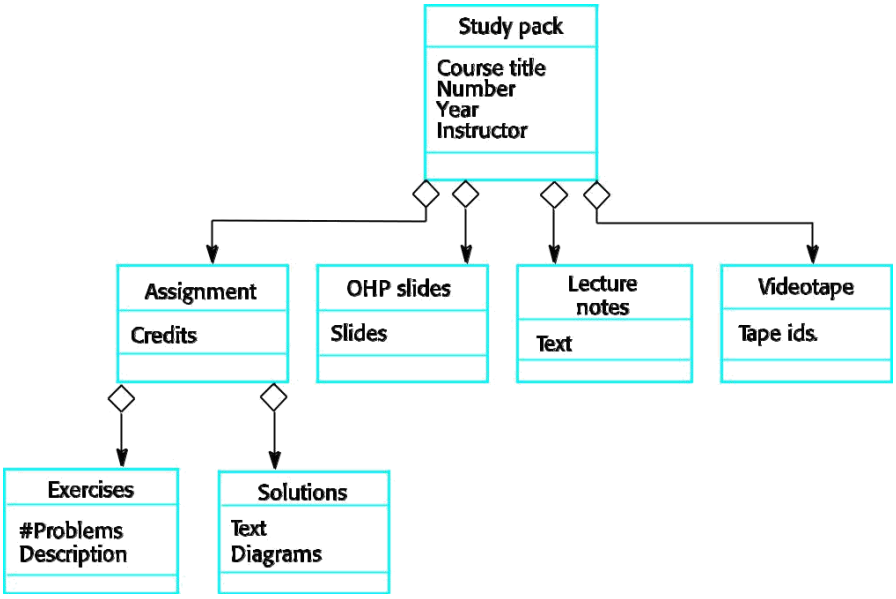


Object aggregation:

An aggregation model shows how classes that are collections are composed of other classes.

Aggregation models are similar to the part-of relationship in semantic data models.

Object aggregation



Object behaviour modelling

A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case.

Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects.

5) STRUCTURED METHODS:

Structured methods incorporate system modelling as an inherent part of the method.

Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models.

CASE tools support system modelling as part of a structured method.

Method weaknesses:

They do not model non-functional system requirements.

They do not usually include information about whether a method is appropriate for a given problem.

The may produce too much documentation.

The system models are sometimes too detailed and difficult for users to understand.

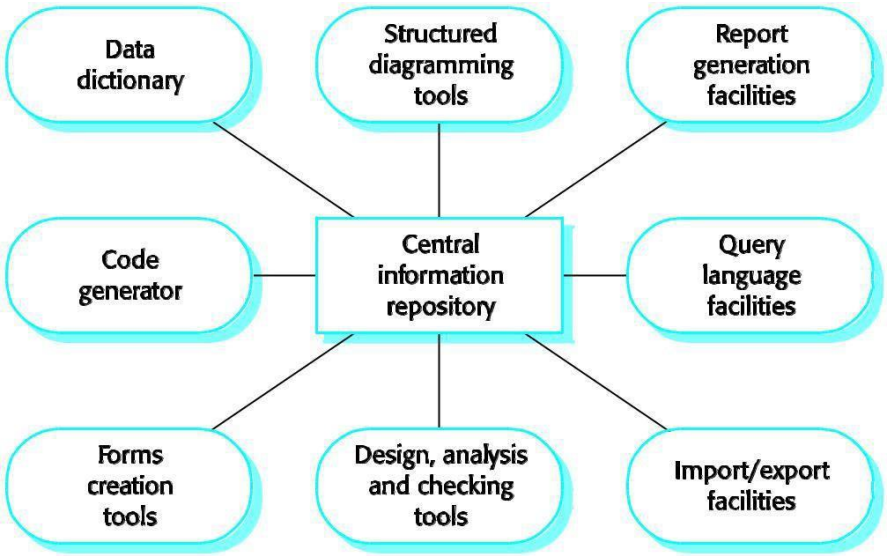
CASE workbenches:

A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.

Analysis and design workbenches support system modelling during both requirements engineering and system design.

These workbenches may support a specific design method or may provide support for a creating several different types of system model.

An analysis and design workbench



Analysis workbench components:

Diagram editors

Model analysis and checking tools

Repository and associated query language

Data dictionary

Report definition and generation tools

Forms definition tools

Import/export translators

Code generation tools

UNIT-3

DESIGN ENGINEERING

Design engineering encompasses the **set of principals, concepts, and practices** that lead to the development of a high- quality system or product.

Design principles establish an overriding philosophy that guides the designer in the work that is performed.
Design concepts must be understood before the mechanics of design practice are applied and

Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

What is design:

Design is what virtually every engineer wants to do. It is the place where creativity rules – customer's requirements, business needs, and technical considerations all come together in the formulation of a product or a system. Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Why is it important:

Design allows a software engineer to model the system or product that Is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end – users become involved in large numbers. Design is the place where software quality is established.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Another **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

1) DESIGN PROCESS AND DESIGN QUALITY:

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

Goals of design:

McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design.

The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality guidelines:

In order to evaluate the quality of a design representation we must establish technical criteria for good design. These are the following guidelines:

A design should exhibit an architecture that

has been created using recognizable architectural styles or patterns

is composed of components that exhibit good design characteristics and

can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

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

A design should contain distinct representation of data, architecture, interfaces and components.

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

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

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

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

A design should be represented using a notation that effectively communication its meaning.

These design guidelines are not achieved by chance. Design engineering encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

Quality attributes:

The FURPS quality attributes represent a target for all software design:

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

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

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), 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 c***ombines the ability to extend the program (extensibility), adaptability, serviceability- these three attributes represent a more common term maintainability

Not every software quality attribute is weighted equally as the software design is developed.

One application may stress functionality with a special emphasis on security.

Another may demand performance with particular emphasis on processing speed.

A third might focus on reliability.

2) DESIGN CONCEPTS:

M.A Jackson once said:"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

Abstraction: Many levels of abstraction are there.

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.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are suppressed.

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 operation, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

Architecture:

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

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

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

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

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

Patterns:

Brad Appleton defines a ***design pattern*** in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns." Stated in another way, a design pattern describes a design structure that solves a particular design 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

Whether the pattern is capable to the current work,
Whether the pattern can be reused,
Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

IV. Modularity:

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

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

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; chamges can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding:

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

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

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

VI. Functional Independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. *Functional independence* is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the program structure.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding.

A cohesion module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

VII. Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Niklaus wirth. A program is development by successively refining levels of procedural detail. A hierarchy is development by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, 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 detail as each successive refinement occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

VIII. Refactoring :

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

IX. Design classes:

The software team must define a set of design classes that

Refine the analysis classes by providing design detail that will enable the classes to be implemented, and

Create a new set of design classes that implement a software infrastructure to support the design solution.

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

User interface classes: define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a *metaphor* and the design classes for the interface may be visual representations of the elements of the metaphor.

Business domain classes: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.

Process classes implement lower – level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. Each design class be reviewed to ensure that it is "well-formed." They define **four characteristics of a well- formed design class.**

Complete and sufficient: A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness: Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion: A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.

Low coupling: Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction, called the *law of Demeter*, suggests that a method should only sent messages to methods in neighboring classes.

THE DESIGN MODEL:

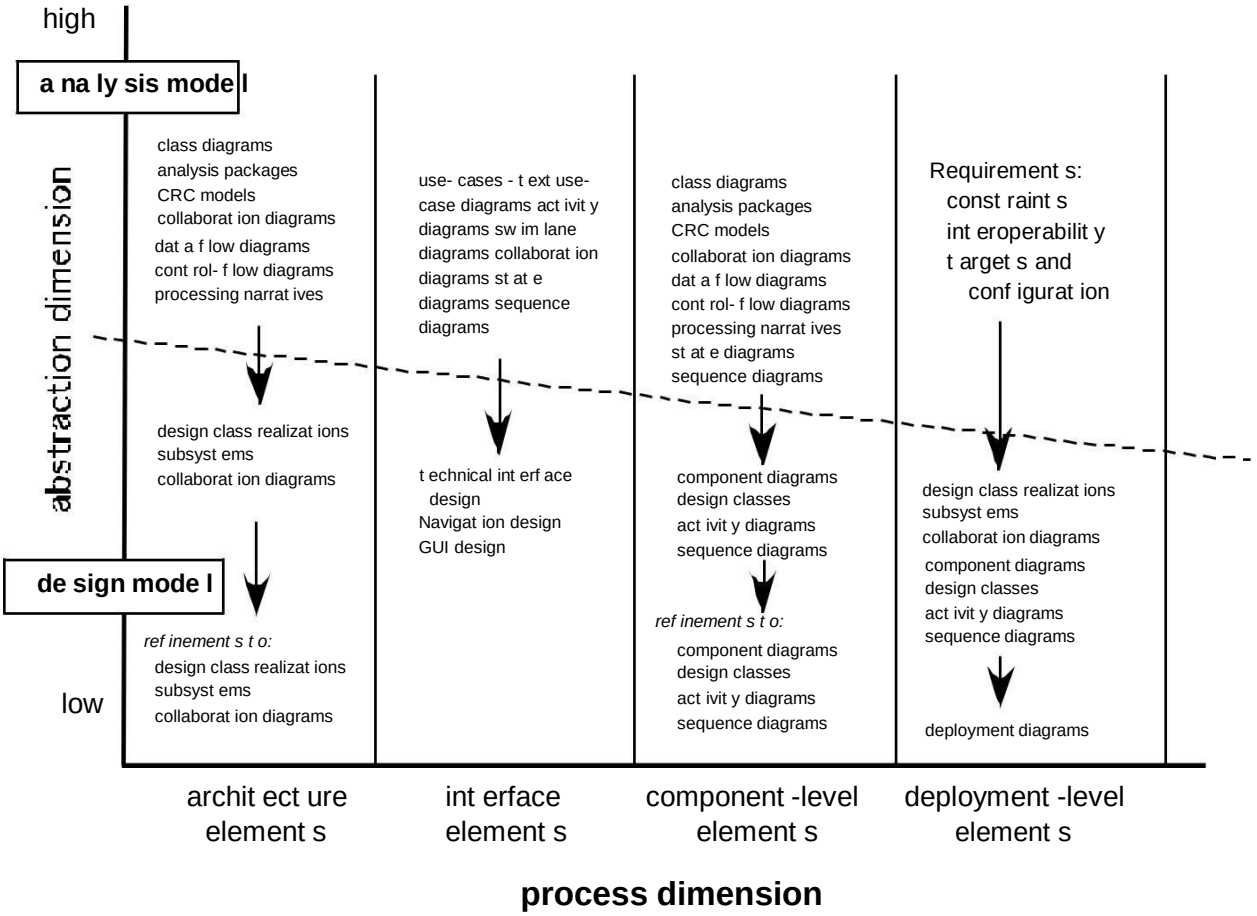
The design model can be viewed into different dimensions.

The process dimension indicates the evolution of the design model as design tasks are executed as a part of the software process.

The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as a path of design; more implementation- specific detail is provided, and architectural structure and style, components that reside within the architecture, and the interface between the components and with the outside world are all emphasized.

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model us usually delayed until the design has been fully developed.



Data design elements:

Data design sometimes referred to as data architecting creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

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

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

At **the application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
At the **business leve**l, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural design elements:

The *architectural design* for software is the equivalent to the floor plan of a house.
The architectural model is derived from three sources.

Information about the application domain for the software to be built.
Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
The availability of architectural patterns

Interface design elements:

The *interface design* for software is the equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.
The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are 3 important elements of interface design:

The user interface(UI);
External interfaces to other systems, devices, networks, or other produces or consumers of information; and
Internal interfaces between various design components.

These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design is a major software engineering action.

The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

UML defines an *interface* in the following manner:"an interface is a specifier for the externally-visible operations of a class, component, or other classifier without specification of internal structure."

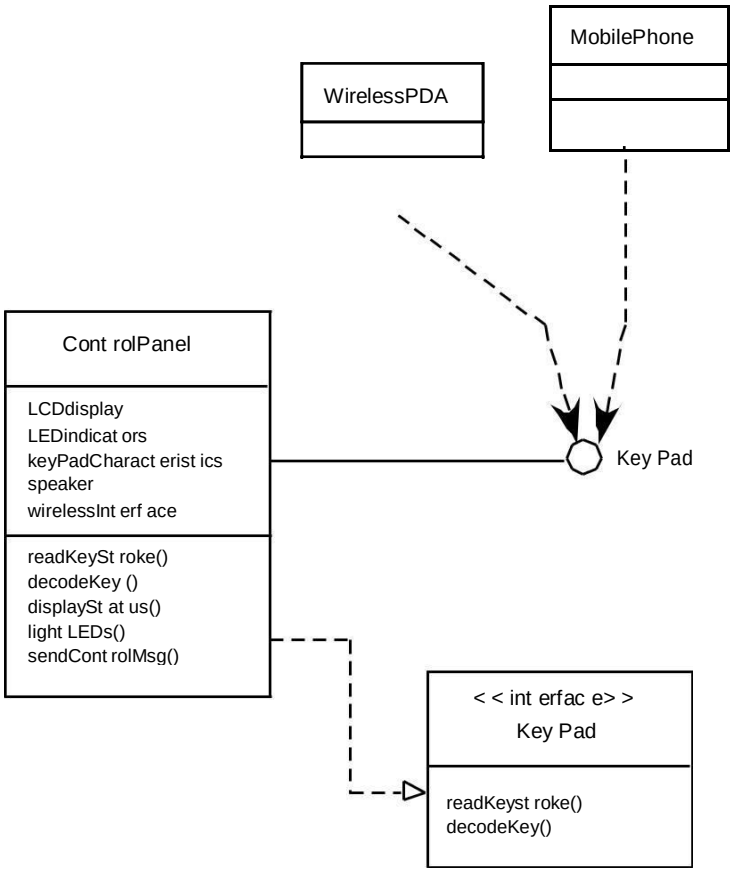
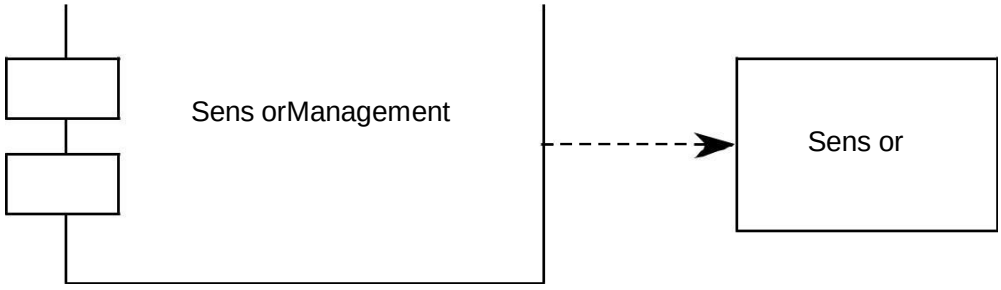


Figure 9 . 6 UML int erfac e represent at ion for **Co n t ro lPa n e l**

Component- level design elements: The component-level design for software is equivalent to a set of detailed drawings.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.



Deployment-level design elements: Deployment-level design elements indicated how software functionality and subsystems will be allocated within the physical computing environment that will support the software

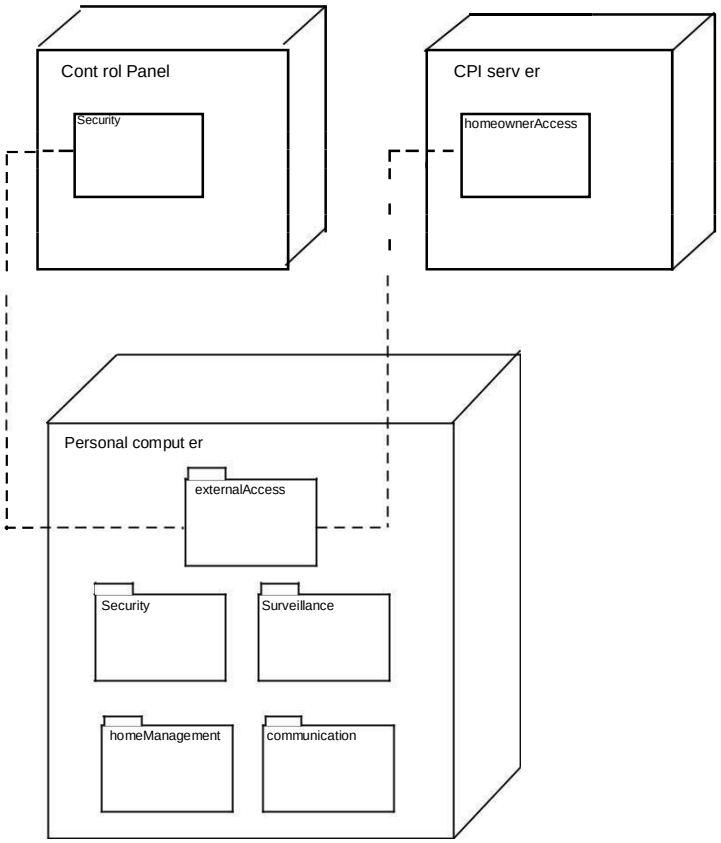


Figure 9 . 8 UML deploy m ent diagram for *SafeHom e*

ARCHITECTURAL DESIGN

SOFTWARE ARCHITECTURE:

What Is Architecture?

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers

the architectural style that the system will take,

the structure and properties of the components that constitute the system, and

the interrelationships that occur among all architectural components of a system.

The architecture is a representation that enables a software engineer to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, (3) reducing the risks associated with the construction of the software. The

design of software architecture considers two levels of the design pyramid

data design

architectural design.

Data design enables us to represent the data component of the architecture.

Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?

Bass and his colleagues [BAS98] identify three key reasons that software architecture is important:

Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

DATA DESIGN:

The data design activity translates data objects as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

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

At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.

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

Data design at the Architectural Level:

The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional.

To solve this challenge, the business IT community has developed ***data mining*** techniques, also called ***knowledge discovery in databases*** (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a ***data warehouse***, adds an additional layer to the data architecture. a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business.

Data design at the Component Level:

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. The following set of principles for data specification:

The systematic analysis principles applied to function and behavior should also be applied to data.

All data structures and the operations to be performed on each should be identified.

A data dictionary should be established and used to define both data and program design.

Low-level data design decisions should be deferred until late in the design process.

The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.

A library of useful data structures and the operations that may be applied to them should be developed.

A software design and programming language should support the specification and realization of abstract data types.

ARCHITECTURAL STYLES AND PATTERNS:

The builder has used an ***architectural style*** as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod).

The software that is built for computer-based systems also exhibits one of many architectural styles.

Each style describes a system category that encompasses

A set of ***components*** (e.g., a database, computational modules) that perform a function required by a system;

A set of ***connectors*** that enable "communication, coordinations and cooperation" among components;

Constraints that define how components can be integrated to form the system; and

Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

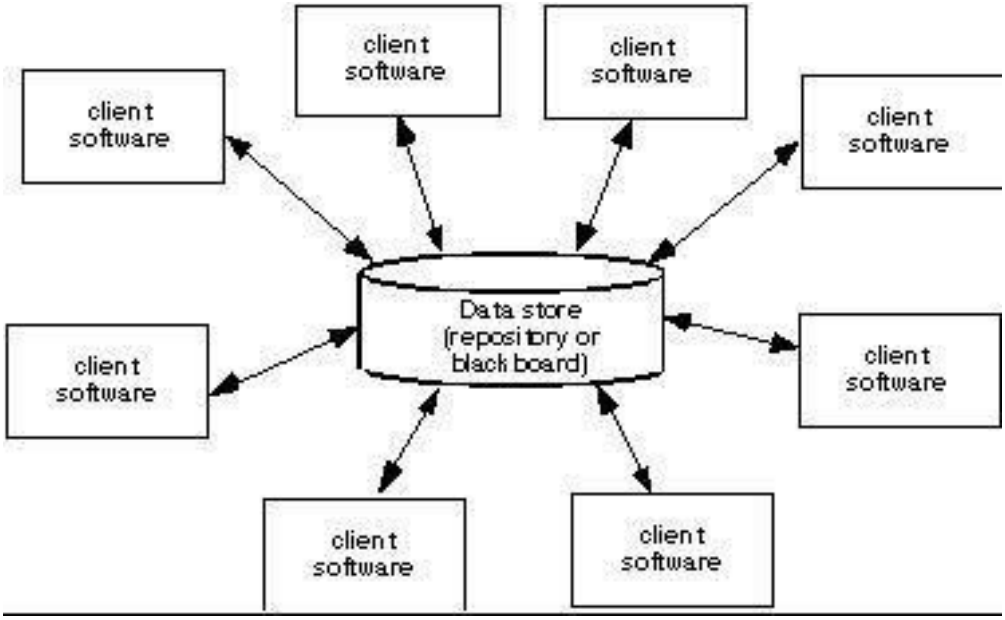
Architectural patterns tend to address specific behavioral issues within the context of the architectural.

A Brief Taxonomy of Styles and Patterns

Data-centered architectures:

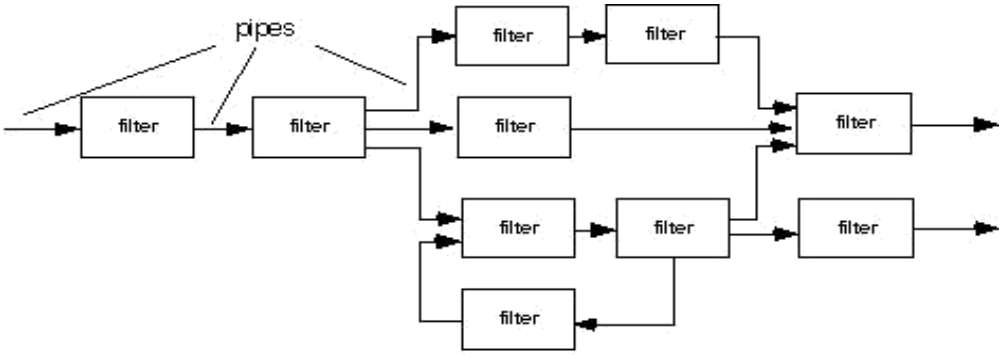
A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A variation on this approach transforms the repository into a "blackboard" that sends notification to client software when data of interest to the client changes

Data-centered architectures promote ***integrability***. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism

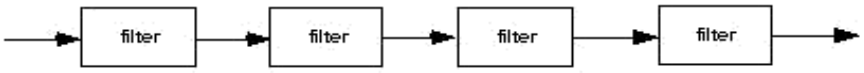


Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A ***pipe and filter pattern*** has a set of components, called ***filters***, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output of a specified form.

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



(a) pipes and filters

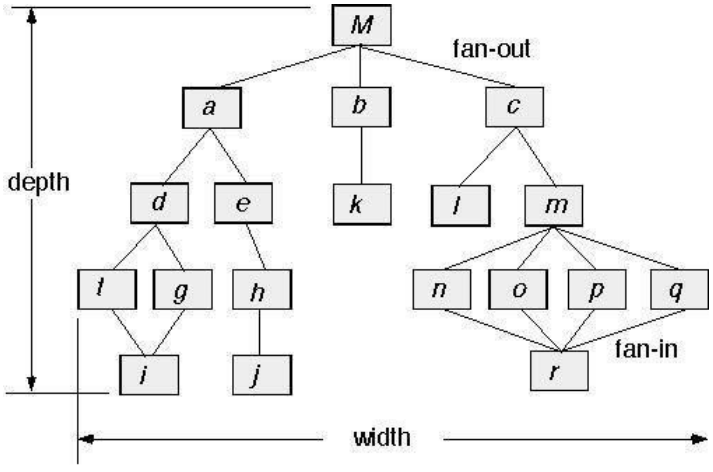


(b) batch sequential

Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

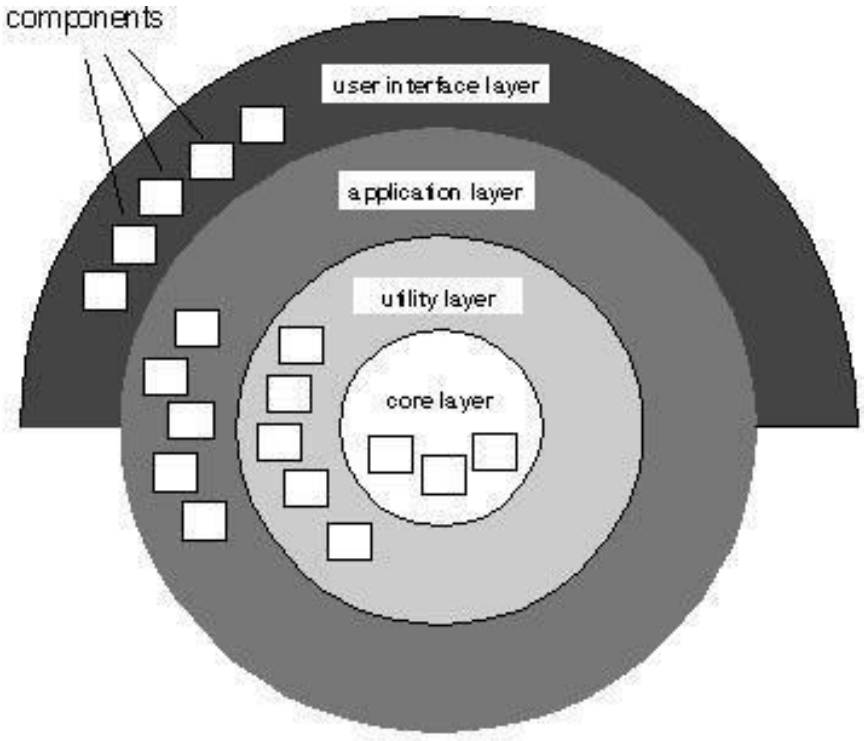
Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.

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



Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 14.3. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



Architectural Patterns:

An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

(3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

operating system process management pattern

task scheduler pattern

Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

a ***database management system*** pattern that applies the storage and retrieval capability of a DBMS to the application architecture

an ***application level persistence*** pattern that builds persistence features into the application architecture

Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

A ***broker*** acts as a 'middle-man' between the client component and a server component**.**

Organization and Refinement:

The design process often leaves a software engineer with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into the architectural style that has been derived:

Control.

How is control managed within the architecture?

Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?

How do components transfer control within the system?

How is control shared among components?

Data.

How are data communicated between components?

Is the flow of data continuous, or are data objects passed to the system sporadically?

What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?

Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?

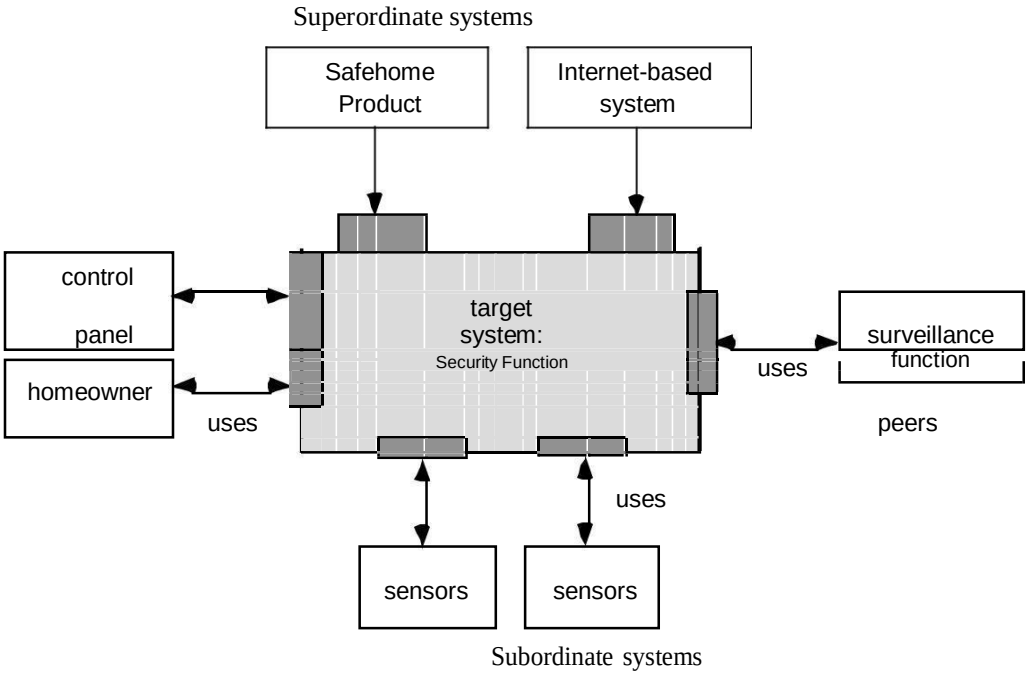
How do functional components interact with data components?

Are data components *passive* or *active* (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

4) ARCHITECTURAL DESIGN:

Representing the System in Context:

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in the figure



Superordinate systems – those systems that use the target system as part of some higher level processing scheme.

Subordinate systems - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems - those systems that interact on a peer-to-peer basis

Actors -those entities that interact with the target system by producing or consuming information that is necessary for requisite processing

Defining Archetypes:

An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system. In general, a relative small set of archetypes is required to design even relatively complex systems.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the analysis model. In safe home security function, the following are the archetypes:

Node: Represent a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors, and (2) a variety of alarm indicators.

Detector: An abstraction that encompasses all sensing equipment that feeds information into the target system

Indicator: An abstraction that represents all mechanisms for indication that an alarm condition is occurring.

Controller: An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

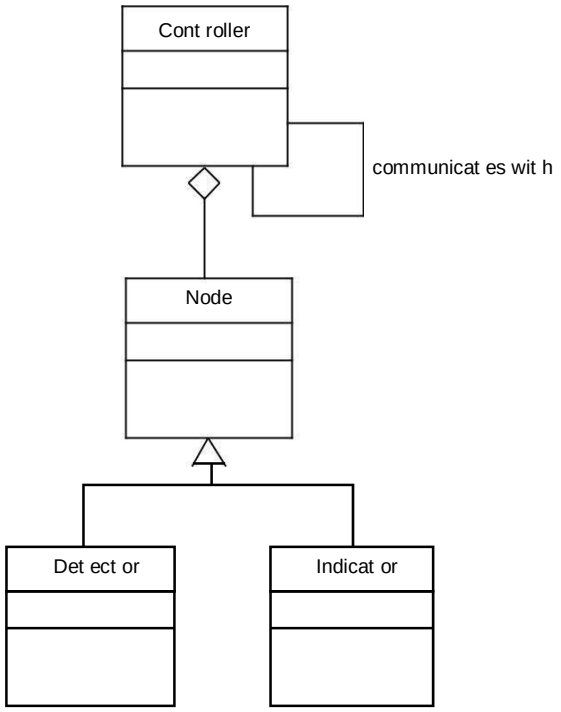


Figure 10.7 UML relat ionships f or Saf eHome securit y f unct ion archet ypes (adapt ed f rom [BOS00])

Refining the Architecture into Components:

As the architecture is refined into components, the structure of the system begins to emerge. The architectural designer begins with the classes that were described as part of the analysis model. These analysis classes represent entities within the application domain that must be addressed within the software architecture. Hence, the application domain is one source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application domain.

For eg: memory management components, communication components database components, and task management components are often integrated into the software architecture.

In the *safeHome* security function example, we might define the set of top-level components that address the following functionality:

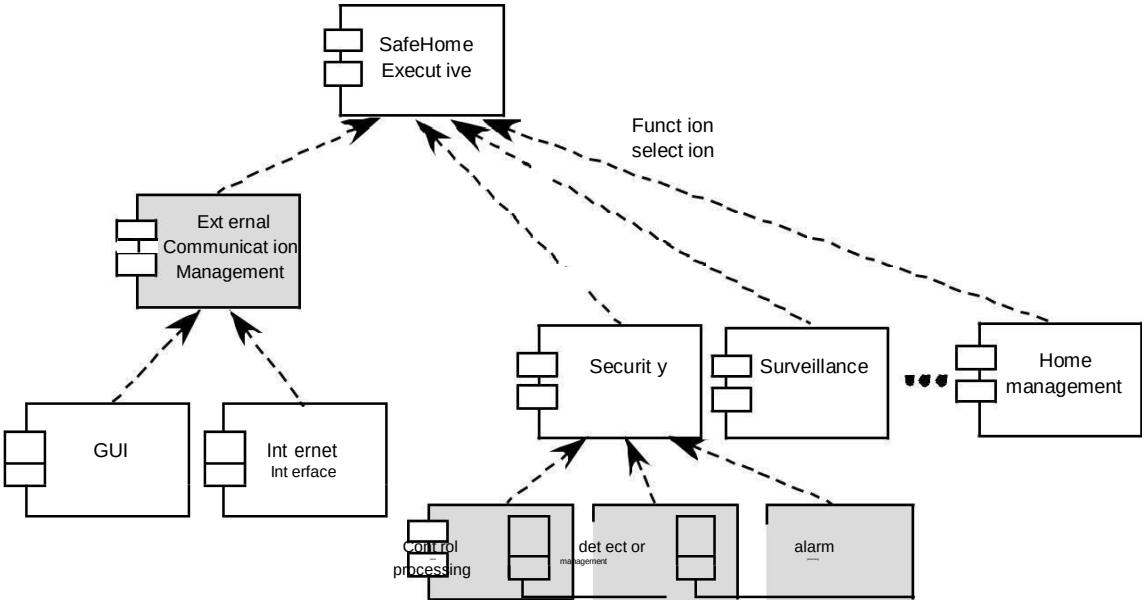
External communication management- coordinates communication of the security function with external entities

Control panel processing- manages all control panel functionality.

Detector management- coordinates access to all detectors attached to the system.

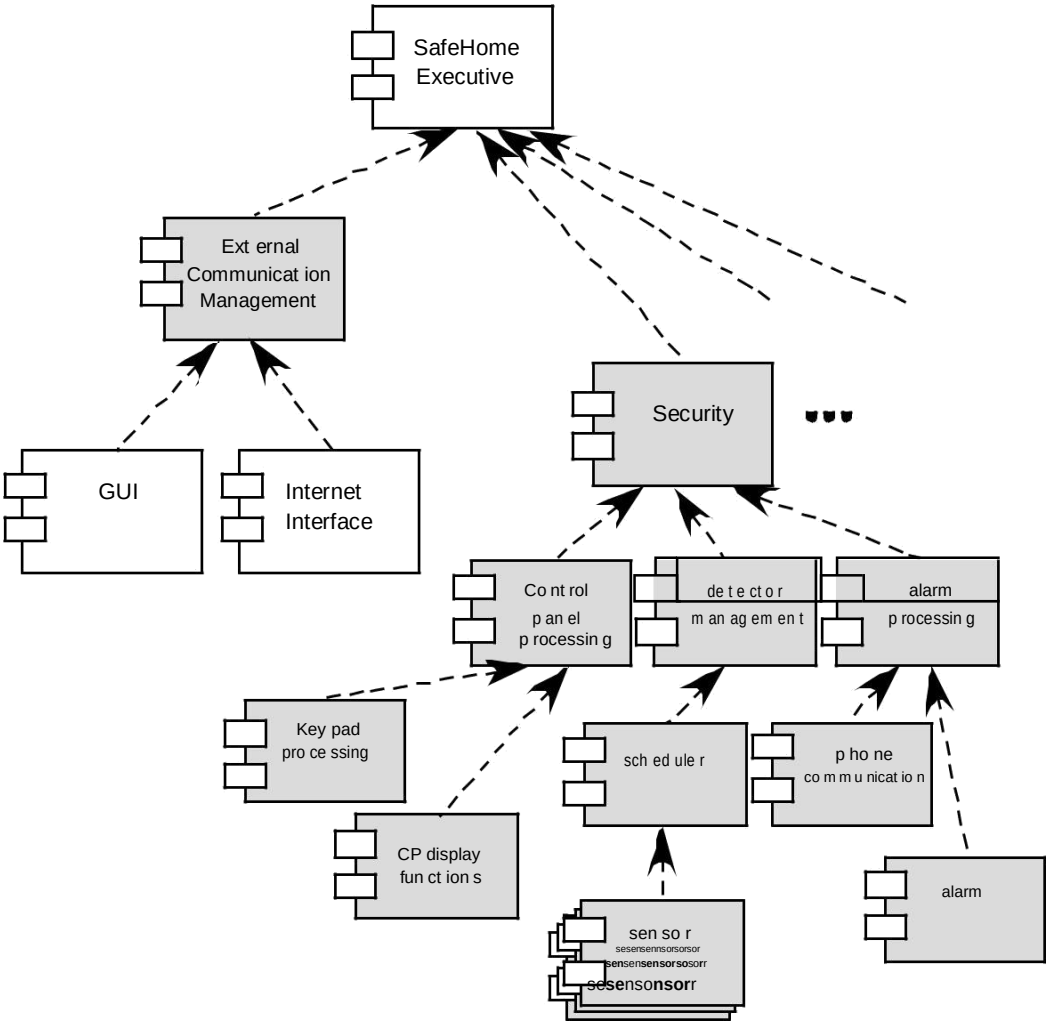
Alarm processing- verifies and acts on all alarm conditions.

Design classes would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design.



Component Structure

IV Describing Instantiations of the System: An actual instantiation of the architecture means the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.



Object And Object Classes

Object : An object is an entity that has a state and a defined set of operations that operate on that state.

An obect class defination is both a type specification and a template for creating obects.

It includes declaration of all the attributes and operations that are associated with object of that class.

Object Oriented Design Process

There are five stages of object oriented design process

1)Understand and define the context and the modes of use of the system.

2)Design the system architecture

3) Identify the principle objects in the system.

4)Develop a design models

5)Specify the object interfaces

Systems context and modes of use

It specify the context of the system.it also specify the relationships between the software that is being designed and its external environment.

If the system context is a static model it describe the other system in that environment.

If the system context is a dynamic model then it describe how the system actually interact with the environment.

System Architecture

Once the interaction between the software system that being designed and the system environment have been defined

We can use the above information as basis for designing the System Architecture.

Object Identification

This process is actually concerned with identifying the object classes.

We can identify the object classes by the following

1)Use a grammatical analysis

2)Use a tangible entities

3)Use a behaviourial approach

4)Use a scenario based approach

Design model

Design models are the bridge between the requirements and implementation.

There are two type of design models

1) Static model describe the relationship between the objects.

2)Dynamic model describe the interaction between the objects

Object Interface SpecificationIt is concerned with specifying the details of the interfaces to an objects.

Design evolution

The main advantage OOD approach is to simplify the problem of making changes to the design.

Changing the internal details of an obect is unlikely to effect any other system object.

USER INTERFACE DESIGN

Interface design focuses on three areas of concern:

the design of interfaces between software components,

the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and

the design of the interface between a human (i.e., the user) and the computer.

What is User Interface Design?

User interface design creates an effective communication medium between a human

and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Why is User Interface Design important?

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. Because it molds a user's perception of the software, the interface has to be right.

THE GOLDEN RULES

Theo Mandel coins three "golden rules":

Place the user in control.

Reduce the user's memory load.

Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important software design activity.

Place the User in Control

Mandel [MAN97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. Word processor – spell checking – move to edit and back; enter and exit with little or no effort

Provide for flexible interaction. Several modes of interaction – keyboard, mouse, digitizer pen or voice recognition, but not every action is amenable to every interaction need. Difficult to draw a circle using keyboard commands.

Allow user interaction to be interruptible and undoable. User stop and do something and then resume where left off. Be able to undo any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Perform same actions repeatedly; have macro mechanism so user can customize interface.

Hide technical internals from the casual user. Never required to use OS commands; file management functions or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. User has feel of control when interact directly with objects; stretch an object.

Reduce the User's Memory Load:

The more a user has to remember, the more error-prone interaction with the system will be.

Good interface design does not tax the user's memory

System should remember pertinent details and assist the user with interaction scenario that assists user recall.

Mandel defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. Complex tasks can put a significant burden on short term memory. System designed to reduce the requirement to remember past actions and results; visual cues to recognize past actions, rather than recall them.

Establish meaningful defaults. Initial defaults for average user; but specify individual preferences with a reset option.
Define shortcuts that are intuitive. Use mnemonics like Alt-P.

The visual layout of the interface should be based on a real world metaphor. Bill payment – check book and check register metaphor to guide a user through the bill paying process; user has less to memorize

Disclose information in a progressive fashion. Organize hierarchically. High level of abstraction and then elaborate. Word underlining function – number of functions, but not all listed. User picks underlining then all options presented

Make the Interface Consistent

Interface present and acquire information in a consistent fashion.

All visual information is organized to a design standard for all screen displays

Input mechanisms are constrained to limited set used consistently throughout the application

Mechanisms for navigation from task to task are consistently defined and implemented

Mandel [MAN97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Because of many screens and heavy interaction, it is important to provide indicators – window tiles, graphical icons, consistent color coding so that the user knows the context of the work at hand; where came from and alternatives of where to go.

Maintain consistency across a family of applications. For applications or products implementation should use the same design rules so that consistency is maintained for all interaction

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Unless a compelling reason presents itself don't change interactive sequences that have become de facto standards. (alt-S to scaling)

USER INTERFACE DESIGN

Interface Design Models

Four different models come into play when a user interface is to be designed.

The software engineer creates a *design model,*

a human engineer (or the software engineer) establishes a *user model,*

the end-user develops a mental image that is often called the *user's model* or the *system perception,* and

the implementers of the system create a *implementation model.*

The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

User Model: The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [SHN90]. In addition, users can be categorized as

Novices.

Knowledgeable, intermittent users.

Knowledgeable, frequent users.

Design Model: A design model of the entire system incorporates data, architectural, interface and procedural representations of the software.

Mental Model: The user's mental model (system perception) is the image of the system that end-users carry in their heads.

Implementation Model: The implementation model combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system syntax and semantics.

These models enable the interface designer to satisfy a key element of the most important principle of user interface design: **"Know the user, know the tasks."**

The User Interface Design Process: (steps in interface design)

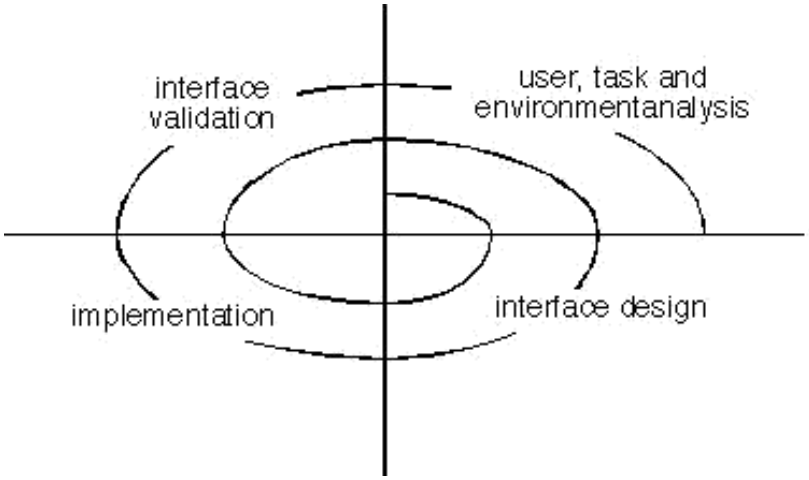
The user interface design process encompasses four distinct framework activities :

User, task, and environment analysis and modeling

Interface design

Interface construction

Interface validation



User Interface Design Process

(1) User Task and Environmental Analysis:

The interface analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, the software engineer attempts to understand the system perception (Section 15.2.1) for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

Where will the interface be located physically?
Will the user be sitting, standing, or performing other tasks unrelated to theinterface?
Does the interface hardware accommodate space, light, or noise constraints?
Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

(2) Interface Design:

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

(3) Interface Construction(implementation)

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 15.5) may be used to complete the construction of the interface.

(4) Interface Validation:

Validation focuses on

the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

the degree to which the interface is easy to use and easy to learn; and

the users' acceptance of the interface as a useful tool in their work.

INTERFACE ANALYUSIS

A Key tenet of all software engineering process models is this: you better understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding (1) The people who will interact with the system through the interface; (2) the tasks that tend-users must perform to do their work, (3) the content that is presented as part of the inter face, an (4) the environment in which these tasks will be conducted. In the sections that follow, we examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

User analysis

Earlier we noted that each user has a mental image or system perception of the software that may be different from the mental image developed by other users.

User Interviews. The most direct approach, interviews involve representatives from the software team who meet with end-users to better understand their needs, motivations work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with customers an users on regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in definition of market segments while providing an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talk with users on a daily basis, making them the most likely soured of information on what works an what doesn't, what users like and what they dislike, what features generate questions, and what features are easy to use.

The following set of questions (adapted form (HAC98)) will help the interface designer better understand the users of a system:

Are user trained professionals, technicians, clerical or manufacturing workers?
What level of formal education does the average user have?
Are the users capable of learning from written materials or have they ecpressed a desire of classroom training?
Are users expert typists or keyboard phobic?
What is the age range of the user community?
Will the users be represented predominately by one gender?
How are users compensated for the work they perform?
Do users work normal office hours, or do they work until the job is done.
Is the software to be an integral part of the work users do, or will it be used only occasionally?
What is the primary spoken language among users?
What are the consequences if a user makes a mistake using the system?
Are users experts in the subject matter the is addressed by the system?
Do users want to know about the technology that sits behind the interface?

The answers to these an similar questions will allow the designer to understand who the end-users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiled, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

Task Analysis and Modeling

The goal of talk analysis is to answer the following questions:

What work will the user perform in specific circumstances?

What specific problem domain objects will the user manipulate as work is performed?

What is the sequence of work tasks-the workflow?

What is the hierarchy of tasks?

To answer these questions, the software engineer must draw upon analysis techniques discussed in Chapters 7 and 8, but in this instance, these techniques are applied to the user interface.

In earlier chapter we noted that the use-case describe the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system.

The use-case provides a basic description of one important work task for the computer-aided design system. From, it, the software engineer can extract tasks, objects, and the overall flow of the interaction.

Task elaboration. Task analysis of interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate. To understand the tasks that must be performed to accomplish the goal of the activity, a human engineer must understand the tasks that humans currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, the human engineer can study an existing specification for computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. By observing an interior designer at work, the engineer notices that interior design comprises a number of major activities: further layout (note the use-case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use-case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions; (2) place windows and doors at appropriate locations;(3a) use furniture templates to draw scaled accents on floor plan(4) move furniture outlines;(6) draw dimensions to show location;(7) draw perspective rendering view for customer. A similar approach could be used for each of the other major tasks.

Object elaboration. The software engineer extracts the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide the designer with a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include **size, shape, location** and others. The interior designer would select the object from the **Furniture** class, move it to a position on the floor plan (another object in this context), draw the furniture outline, and so forth. He tasks select, move, and draw are operations. The user interface analysis model would not provide a literal implementation for each of these operation for each of these operations. How ever, as the design is elaborated, the details of each operation are defined.

Workflow analysis. When a number of different users, each playing different roles, makes uses of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows a software engineer to understand how a work process is completed when several people are involved.

The flow of events (shown in the figure) enable the interface designer to recognize three day interface characteristics.

Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different form the one defined for pharmacists or physicians.

The interface design for pharmacists and physicians must accommodate access to and display of information form secondary information sources(e.g., access to inventory of the pharmacist and access to information about alternative medications for the physician)

Many of the activities noted in the swimlane diagram can be further elaborated using talk analysis and /or object elaboration(e.g., fills prescription could imply a mail-order deliver, a visit to a pharmacy, or a visit to a special drug distribution center.

Hierarchical representation. As the interface is analyzed, a process of elaboration occurs. Once workflow has been established, a task hierarchy can e defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the user task requests that a prescription be refilled. The following task hierarchy is developed:

Request that a prescription be refilled

Provide identifying information
Specify name
Specify userid
Specify PIN and password
Specify prescription number
Specify date refill is required

To complete the request that a prescription be refilled tasks, three subtasks are defined. One of these subtasks, provide indentifying information, is further elaborated in three additional sub-subtasks.

Analysis of Display Content

System response time is measured from the point at which the user performs some control action(e.g., hits the return key or clicks a mouse)until the software responds with the desired output or action.

System response time has two important characteristics: length and variability. If system response is is too long, user frustration and stress is the inevitable result. Variability refers to the deviation form average response time, and, in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "defferent"has occurred behind the scenes.

Help facilities. Modern software provides on-line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered:

Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.

How will the user request help? Options include a help menu, a special function day, or a HELP command.

How will help be represented? Options include a separate window, a reference to a printed document, or a one-or two-line suggestion produced in a fixed screen location.

How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy or information that provides increasing detail as the user proceeds into the structure, or the user of hypertext.

In general, every error message or warning produced by an interactive system should have the following characteristics:

The message should describe the problem in language the user can understand.

The message should provide constructive advice for recovering form the error.

The message should indicate any negative consequences of the error(e.g., potentially corrupted data files)so that the user can check to ensure that they have not occurred.

The message should be nonjudgmental. That is, the wording should never place blame on the user.

But an-effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

A number of design issues arise when typed commands or menu labels are provided as mode of interaction:

Will every menu option have a corresponding command?

What form will commands take? Options include a control sequence (e.g., alt-p), function keys, or a typed word.

How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

Can commands be customized or abbreviated by the user?

Are menu labels self-explanatory within the context of the interface?

Are submenus consistent with the function implied by a master menu item?

Application accessibility .Accessibility for users and software engineers) who may be physically challenged is an imperative for moral, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software-provide detailed suggestions for designing interfaces that achieve vary8ing levels of accessibility. Others provide specific guidelines or "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. The challenge should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. Localization features enable the interface to be customized for a specific market.

A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues and discrete implementation issues. The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundred of characters and symbols.

12.5 DESIGN EVALUATION

After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used e.g., questionnaires, rating sheets), the designer may extract information form these data (e.g., 80percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by user of the system.

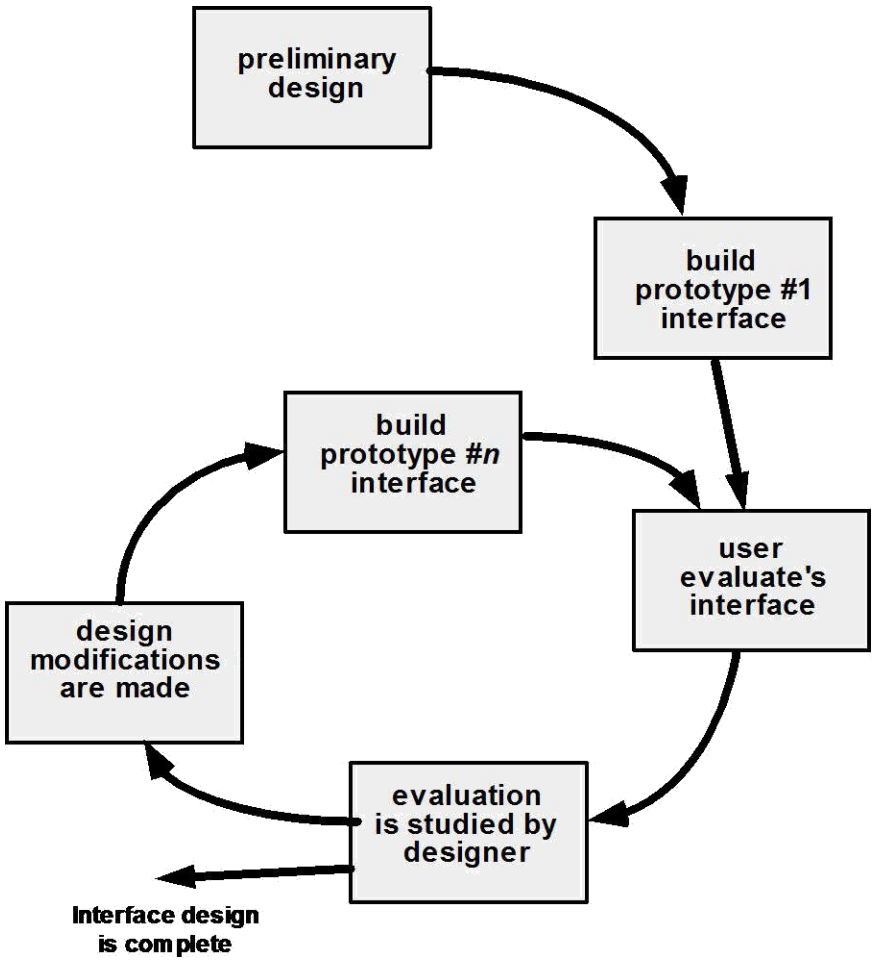
The number of user tasks specified and the average number of actions per task provide an indication on interaction time and the overall efficiency of the system.

The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

Interface styles, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect 2qualitaive data, questionnaires can be distributed to users of the prototype. Questions can be (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response,(4) Likert scales(e.g., strongly.

Users are observed during interaction, and data-such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period-are collected and used as a guide for interface modification.


preliminary design
build prototype #1 interface
build prototype #n interface
user evaluate's interface
design modifications are made
evaluation is studied by designer
Interface design is complete

UNIT-4

A strategic Approach for Software testing

Software Testing

One of the important phases of software development

Testing is the process of execution of a program with the intention of finding errors

Involves 40% of total project cost

Testing Strategy

A road map that incorporates test planning, test case design, test execution and resultant data collection and execution

Validation refers to a different set of activities that ensures that the software is traceable to the customer requirements.

V&V encompasses a wide array of Software Quality Assurance

Perform Formal Technical reviews(FTR) to uncover errors during software development

Begin testing at component level and move outward to integration of entire component based system.

Adopt testing techniques relevant to stages of testing

Testing can be done by software developer and independent testing group

Testing and debugging are different activities. Debugging follows testing

Low level tests verifies small code segments.

High level tests validate major system functions against customer requirements

Testing Strategies for Conventional Software

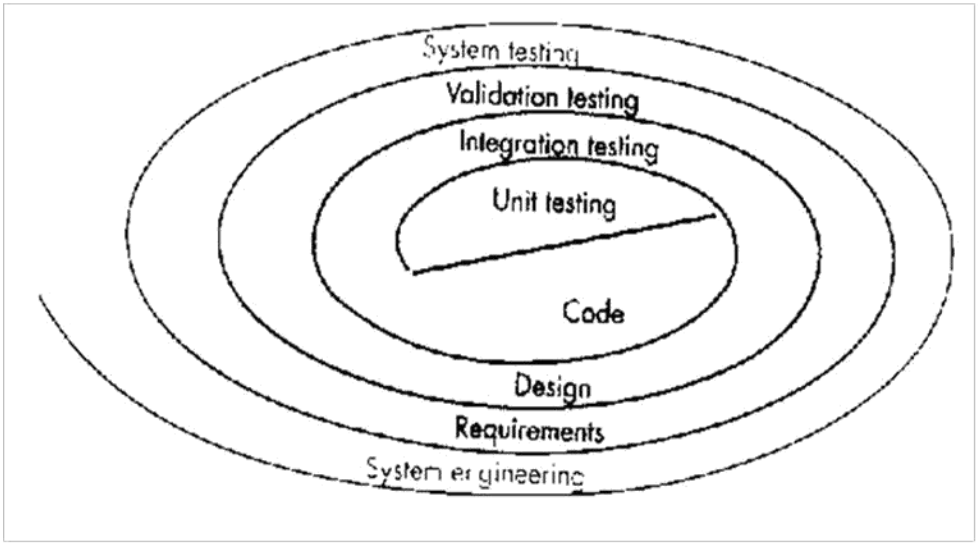
1)Unit Testing

2) Integration Testing

3)Validation Testing and

4)System Testing

Spiral Representation for Conventional Software



Criteria for completion of software testing

No body is absolutely certain that software will not fail

Based on statistical modeling and software reliability models

95 percent confidence(probability) that 1000 CPU hours of failure free operation is at least 0.995

Software Testing

Two major categories of software testing
Black box testing

White box testing
Black box testing

Treats the system as black box whose behavior can be determined by studying its input and related output

Not concerned with the internal structure of the program

Black Box Testing

It focuses on the functional requirements of the software ie it enables the sw engineer to derive a set of input conditions that fully exercise all the functional requirements for that program.

Concerned with functionality and implementation

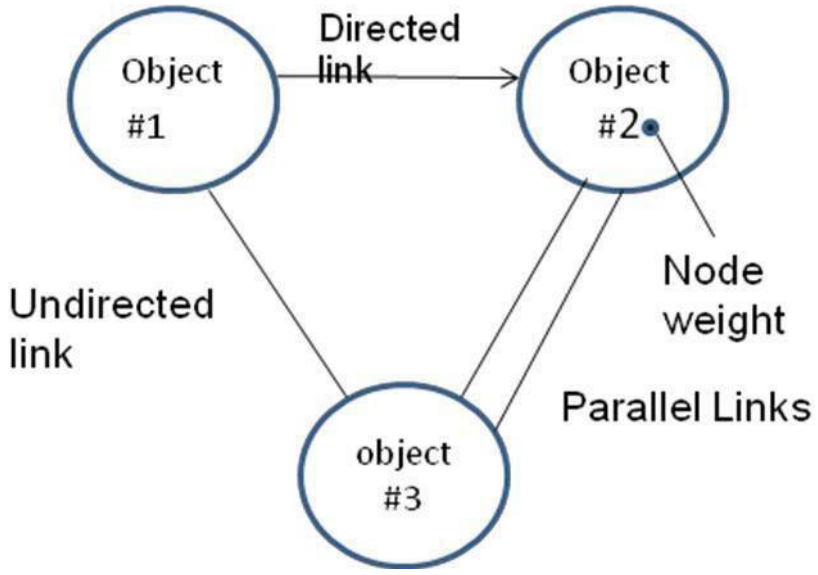
1)Graph based testing method

2) Equivalence partitioning

Graph based testing

Draw a graph of objects and relations

Devise test cases t uncover the graph such that each object and its relationship exercised.



Equivalence partitioning

Divides all possible inputs into classes such that there are a finite equivalence classes.

Equivalence class

Set of objects that can be linked by relationship

Reduces the cost of testing

Example

Input consists of 1 to 10

Then classes are n<1,1<=n<=10,n>10

Choose one valid class with value within the allowed range and two invalid classes where values are greater than maximum value and smaller than minimum value.

Boundary Value analysis

- Select input from equivalence classes such that the input lies at the edge of the equivalence classes

Set of data lies on the edge or boundary of a class of input data or generates the data that lies at the boundary of a class of output data

Example

If 0.0<=x<=1.0

Then test cases (0.0,1.0) for valid input and (-0.1 and 1.1) for invalid input

Orthogonal array Testing

To problems in which input domain is relatively small but too large for exhaustive testing

Example

Three inputs A,B,C each having three values will require 27 test cases

L9 orthogonal testing will reduce the number of test case to 9 as shown below

A	B	C
1	1	1
1	2	2
1	3	3
2	1	3
2	2	3
2	3	1
3	1	3
3	2	1

3 3 2

White Box testing

Also called glass box testing

Involves knowing the internal working of a program

Guarantees that all independent paths will be exercised at least once.

Exercises all logical decisions on their true and false sides

Executes all loops

Exercises all data structures for their validity

White box testing techniques

Basis path testing

Control structure testing

Basis path testing

Proposed by Tom McCabe

Defines a basic set of execution paths based on logical complexity of a procedural design

Guarantees to execute every statement in the program at least once

Steps of Basis Path Testing

Draw the flow graph from flow chart of the program

Calculate the cyclomatic complexity of the resultant flow graph

Prepare test cases that will force execution of each path

Three methods to compute Cyclomatic complexity number

V(G)=E-N+2(E is number of edges, N is number of nodes

V(G)=Number of regions

V(G)= Number of predicates +1

Control Structure testing

Basis path testing is simple and effective

It is not sufficient in itself

Control structure broadens the basic test coverage and improves the quality of white box testing

Condition Testing

Data flow Testing

Loop Testing

- Condition Testing

  --Exercise the logical conditions contained in a program module

  --Focuses on testing each condition in the program to ensure that it does contain errors

  --Simple condition

  E1<relation operator>E2

  --Compound condition

  simple condition<Boolean operator>simple condition

-

-

Data flow Testing

Selects test paths according to the locations of definitions and use of variables in a program

Aims to ensure that the definitions of variables and subsequent use is tested

First construct a definition-use graph from the control flow of a program

Loop Testing

Focuses on the validity of loop constructs

Four categories can be defined

Simple loops

Nested loops

Concatenated loops

Unstructured loops

Testing of simple loops

N is the maximum number of allowable passes through the loop

Skip the loop entirely

Only one pass through the loop

Two passes through the loop

m passes through the loop where m>N

N-1,N,N+1 passes the loop

Nested Loops

Start at the innermost loop. Set all other loops to maximum values

Conduct simple loop test for the innermost loop while holding the outer loops at their minimum iteration parameter.

Work outward conducting tests for the next loop but keeping all other loops at minimum.

Follow the approach defined for simple loops, if each of the loop is independent of other.

If the loops are not independent, then follow the approach for the nested loops

Unstructured Loops

Redesign the program to avoid unstructured loops

Validation Testing

It succeeds when the software functions in a manner that can be reasonably expected by the customer.

1) Validation Test Criteria

2)Configuration Review

3)Alpha And Beta Testing

System Testing

Its primary purpose is to test the complete software.

1)Recovery Testing

2) Security Testing

3Stress Testing and

4)Performance Testing

The Art of Debugging

Debugging occurs as a consequences of successful testing.

Debugging Stratergies

1) Brute Force Method.

2) Back Tracking

3) Cause Elimination and

4)Automated debugging

Brute force

Most common and least efficient

Applied when all else fails

Memory dumps are taken

Tries to find the cause from the load of information

Back tracking

Common debugging approach

Useful for small programs

Beginning at the system where the symptom has been uncovered, the source code traced backward until the site of the cause is found.

Cause Elimination

Based on the concept of Binary partitioning

A list of all possible causes is developed and tests are conducted to eliminate each

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

Factors that affect software quality can be categorized in two broad groups:

Factors that can be directly measured (e.g. defects uncovered during testing)

Factors that can be measured only indirectly (e.g. usability or maintainability)

McCall's quality factors

Product operation

Correctness

Reliability

Efficiency

Integrity

Usability

Product Revision

Maintainability

Flexibility

Testability

Product Transition

Portability

Reusability

Interoperability

ISO 9126 Quality Factors

Functionality

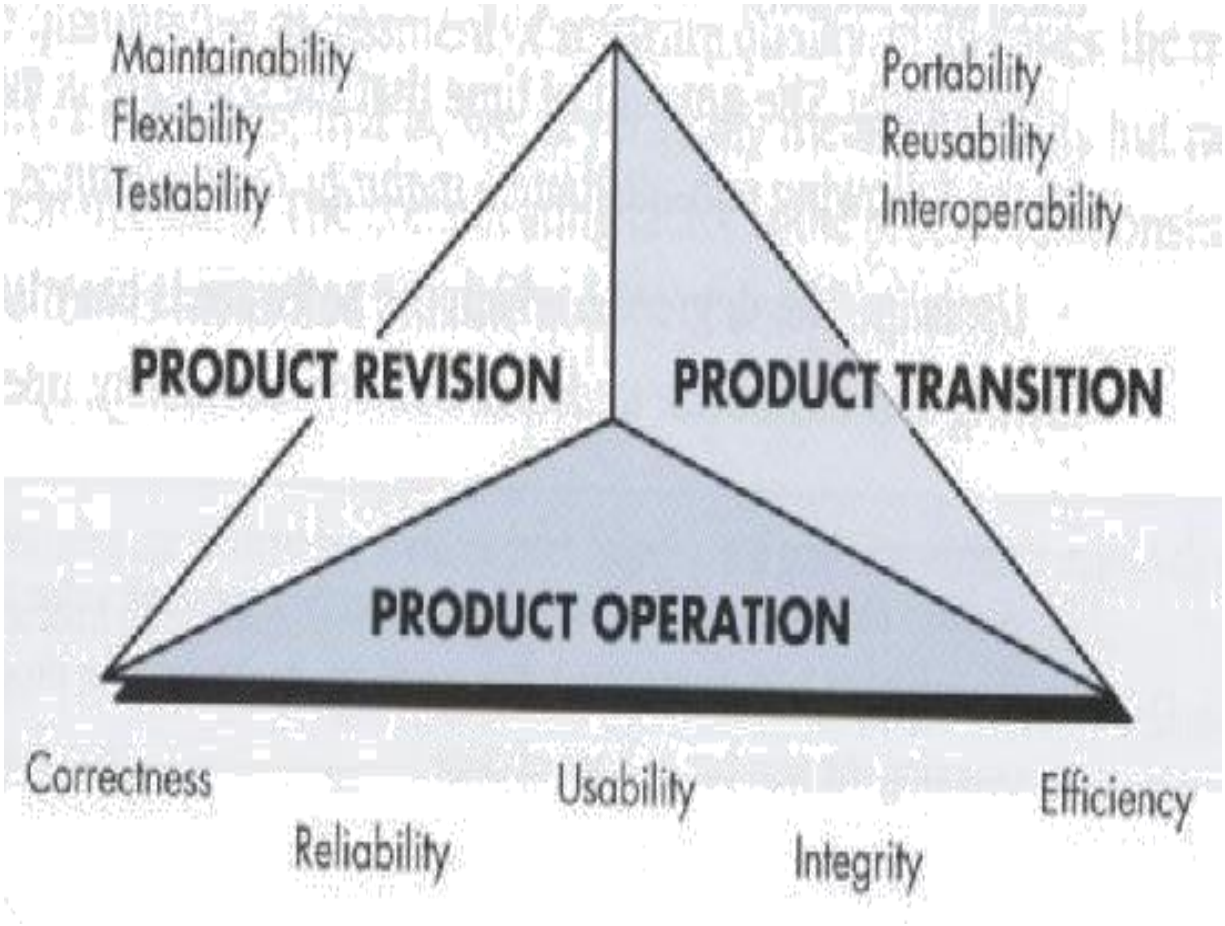
Reliability

Usability

Efficiency

Maintainability

Portability



Product metrics

Product metrics for computer software helps us to assess quality.

Measure

Provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process

Metric(IEEE 93 definition)

A quantitative measure of the degree to which a system, component or process possess a given attribute

Indicator

A metric or a combination of metrics that provide insight into the software process, a software project or a product itself

Product Metrics for analysis,Design,Test and maintenance

Product metrics for the Analysis model
Function point Metric
First proposed by Albrecht

Measures the functionality delivered by the system

FP computed from the following parameters

Number of external inputs(EIS)

Number external outputs(EOS)

Number of external Inquiries(EQS)

Number of Internal Logical Files(ILF)

Number of external interface files(EIFS)

Each parameter is classified as simple, average or complex and weights are assigned as follows

•Information Domain	Count	Simple	avg	Complex
EIS	3	4	6	
EOS	4	5	7	
EQS	3	4	6	
ILFS	7	10	15	
EIFS	5	7	10	

FP=Count total *[0.65+0.01*E(Fi)]

Metrics for Design Model

DSQI(Design Structure Quality Index)

US air force has designed the DSQI

Compute s1 to s7 from data and architectural design

S1:Total number of modules

S2:Number of modules whose correct function depends on the data input

S3:Number of modules whose function depends on prior processing

S4:Number of data base items

S5:Number of unique database items

S6: Number of database segments

S7:Number of modules with single entry and exit

Calculate D1 to D6 from s1 to s7 as follows:

D1=1 if standard design is followed otherwise D1=0

D2(module independence)=(1-(s2/s1))

D3(module not depending on prior processing)=(1-(s3/s1))

D4(Data base size)=(1-(s5/s4))

D5(Database compartmentalization)=(1-(s6/s4)

D6(Module entry/exit characteristics)=(1-(s7/s1))

DSQI=sigma of WiDi

i=1 to 6,Wi is weight assigned to Di

If sigma of wi is 1 then all weights are equal to 0.167

DSQI of present design be compared with past DSQI. If DSQI is significantly lower than the average, further design work and review are indicated

METRIC FOR SOURCE CODE

HSS(Halstead Software science)

Primitive measure that may be derived after the code is generated or estimated once design is complete

n_1 = the number of distinct operators that appear in a program

n_2 = the number of distinct operands that appear in a program

N_1 = the total number of operator occurrences.

N_2 = the total number of operand occurrence.

Overall program length N can be computed:

$N = n_1 \log2\ n_1 + n_2 \log2\ n_2$

$V = N \log_2 (n_1 + n_2)$

METRIC FOR TESTING

n_1 = the number of distinct operators that appear in a program

n_2 = the number of distinct operands that appear in a program

N_1 = the total number of operator occurrences.

N_2 = the total number of operand occurrence.

Program Level and Effort

$PL = 1/[(n_1 / 2) \times (N_2 / n_2\ l)]$

$e = V/PL$

•

METRICS FOR MAINTENANCE

M_t = the number of modules in the current release

F_c = the number of modules in the current release that have been changed

F_a = the number of modules in the current release that have been added.

F_d = the number of modules from the preceding release that were deleted in the current release

The Software Maturity Index, SMI, is defined as:

$SMI = [M_t - (F_c + F_a + F_{d)/} M_t]$

METRICS FOR PROCESS AND PROJECTS

SOFTWARE MEASUREMENT

Software measurement can be categorized in two ways.

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced.

To develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

Errors per KLOC (thousand lines of code).

Defects per KLOC.

$ per LOC.

Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

Errors per person-month.

LOC per person-month.

$ per page of documentation.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the ***function point***. **Function points** are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Proponents claim that FP is programming language independent, making it ideal for application using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

Opponents claim that the method requires some "sleight of hand " in that computation is basedsubjective rather than objective data, that counts of the information domain can be difficult to collect after the fact, and that FP has no direct physical meaning- it's just a number.

Typical Function-Oriented Metrics:

errors per FP (thousand lines of code)

defects per FP

$ per FP

pages of documentation per FP

FP per person-month

Reconciling Different Metrics Approaches

The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost.

Object Oriented Metrics:

Conventional software project metrics (LOC or FP) can be used to estimate object oriented software projects. Lorenz and Kidd suggest the following set of metrics for OO projects:

Number of scenario scripts: A scenario script is a detailed sequence of steps that describes the interaction between the user and the application.

Number of key classes: Key classes are the "highly independent components that are defined early in object-oriented analysis.

Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain.

Average number of support classes per key class: Of the average number of support classes per key class were known for a given problem domain estimation would be much simplified. Lorenz

and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes.

Number of subsystems: A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in ehic work on subsystems is partitioned among project staff.

Use-Case Oriented Metrics

Use-cases describe user-visible functions and features that are basic requirements for a system. The use-cases is directly proportional to the size of the application in LOC and to the number of use-cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use-cases can be created at vastly different levels of abstraction, there is no standard size for a use-case. Without a standard measure of what a use-case is, its application as a normalization measure is suspect.

Web Engineering Project Metrics

The objective of all web engineering projects is to build a Web application that delivers a combination of content and functionality to the end-user.

Number of static Web pages: These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measures provides an indication of the overall size of the application and the effort required to develop it.

Number of dynamic Web pages: : Web pages with dynamic content are essential in all e-commerce applications, search engines, financial application, and many other Web App categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

Number of internal page link: Internal page links are pointers that provide an indication of the degree of architectural coupling within the Web App.

Number of persistent data objects: As the number of persistent data objects grows, the complexity of the Web App also grows, and effort to implement it increases proportionally.

Number of external systems interfaced: As the requirement for interfacing grows, system complexity and development effort also increase.

Number of static content objects: Static content objects encompass static text- based, graphical, video, animation, and audio information that are incorporated within the Web App.

Number of dynamic content objects: Dynamic content objects are generated based on end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the Web App.

Number of executable functions: An executable function provides some computational service to the end-user. As the number of executable functions increases, modeling and construction effort also increase.

2) METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

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

Maintainability. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is ***mean-time-tochange*** (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

Integrity. Attacks can be made on all three components of software: programs, data, and documents.

To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

$$\text{integrity} = \sum [1 - (\text{threat} \times (1 - \text{security}))]$$

Usability: Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = E/(E + D)$$

where E is the number of errors found before delivery of the software to the end-user and

D is the number of defects found after delivery.

Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

$DRE_i = E_i/(E_i+E_{i+1})$

E_i is the number of errors found during software engineering activity *i* and

E_{i+1} is the number of errors found during software engineering activity *i+1* that are traceable to errors that were not discovered in software engineering activity *i*.

A quality objective for a software team (or an individual software engineer) is to achieve DRE

that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

UNIT-5

RISK MANAGEMENT

1) REACTIVE VS. PROACTIVE RISK STRATEGIES

At best, a **reactive strategy** monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode.*

project team reacts to risks when they occur

mitigation—plan for additional resources in anticipation of fire fighting

fix on failure—resource are found and applied when the risk strikes

crisis management—failure does not respond to applied resources and project is in jeopardy

A **proactive strategy** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk.

formal risk analysis is performed

organization corrects the root causes of risk

examining risk sources that lie beyond the bounds of the software

developing the skill to manage change

Risk Management Paradigm

2) SOFTWARE RISK

Risk always involves two characteristics

Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks
Loss—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty in the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are

Building a excellent product or system that no one really wants (market risk),

Building a product that no longer fits into the overall business strategy for the company (strategic risk),

Building a product that the sales force doesn't understand how to sell,

Losing the support of senior management due to a change in focus or a change in people (management risk), and

Losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources.

Predictable risks are extrapolated from past project experience.

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

3) RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan. There are two distinct types of risks.

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 the technology, the people, and the environment that is specific to the project that is to be built.

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.

Customer characteristics—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

Process definition—risks associated with the degree to which the software process has been defined and is followed 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.

Assessing Overall Project Risk

The questions are ordered by their relate importance to the success of a project.

Have top software and customer managers 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 their customers?

Have customers been involved fully in the definition of requirements?

Do end-users have realistic expectations?

Is 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 constituencies agree on the importance of the project and on the requirements for the system/product to be built?

3.2 Risk Components and Drivers

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.

RISK PROJECTION

Risk projection, also called ***risk estimation***, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur.

The project planner, along with other managers and technical staff, performs four risk projection activities:

establish a scale that reflects the perceived likelihood of a risk,

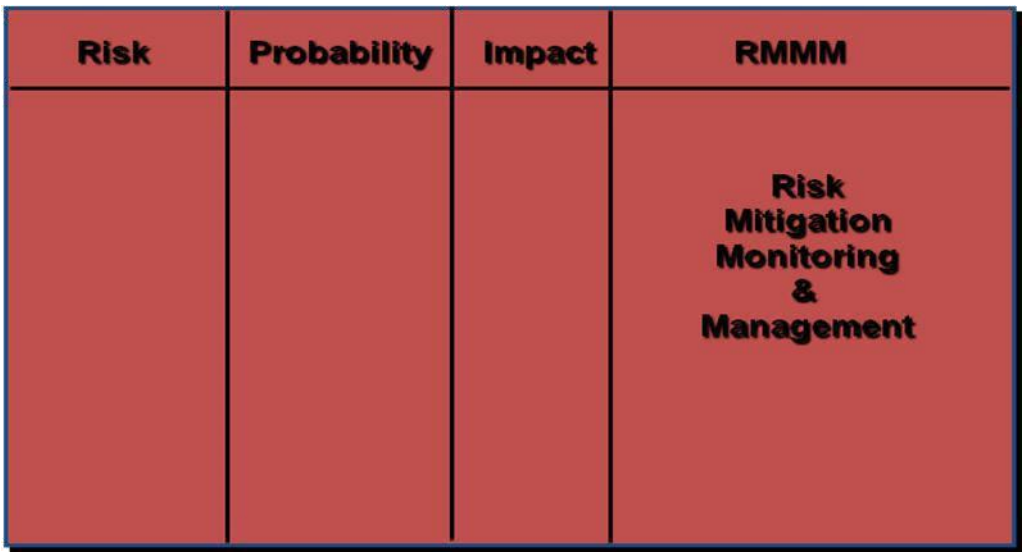
delineate the consequences of the risk,

estimate the impact of the risk on the project and the product, and

note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table

Building a Risk Table



A project team begins by listing all risks (no matter how remote) in the first column of the table.

Each risk is categorized in Next; 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.

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 project manager studies the resultant sorted table and defines a cutoff line.

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 re-evaluated to accomplish second-order prioritization.

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 ***scope*** of a risk combines the severity (just how serious is it?) with its overall distribution.

Finally, the ***timing*** of a risk considers when and for how long the impact will be felt.

The overall ***risk exposure***, RE, is determined using the following relationship

RE = *P* x *C*

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

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% (likely).

Risk impact. 60 reusable software components were planned.

Risk exposure. *RE* = 0.80 x 25,200 ~ $20,200.

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 etc.

RISK REFINEMENT

One way for risk refinement is to represent the risk in *condition-transition-consequence(CTC)* format.

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.

RISK MITIGATION, MONITORING, AND MANAGEMENT

An effective strategy must consider three issues:

Risk avoidance

Risk monitoring

Risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy.

To mitigate this risk, project management must 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 our control before the project starts.

Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

Organize project teams so that information about each development activity is widely dispersed.

Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.

Conduct peer reviews of all work (so that more than one person is "up to speed”). • Assign a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities commence. The following factors can be monitored:

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

The availability of jobs within the company and outside it.

Software safety **and** ***hazard analysis*** are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

7) THE RMMM 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.*

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.

Risk monitoring is a project tracking activity with three primary objectives:

to assess whether predicted risks do, in fact, occur;

to ensure that risk aversion steps defined for the risk are being properly applied; and

to collect information that can be used for future risk analysis.

QUALITY MANAGEMENT

QUALITY CONCEPTS:

Quality management encompasses

a quality management approach,

effective software engineering technology (methods and tools),

formal technical reviews that are applied throughout the software process,

a multitiered testing strategy,

control of software documentation and the changes made to it,

a procedure to ensure compliance with software development standards (when applicable), and

measurement and reporting mechanisms.

Variation control is the heart of quality control.

Quality

The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something."

Quality of design refers to the characteristics that designers specify for an item.

Quality of conformance is the degree to which the design specifications are followed during manufacturing.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

Robert Glass argues that a more "intuitive" relationship is in order:

User satisfaction = compliant product + good quality + delivery within budget and schedule

Quality Control

Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.

A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

Quality Assurance

Quality assurance consists of the auditing and reporting functions that assess the effectiveness and completeness of quality control activities. The **goal of quality** assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities.

Quality costs may be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include

quality planning

formal technical reviews

test equipment

training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include

in-process and interprocess inspection

equipment calibration and maintenance

testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.

Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

rework

repair

failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

complaint resolution

product return and replacement

help line support

warranty work

2) SOFTWARE QUALITY ASSURANCE

Software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

The definition serves to emphasize three important points:

Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.

Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.

A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Background Issues

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.Today, every company has mechanisms to ensure quality in its products.

During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s.

Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—

the software engineers who do technical work and

an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that conducts the following activities.

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

evaluations to be performed

audits and reviews to be performed

standards that are applicable to the project

procedures for error reporting and tracking

documents to be produced by the SQA group

amount of feedback provided to the software project team

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for

compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

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

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

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

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

3) SOFTWARE REVIEWS

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed.

Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.*

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review

A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

3.1 Cost Impact of Software Defects:

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors during the software process. However, formal review techniques have been shown to be up to 75 percent effective] in uncovering design errors. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects Assume that an error uncovered during design will cost 1.0 monetary unit to correct.

just before testing commences will cost 6.5 units;

during testing, 15 units;

and after release, between 60 and 100 units.

3.2) Defect Amplification and Removal:

(*This topic I will tell you later)*

FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are

to uncover errors in function, logic, or implementation for any representation of the software;

to verifythat the software under review meets its requirements;

to ensure that the software has been represented according to predefined standards;

to achieve software that is developed in a uniform manner; and

to make projects more manageable.

The Review Meeting

Every review meeting should abide by the following constraints:

Between three and five people (typically) should be involved in the review.

Advance preparation should occur but should require no more than two hours of work for each person.

The duration of the review meeting should be less than two hours.

The focus of the FTR is on a work product.

The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required.

The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.

Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review.

At the end of the review, all attendees of the FTR must decide whether to

accept the product without further modification,

reject the product due to severe errors (once corrected, another review must be performed), or

accept the product provisionally.

The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping

At the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A ***review summary report*** answers three questions:

What was reviewed?

Who reviewed it?

What were the findings and conclusions?

The review summary report is a single page form.

It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

Review Guidelines

The following represents a minimum set of guidelines for formal technical reviews:

Review the product, not the producer. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment.

Set an agenda and maintain it. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.

Limit debate and rebuttal. When an issue is raised by a reviewer, there may not be universal agreement on its impact.

Enunciate problem areas, but don't attempt to solve every problem noted. A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.

Take written notes. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.

Limit the number of participants and insist upon advance preparation. Keep the number of people involved to the necessary minimum.

Develop a checklist for each product that is likely to be reviewed. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.

Allocate resources and schedule time for FTRs. For reviews to be effective, they should be scheduled as a task during the software engineering process

Conduct meaningful training for all reviewers. To be effective all review participants should receive some formal training.

Review your early reviews. Debriefing can be beneficial in uncovering problems with the review process itself.

Sample-Driven Reviews (SDRs):

SDRs attempt to quantify those work products that are primary targets for full FTRs.To accomplish this the following steps are suggested...

Inspect a fraction a_i of each software work product, *i.* Record the number of faults, f_i found within a_i.

Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.

Sort the work products in descending order according to the gross estimate of the number of faults in each.

Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must

Be representative of the work product as a whole and

Large enough to be meaningful to the reviewer(s) who does the sampling.

5) STATISTICAL SOFTWARE QUALITY ASSURANCE

For software, statistical quality assurance implies the following steps:

Information about software defects is collected and categorized.

An attempt is made to trace each defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).

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").

Once the vital few causes have been identified, move to correct the problems that have caused the

For software, statistical quality assurance implies the following steps:

.

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*.

Six Sigma for software Engineering:

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. 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:

Define customer requirements and deliverables and project goals via well-defined methods of customer communication

Measure the existing process and its output to determine current quality performance (collect defect metrics)

Analyze defect metrics and determine the vital few causes.

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

Improve the process by eliminating the root causes of defects.

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

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

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

Design the process to

avoid the root causes of defects and

to meet customer requirements

Verify 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.

6) THE ISO 9000 QUALITY STANDARDS

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

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

ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines have been developed to help interpret the standard for use in the software process.
The requirements delineated by ISO 9001 address topics such as

management responsibility,

quality system, contract review,

design control,

document and data control,

product identification and traceability,

process control,

inspection and testing,

corrective and preventive action,

control of quality records,

internal quality audits,

training,

servicing and

statistical techniques.

In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.

7) SOFTWARE RELIABILITY

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

Measures of Reliability and Availability:

Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

A simple measure of reliability is *meantime-between-failure* (MTBF), where

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

Availability = [MTTF/(MTTF + MTTR)] 100%

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

7.2) Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

For example, some of the hazards associated with a computer-based cruise control for an automobile might be

causes uncontrolled acceleration that cannot be stopped

does not respond to depression of brake pedal (by turning off)

does not engage when switch is activated

slowly loses or gains speed

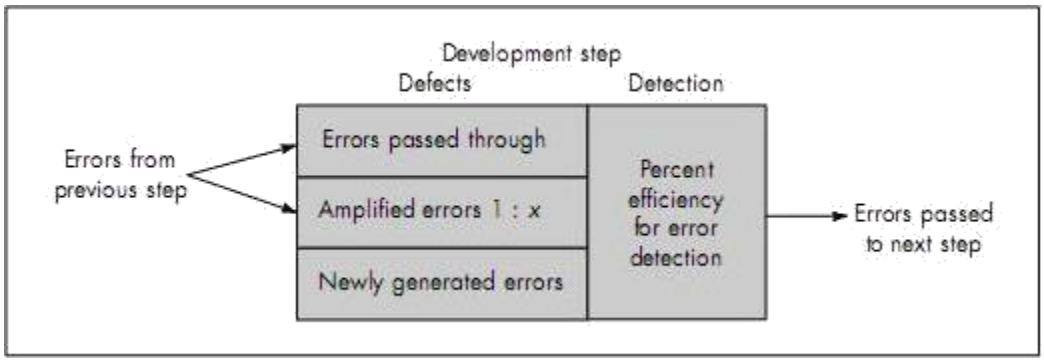
Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.To be effective, software must be analyzed in the context of the entire system.

If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as *fault tree analysis* [VES81], *real-time logic* [JAN86], or *petri net models* [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap.

Defect Amplification and Removal:



Defect Amplification Model

A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process.

A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

Referring to the figure8.3 each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field.

Figure8.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist.

Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 8.3 and 8.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).

Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units.
When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

To conduct reviews, a software engineer must expend time and effort and the development organization must spend money. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

FIGURE 8.3

Defect amplification, no reviews

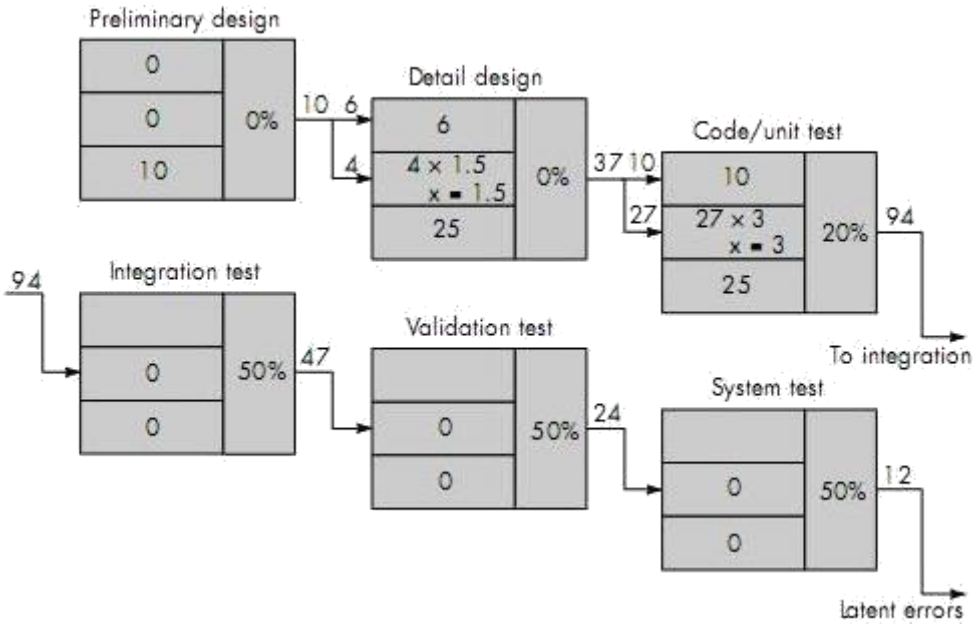


FIGURE 8.4

Defect amplification, reviews conducted