UNIT-I
INTRODUCTION TO SOFTWARE ENGINEERING

Software: Software is

(1) Instructions (computer programs) that provide desired features, function, and performance, when executed

(2) Data structures that enable the programs to adequately manipulate information,

(3) Documents that describe the operation and use of the programs.

Characteristics of Software:

(1) Software is developed or engineered; it is not manufactured in the classical sense.

(2) Software does not "wear out"

(3) Although the industry is moving toward component-based construction, most software continues to be custom built.

Software Engineering:

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

(2) 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:

1) System software
2) Application software
3) Engineering/scientific software
4) Embedded software
5) Product-line software
6) Web-applications
7) 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.

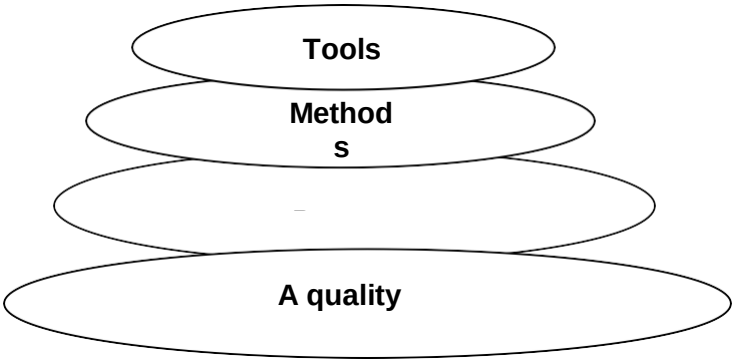
- **Product-line software:** Designed to provide a specific capability for use by many different customers, product-line software can focus on a limited and esoteric market place or address mass consumer markets

  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.

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

SOFTWARE ENGINEERING

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

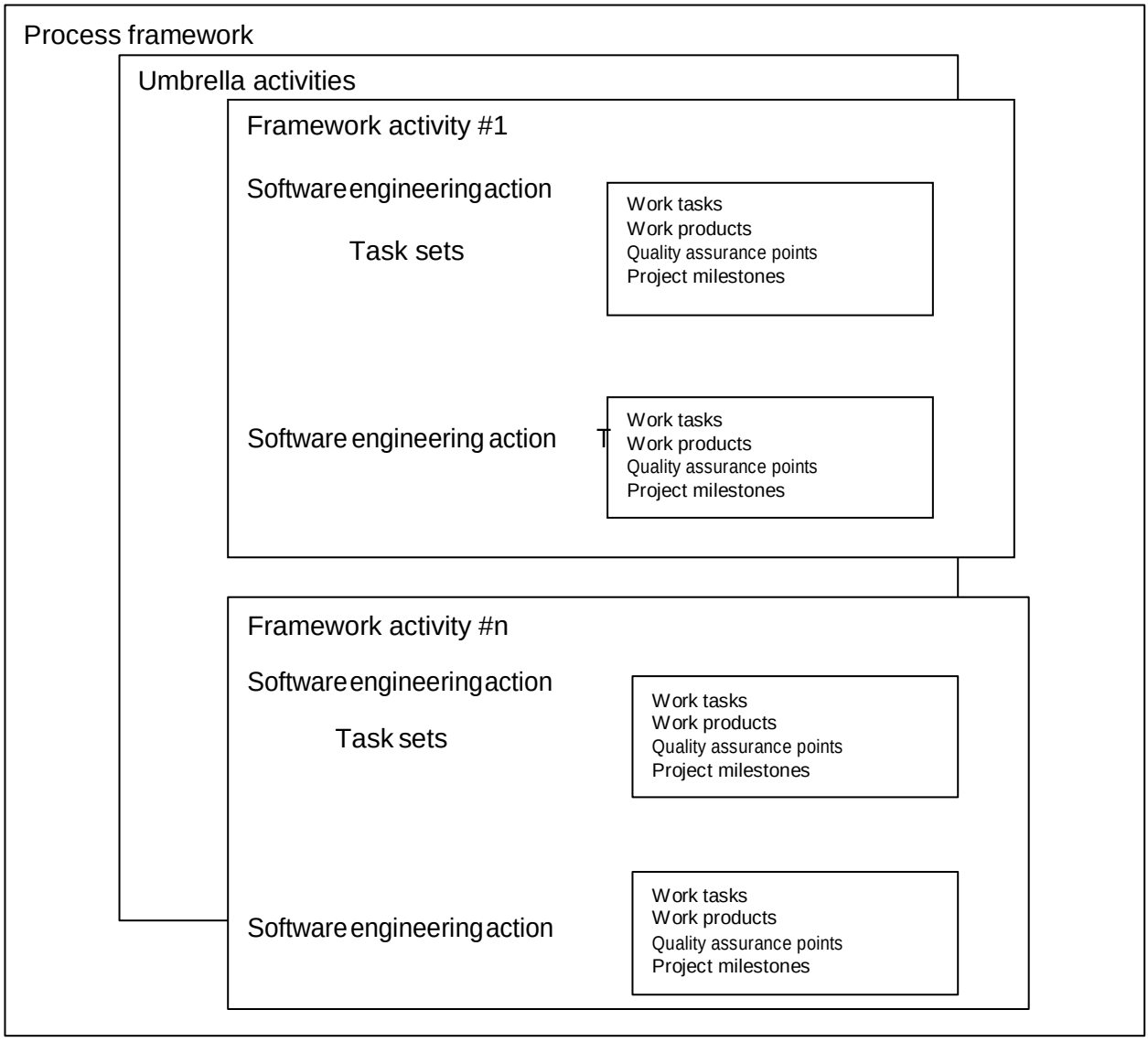
work tasks

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

1) **Communication**: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.
2) **Planning**: This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
3) **Modeling:** This activity encompasses the creation of models that allow the developer 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.
4) **Construction:** This activity combines core generation and the testing that is required to uncover the errors in the code.
5) **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.**

1) **Software project tracking and control –** allows the software team to assess progress against the project plan and take necessary action to maintain schedule.
2) **Risk Management -** assesses risks that may effect the outcome of the project or the quality of the product.
3) **Software Quality Assurance -** defines and conducts the activities required to ensure software quality.
4) **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.

5) **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.

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

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

8) **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;

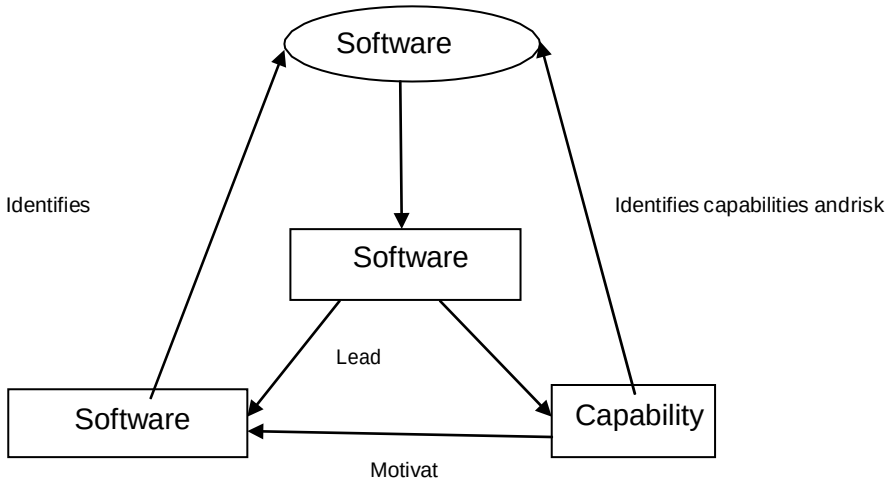
Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributes and work products, measures and other process-improvement information to the organizational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

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.

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.

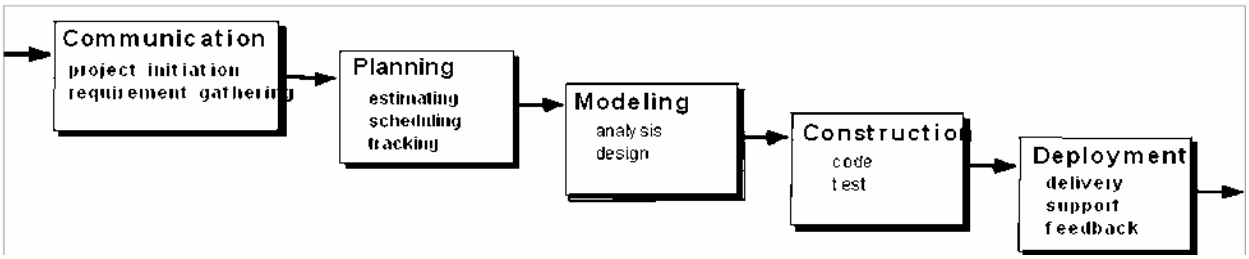
THE WATERFALL MODEL:

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:

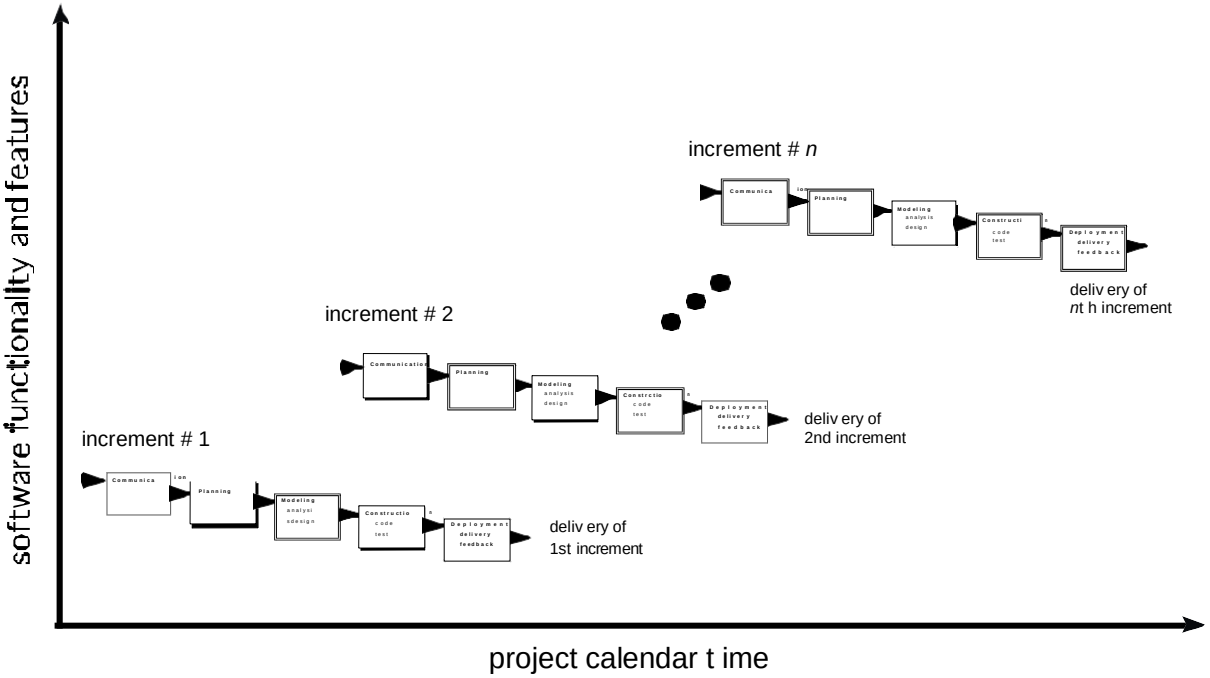
1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.
3. 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:

1) The incremental model
2) 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.

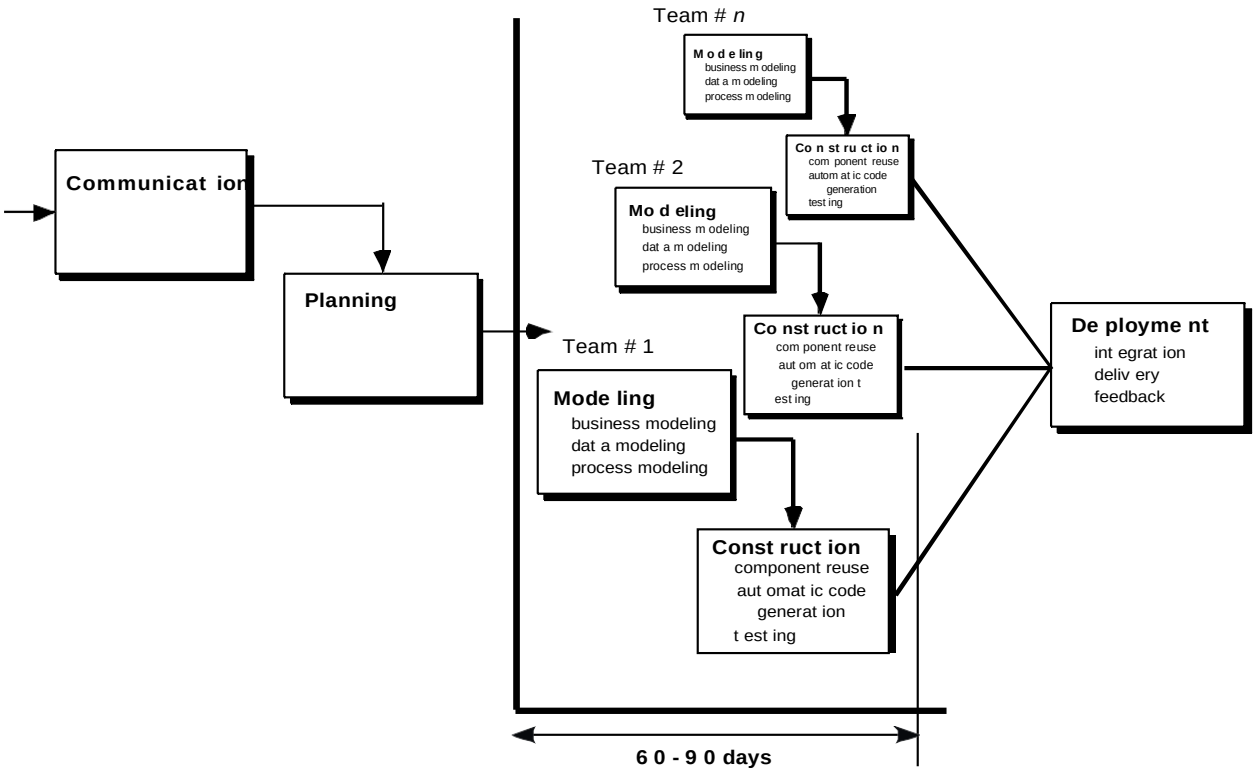
For *example*, word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking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

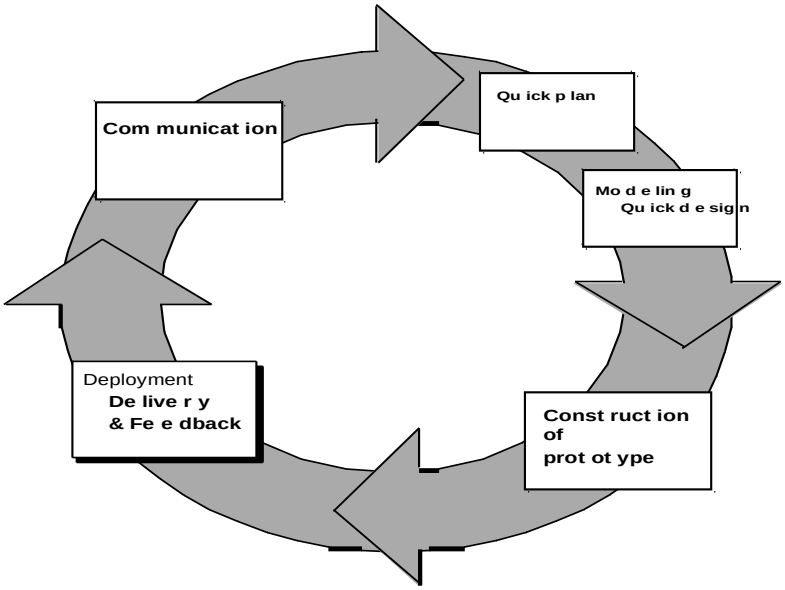
PROTOTYPING:

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:

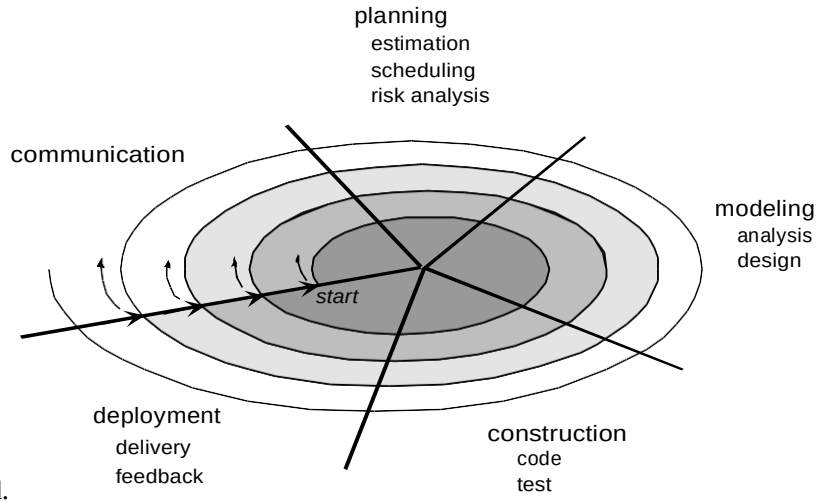
1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product 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.
2. 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 morecomplete 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.

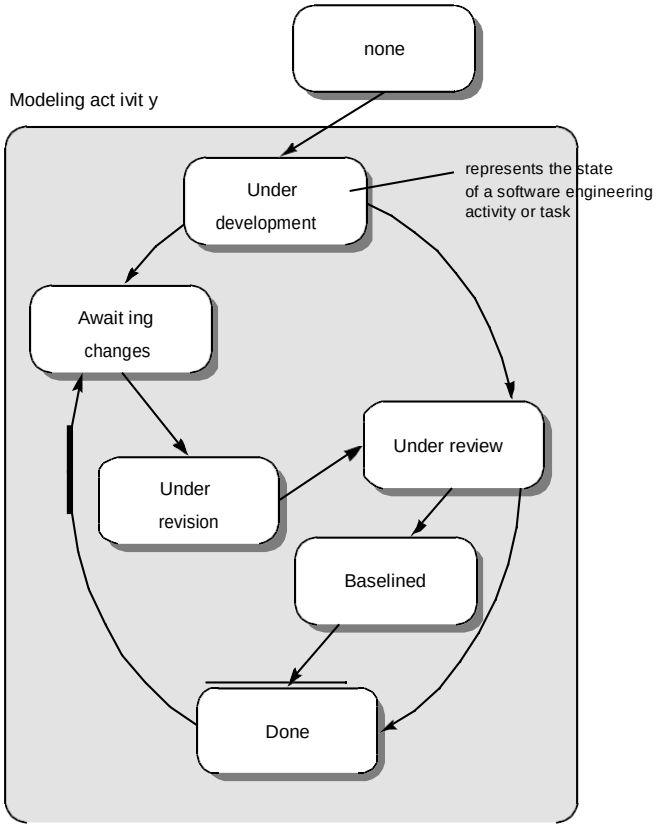
Advantages:
It provides the potential for rapid development of increasingly more complete versions of the software.

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:

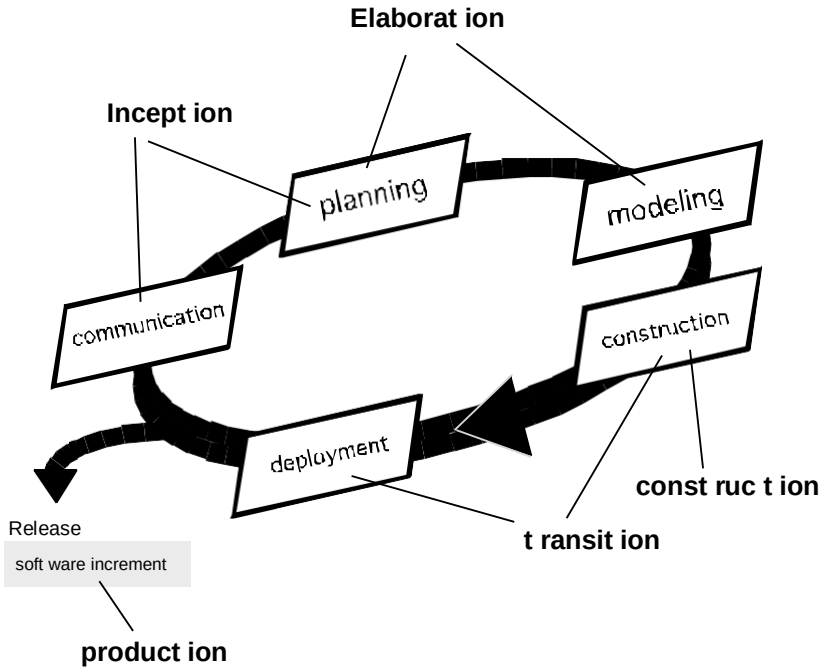
The ***inception*** phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, business requirements for the software are identified, a rough architecture for the system is proposed and a plan for the iterative, incremental nature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

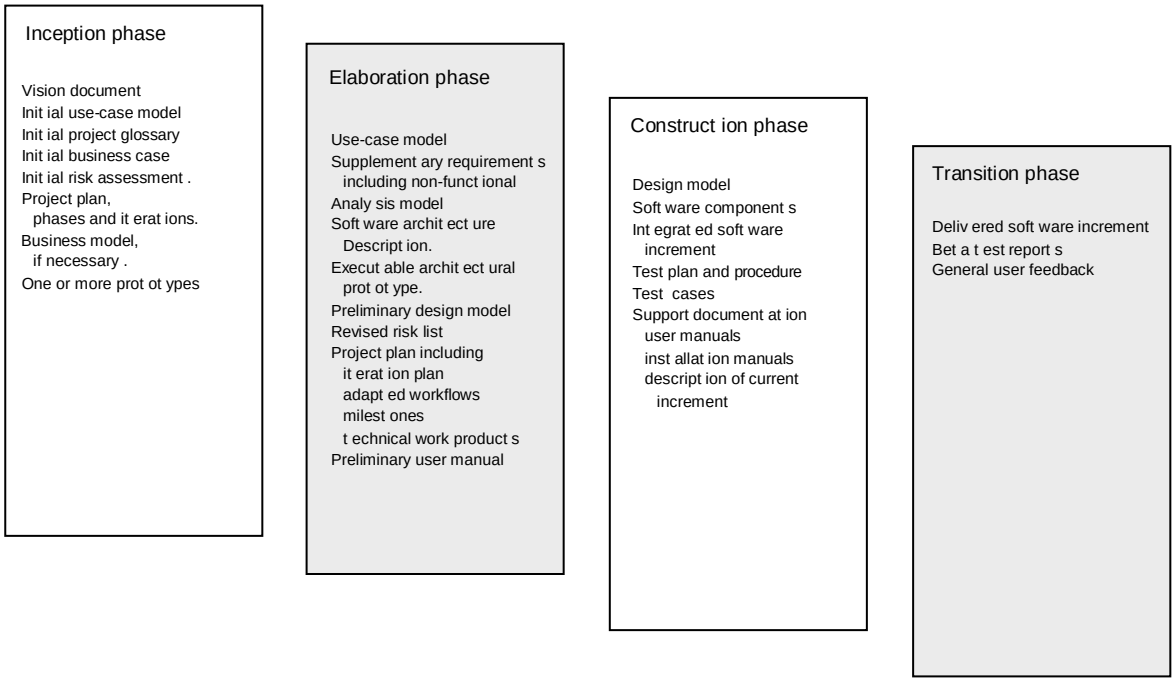
The most important work product produced during the inception is the use-case modell-a collection of use-cases that describe how outside actors interact with the system and gain value from it. The use-case model is a collection of software features and functions by describing a set of preconditions,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



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

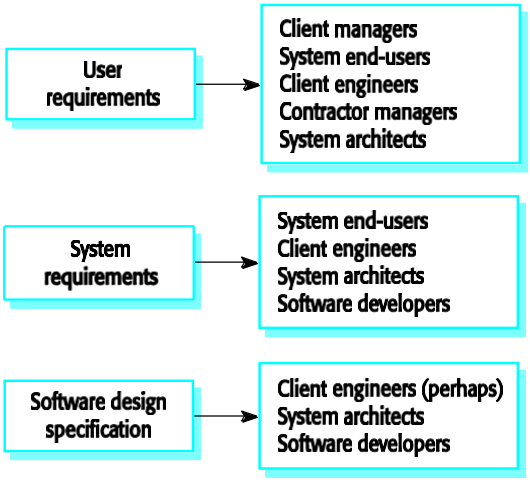
Requirement Definition:

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:



1) 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.

1.1) 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

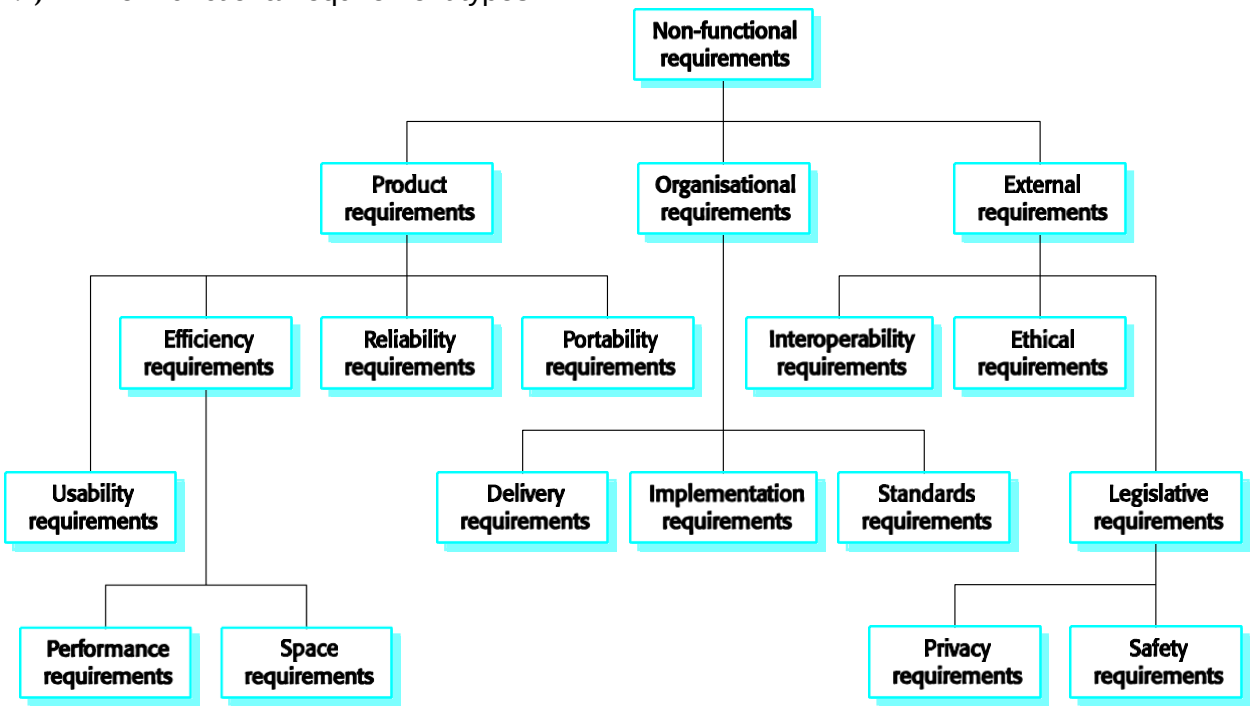
- There should be no conflicts or contradictions in the descriptions of the system facilities.

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.

1.2) 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.

1.3) 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.

2) 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.

3) 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.

3.1) 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 belimited.
- 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.

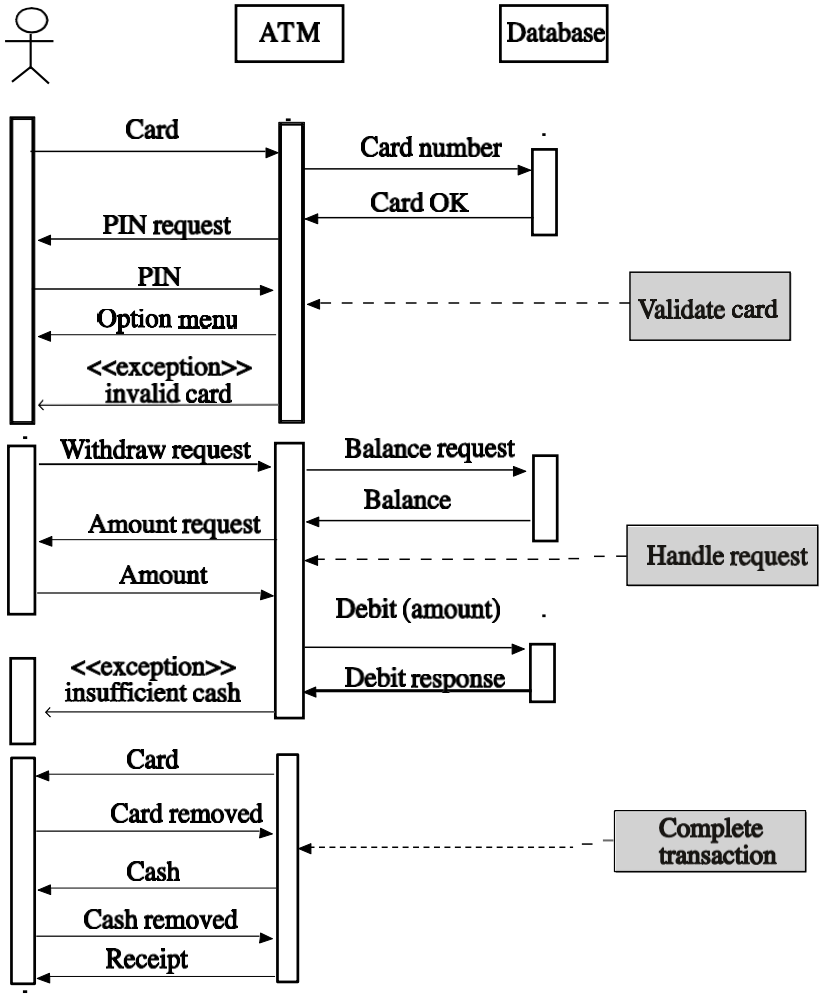
Graphical models

- 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:

1. Function
2. Description
3. Inputs
4. Source
5. Outputs
6. Destination
7. Action
8. Requires
9. Pre-condition
10. Post-condition
11. Side-effects

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

1. Description of the function or entity being specified
2. Description of its inputs and where these come from
3. Description of its outputs and where these go to
4. Indication of what other entities are used
5. Description of the action to be taken
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called

7. Description of the side effects of the operation.

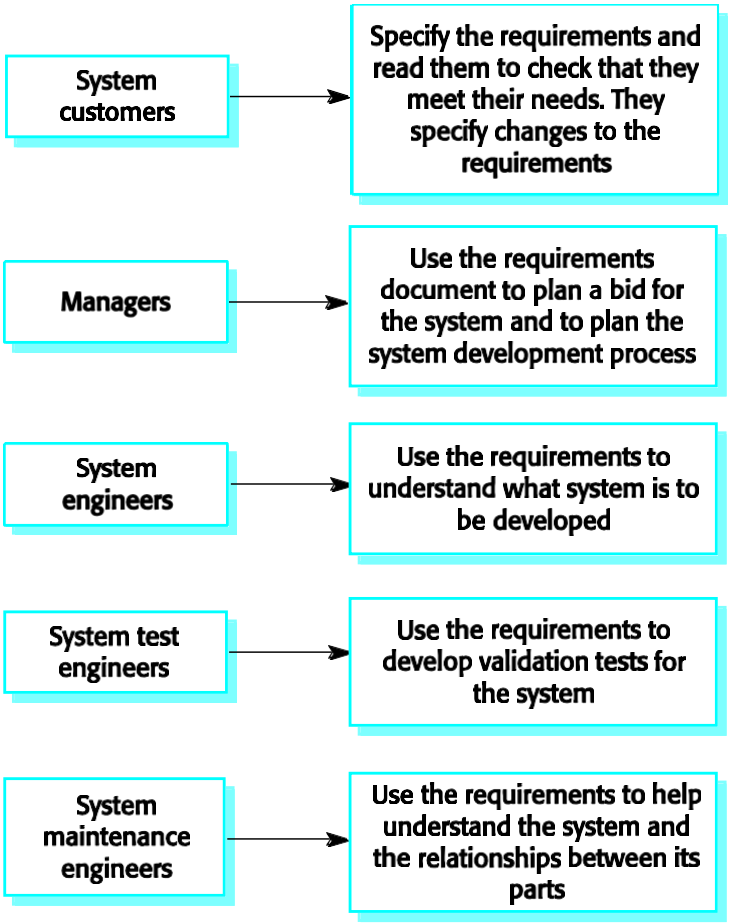
4) INTERFACE SPECIFICATION

- 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*** that 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.

5) 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.

1. Introduction.
   i) Purpose of the requirements document
   ii) Scope of the project
   iii) Definitions, acronyms and abbreviations
   iv) References
   v) Overview of the remainder of the document
2. General description.
   i) Product perspective
   ii) Product functions
   iii) User characteristics
   iv) General constraints
   v) Assumptions and dependencies
3. Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.
4. Appendices.
5. Index.

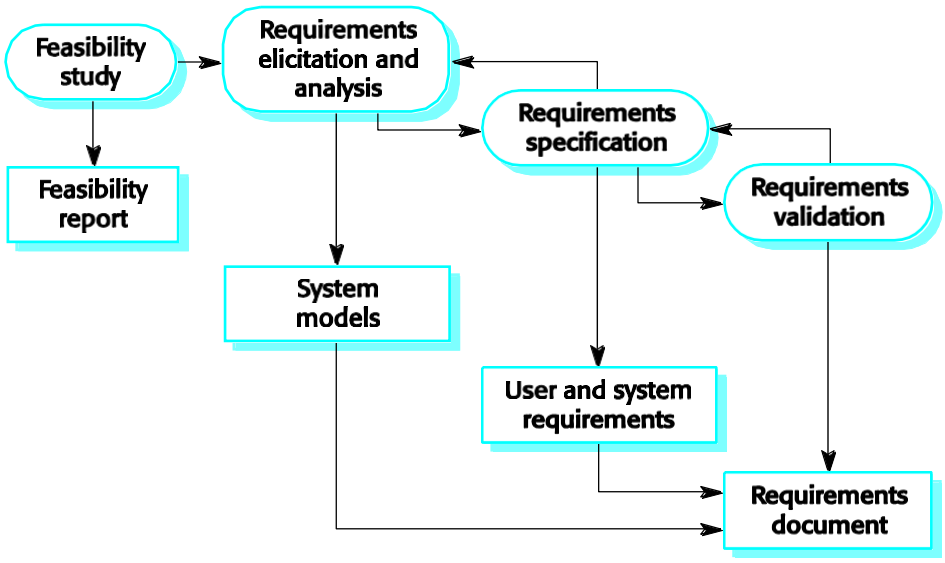
REQUIREMENTS ENGINEERING PROCESSES

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



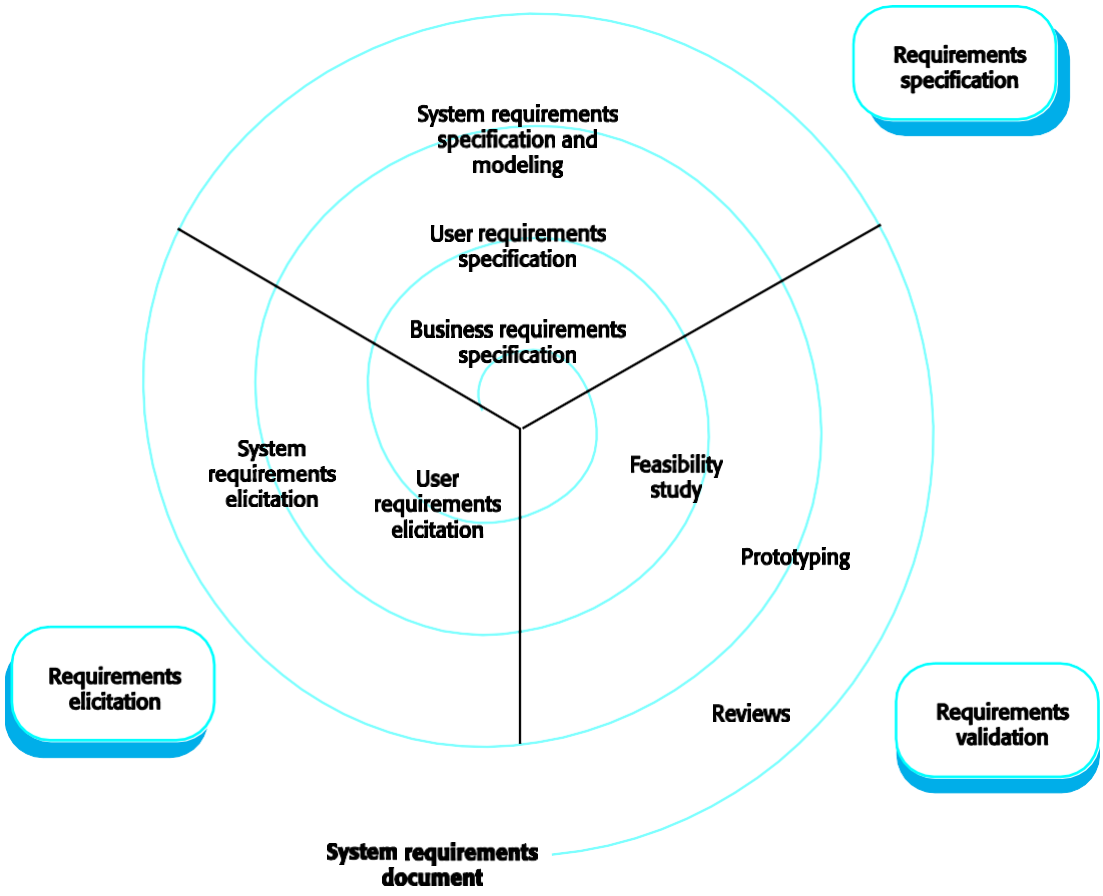
Requirements engineering:

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:

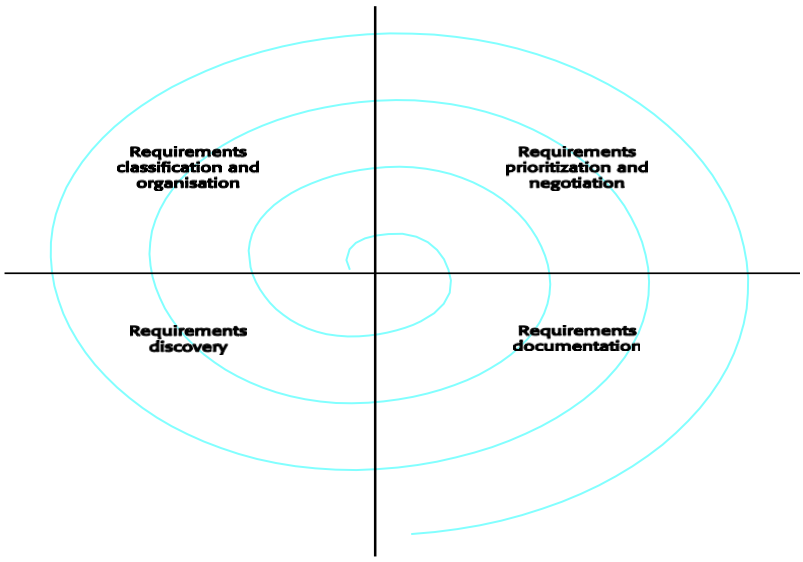
The requirement engineering process is requirements elicitation and analysis.

- 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

1. Requirements discovery
   - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
2. Requirements classification and organisation
   - Groups related requirements and organises them into coherent clusters.
3. Prioritisation and negotiation
   - Prioritising requirements and resolving requirements conflicts.
4. 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.

2.1) 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
  1. Bank customers
  2. Representatives of other banks
  3. Bank managers
  4. Counter staff
  5. Database administrators
  6. Security managers
  7. Marketing department
  8. Hardware and software maintenance engineers
  9. 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:

1. **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.
2. 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.

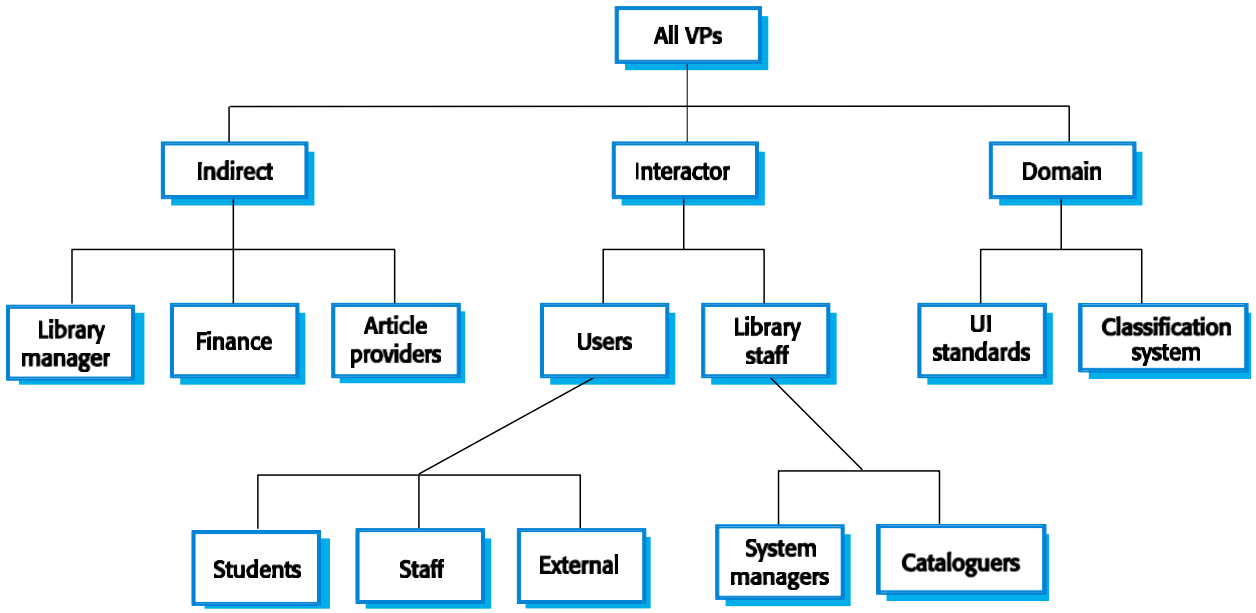
3. Domain viewpoints
   - Domain characteristics and constraints that influence the requirements. These viewpoints normally provide domain constraints that apply to the system. In an ATM, an example would be standards for inter-bank communications.

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 ofissues 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:

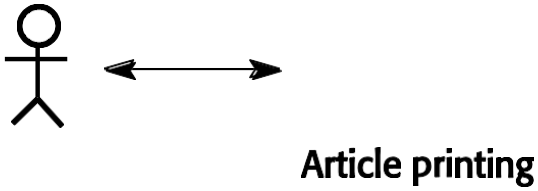
Scenarios are real-life examples of how a system can be used.

- 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.

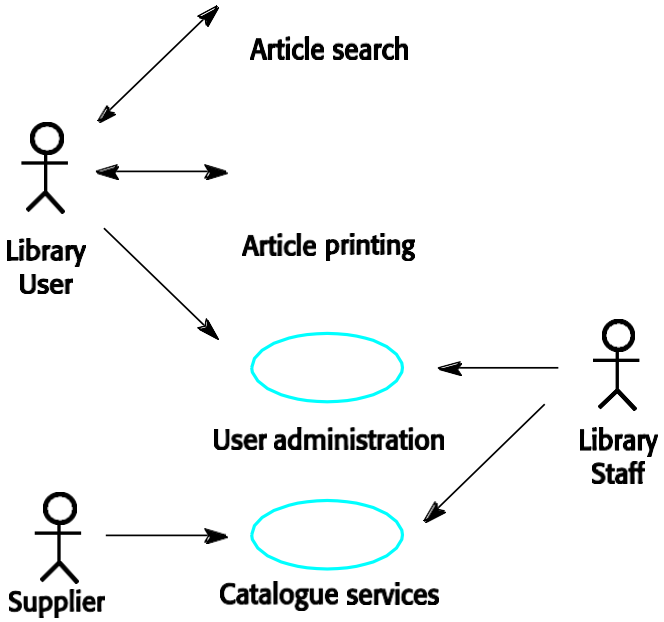
Use cases

- 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.

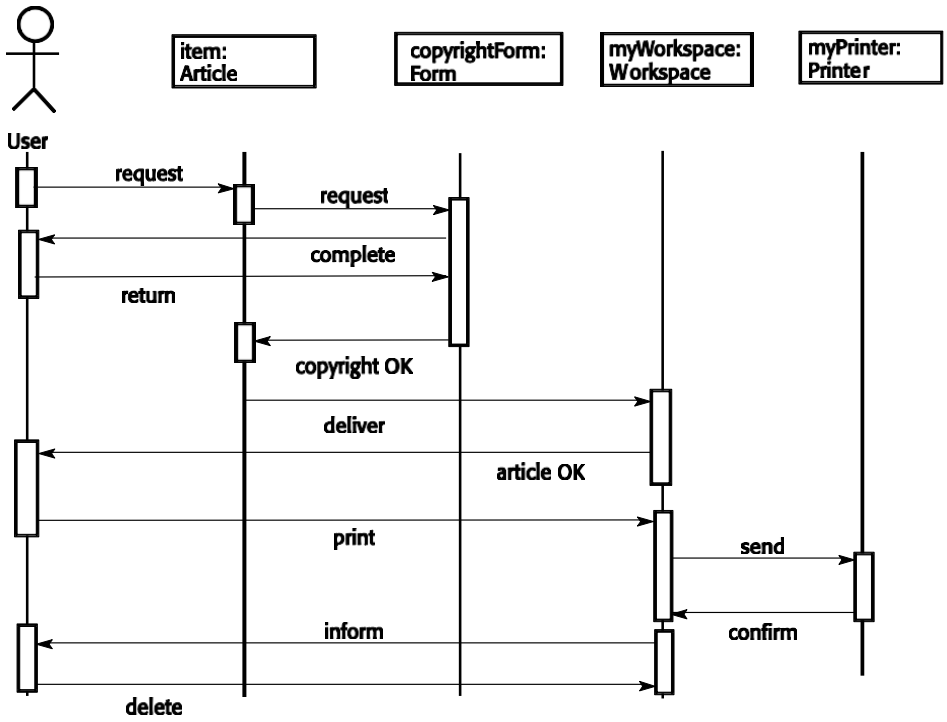
Article printing use-case:



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.

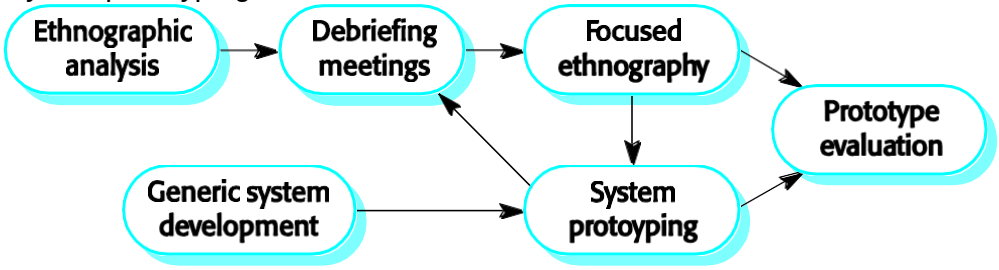
2.2) ETHNOGRAPHY:

- 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.

3) 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?

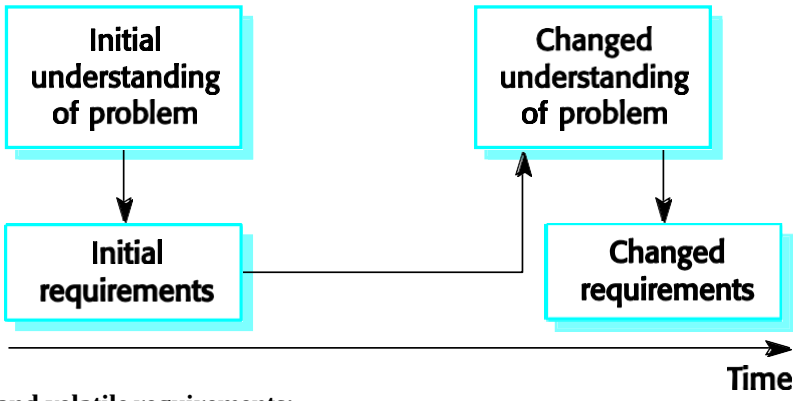
4) REQUIREMENTS MANAGEMENT

- 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:



4.1) 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.

4.2) 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;

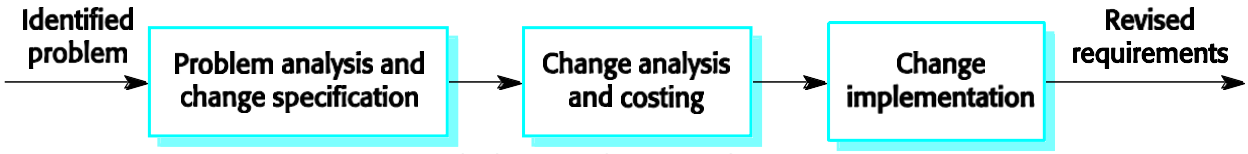
CASE tool support:

- 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.

4.3) 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.

UNIT-II

DESIGN ENGINEERING

Design engineering: Design process and design quality, design concepts,the design model, Creating an Architectural Design:Software architecture, data design, architectural styles and patterns, architectural design.
Modeling component-level design & performing user interface design: Designing Class based components, conducting component level design, Golden rules, user interface analysis and design.

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 DESIGNQUALITY:

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:

1. A design should exhibit an architecturethat
   a. has been created using recognizable architectural styles orpatterns
   b. is composed of components that exhibit good design characteristicsand
   c. canbeimplementedinanevolutionaryfashion,therebyfacilitatingimplementationand testing.
2. Adesignshouldbemodular;thatis,thesoftwareshouldbelogicallypartitionedintoelementsor subsystems.

3. A design should contain distinct representation of data, architecture, interfaces andcomponents.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable datapatterns.
5. A design should lead to components that exhibit independent functionalcharacteristics.

6. A design should lead to interface that reduce the complexity of connections between components and with the externalenvironment.
7. A design should be derived using a repeatable method that is driven byinformation obtained during software requirementsanalysis.
8. A design should be represented using a notation that effectively communication itsmeaning.

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 ofthe program, the generality of the functions that are delivered, and the security of the overallsystem.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistencyand documentation.
- ***Reliabilit****y* 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 theprogram.
- ***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 isdeveloped.

One application may stress functionality with a speci

al emphasis onsecurity.

Another may demand performance with particular emphasis on processing speed.

A third might focus on reliability.

2) DESIGNCONCEPTS:

M.A Jackson once said:"The beginning of wisdom for a software engineer is to recognize the difference betweengettingaprogramtowork,andgettingitright."Fundamentalsoftwaredesignconceptsprovidethe necessary framework for "getting it right."

I. Abstraction: Many levels of abstraction arethere.

- ✓ At the highest level of abstraction, a solution is stated in broad terms using the language ofthe problemenvironment.
- ✓ At lower levels of abstraction, a more detailed description of the solution isprovided.

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.**

II. 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 thecomponents.

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 externalevents.

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.

III. 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

1) Whether the pattern is capable to the currentwork,
2) Whether the pattern can bereused,
3) Whether the pattern can serve as a guide for developing a similar, but functionally orstructurally differentpattern.

IV. Modularity:

Software architecture and design patterns embody ***m odularity*;** 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 beintellectually 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 intomanageable 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 alsogrow.

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.

V. 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 inaccessibleto 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 with in 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. Designclasses:

The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes tobe implemented, and

2. 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 themetaphor.
- **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 domainclasses.
-
-
- **Persistent classes** represent data stores that will persist beyond the execution of thesoftware.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outsideworld.

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 designclass.**

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 noless.

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.

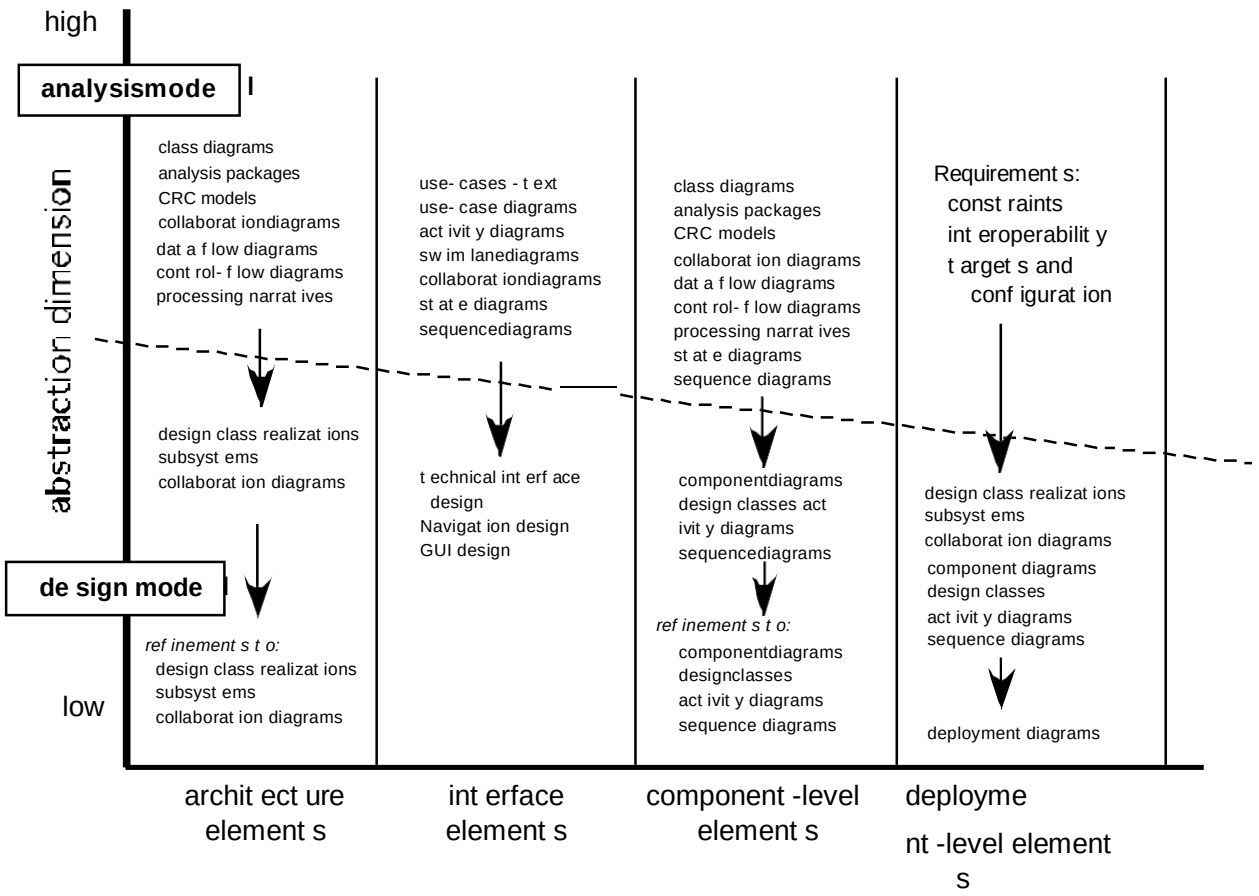
THE DESIGN MODEL:

- The design model can be viewed into differentdimensions.
- The process dimension indicates the evolution of the design model as design tasks are executed as a part of the softwareprocess.

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 fullydeveloped.



i. **Data designelements:**

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

ii. **Architectural designelements:**

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

1) Information about the application domain for the software to bebuilt.
2) Specific analysis model elements such as data flow diagrams or analysis classes,their relationships and collaborations for the problem at hand,and
3) The availability of architecturalpatterns

iii. **Interface designelements:**

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:

1) The userinterface(UI);
2) External interfaces to other systems, devices, networks, or other produces or consumers of information;and
3) Internal interfaces between various designcomponents.

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."

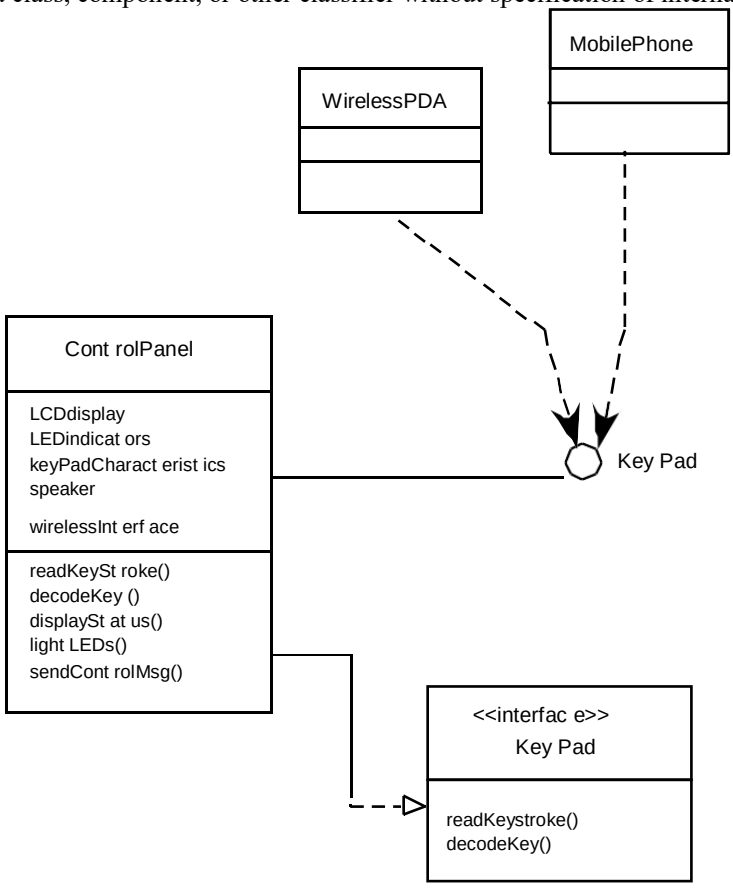
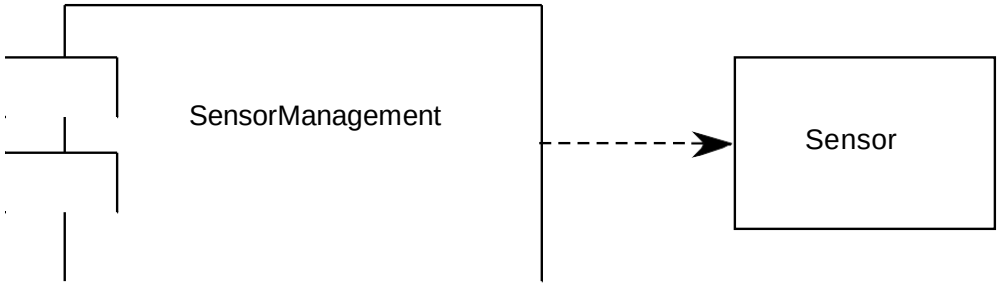


Figure 9 . 6 UML int erfac e represent at ion for **Co n t ro l Pa n e l**

iv. **Component- level design elements:** The component-level design for software is equivalent toa set of detaileddrawings.

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.



v. **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 thesoftware

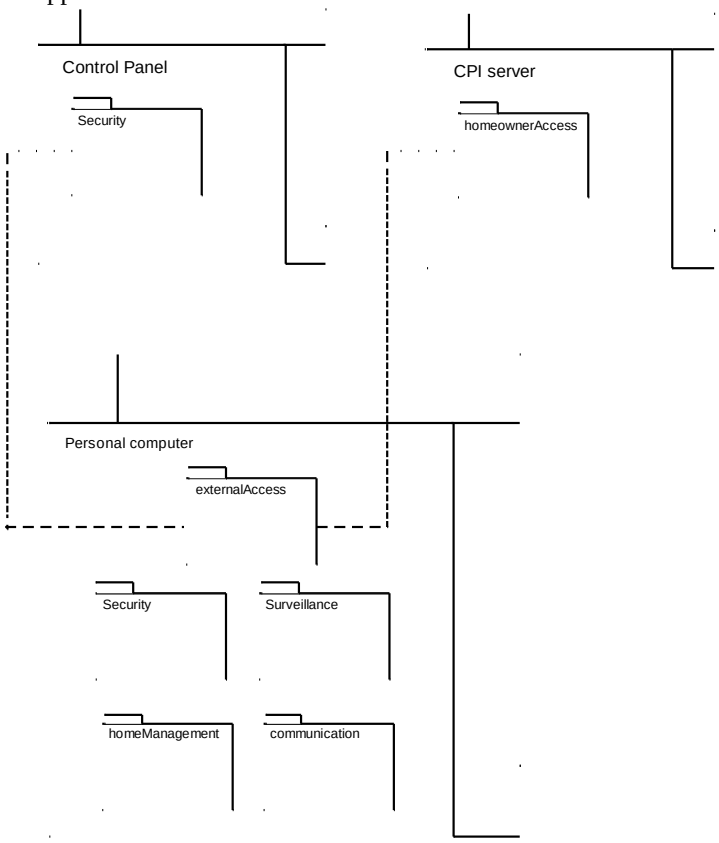


Figure 9 . 8 UML deploy m ent diagram for *SafeHom e*

ARCHITECTURAL DESIGN

1) SOFTWAREARCHITECTURE:

What IsArchitecture?

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 willtake,
- the structure and properties of the components that constitute the system,and
- the interrelationships that occur among all architectural components of asystem.

The architecture is a representation that enables a software engineer to
(1) analyze the effectiveness of the design in meeting its statedrequirements,
(2) consider architectural alternatives at a stage when making design changes is still relatively easy, (3) reducing the risks associated with the construction of thesoftware.

The design of software architecture considers two levels of the design pyramid
- datadesign
- architecturaldesign.

✓ Data design enables us to represent the data component of thearchitecture.
✓ Architecturaldesignfocusesontherepresentationofthestructureofsoftwarecomponents,their properties, andinteractions.

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 allparties (stakeholders) interested in the development of a computer-basedsystem.
- 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 operationalentity.
- Architecture"constitutesarelativelysmall,intellectuallygraspablemodelofhowthesystemis structured and how its components worktogether"

2) DATADESIGN:
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.

➢ Attheprogramcomponentlevel,thedesign ofdatastructuresandtheassociatedalgorithms required to manipulate them is essential to the creation of high-qualityapplications.
➢ Attheapplicationlevel,thetranslation ofadata model(derivedaspart ofrequirements engineering)intoadatabaseispivotaltoachievingthebusinessobjectivesofasystem.
➢ 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 businessitself.

2.1) 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.

2.2) Data design at the ComponentLevel:
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:
1. The systematic analysis principles applied to function and behavior should also be applied todata.
2. All data structures and the operations to be performed on each should beidentified.
3. A data dictionary should be established and used to define both data and programdesign.
4. Low-level data design decisions should be deferred until late in the designprocess.

5. The representation of data structure should beknown only to those modules that must make direct use of the data contained within thestructure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract datatypes.

3) ARCHITECTURAL STYLES ANDPATTERNS:

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

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

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

(3) ***Constraints*** that define how components can be integrated to form the system;and

(4) ***Semantic models*** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituentparts.

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:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather thanthe architecture in itsentirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle someaspect of its functionality at the infrastructurelevel.

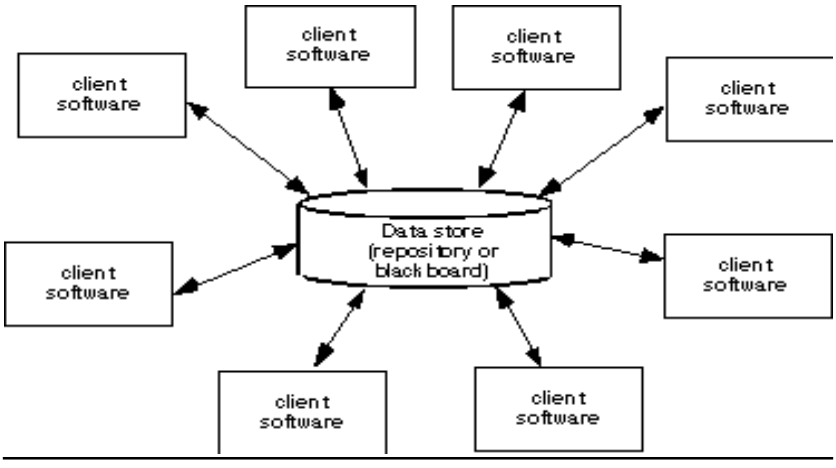
(3) Architectural patterns tend to address specific behavioral issues within the context ofthe architectural.

3.1) A Brief Taxonomy of Styles andPatterns

Data-centeredarchitectures:

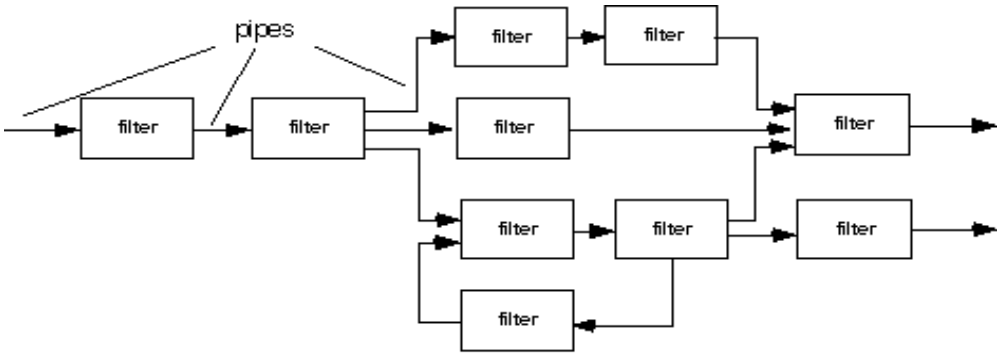
A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A variation on this approach transforms the repository into a "blackboard" that sends notification to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

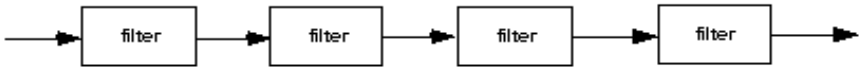


Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A ***pipe and filter pattern*** has a set of components, called ***filters***, connected by pipes that transmit data from one component 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 specifiedform.

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.



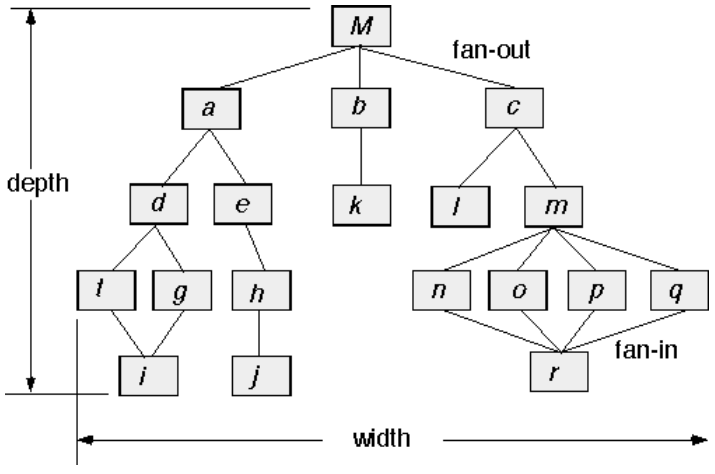
(a) pipes and filters



(b) batch sequential

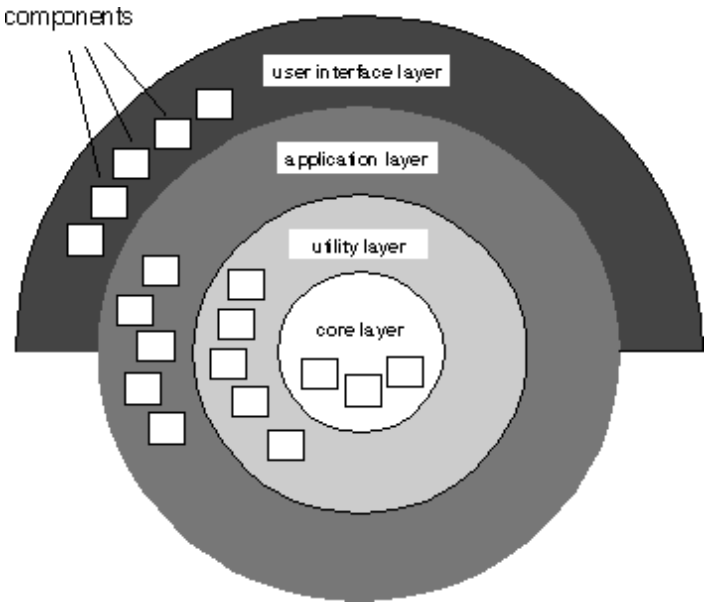
Call and return architectures. This architectural style enables a software designer(systemarchitect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within thiscategory:

- ***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 anetwork



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.



3.2) ArchitecturalPatterns:

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:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather thanthe architecture in itsentirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle someaspect of its functionality at the infrastructurelevel.

(3) Architectural patterns tend to address specific behavioral issues within the context ofthe 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 capabilityof a DBMS to the applicationarchitecture
- an ***application level persistence*** pattern that builds persistence features intothe applicationarchitecture

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 servercomponent**.**

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 thearchitecture?
- ✓ Does a distinct control hierarchy exist, and if so, what is the role ofcomponents within this control hierarchy?
- ✓ How do components transfer control within thesystem?
- ✓ How is control shared amongcomponents?

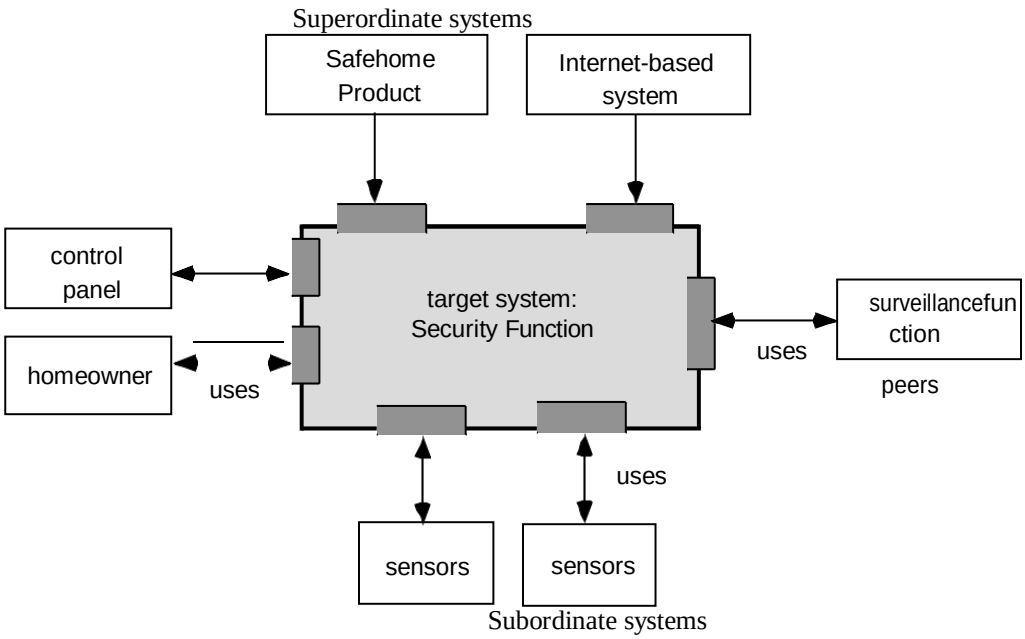
Data.

- ✓ How are data communicated between components?
- ✓ Is the flow of data continuous, or are data objects passed to the system sporadically?
- ✓ Whatisthe mode ofdatatransfer(i.e.,aredatapassedfromonecomponenttoanotheroraredata available globally to be shared among systemcomponents)?
- ✓ Do data components (e.g., a blackboard or repository) exist, and if so, what is theirrole?
- ✓ How do functional components interact with datacomponents?
- ✓ Are data components *passive* or *active* (i.e., does the data component actively interact withother components in the system)? How do data and control interact within thesystem?

4) ARCHITECTURALDESIGN:

I Representing the System inContext:

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 thefigure



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

II DefiningArchetypes:

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 alarmindicators.
- **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the targetsystem
- **Indicator:** An abstraction that represents all mechanisms for indication that an alarm condition isoccurring.
- **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.

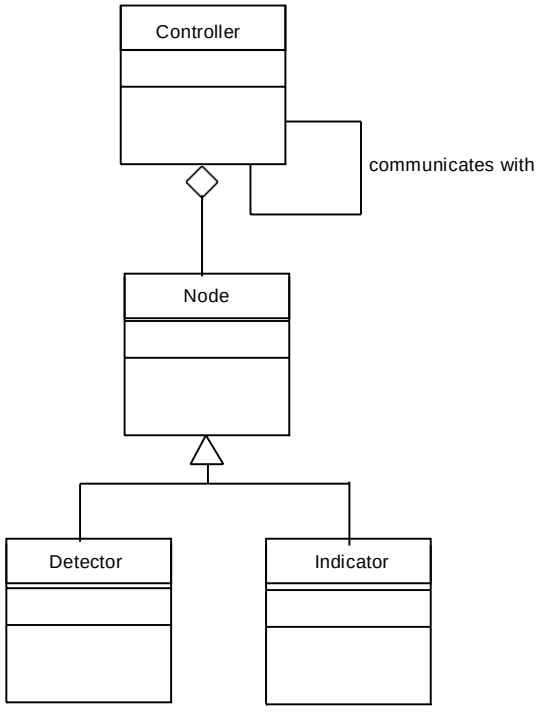


Figure 10.7 UML relat ionships for SafeHome security function archetypes (adapted f rom [BOS00])

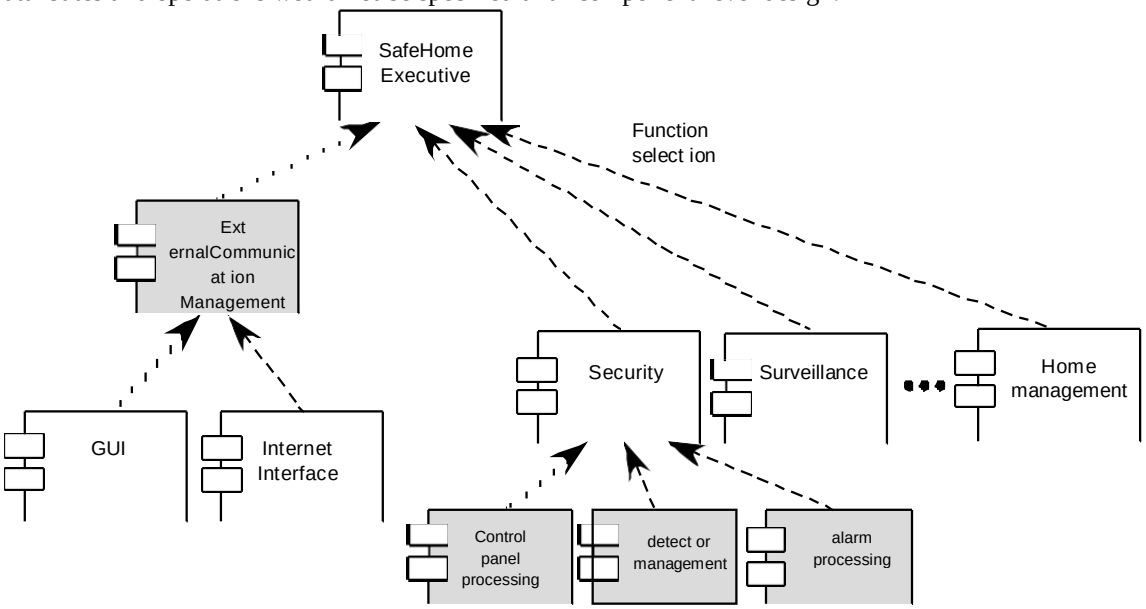
III Refining the Architecture intoComponents:

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:

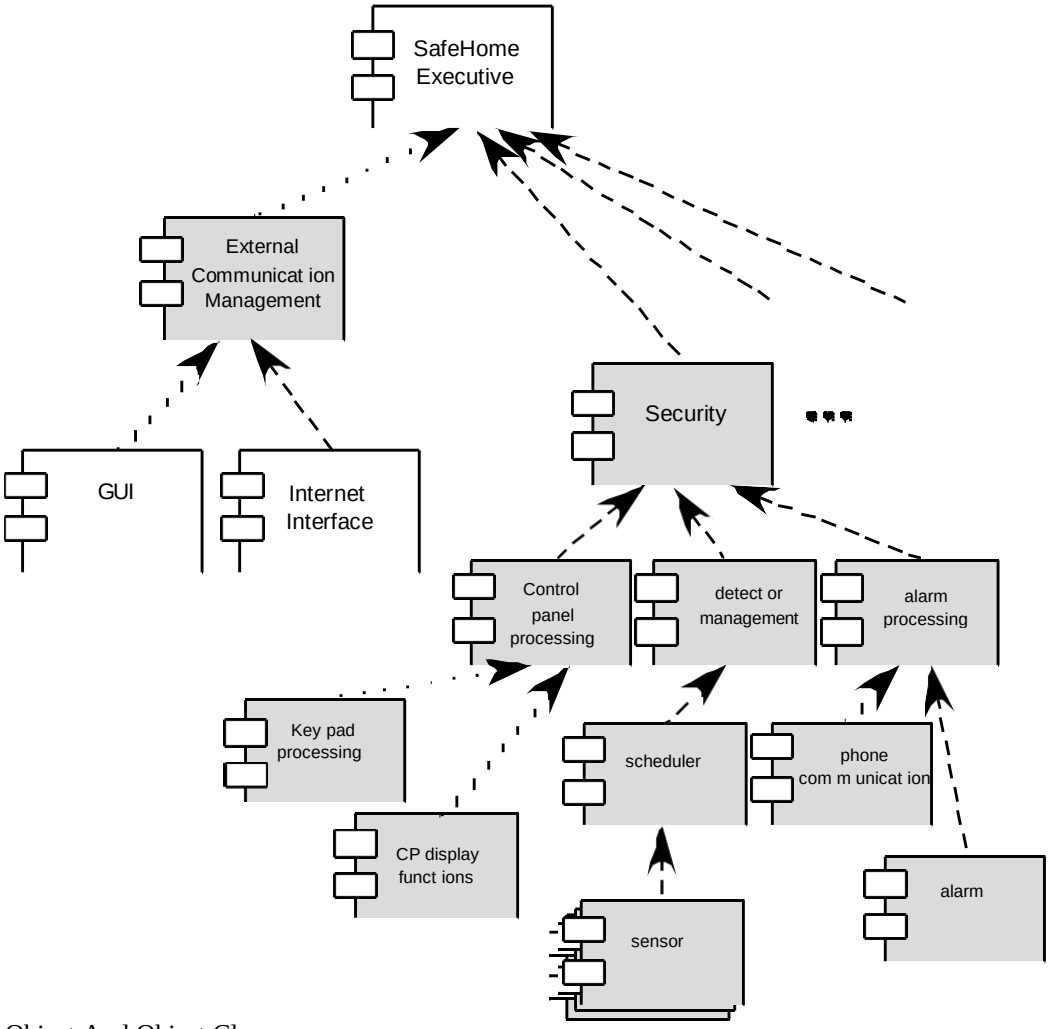
- *External communication management*- coordinates communication of thesecurity function with externalentities
- *Control panel processing*- manages all control panelfunctionality.
- *Detector management*- coordinates access to all detectors attached to thesystem.
- *Alarm processing*- verifies and acts on all alarmconditions.

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 structureand components areappropriate.



Object And Object Classes

- Object:Anobjectisanentitythathasastateandadefinedsetofoperationsthatoperateonthat state.
- An obect class defination is both a type specification and a template for creatingobects.
- It includes declaration of all the attributes and operations that are associated with objectof 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 thesystem.

2)Design the systemarchitecture

3)Identify the principle objects in the system.

4)Develop a design models

5) Specify the object interfaces

Systems context and modes ofuse

- Itspecifythecontextofthe system.italsospecifytherelationshipsbetweenthesoftwarethatis being designed and its externalenvironment.
- If the system context is a static model it describe the other system in thatenvironment.
- 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 systemenvironment have beendefined
- We can use the above information as basis for designing the SystemArchitecture.

ObjectIdentification

- This process is actually concerned with identifying the objectclasses.
- We can identify the object classes by thefollowing

1)Use a grammaticalanalysis
2)Use a tangible entities
3)Use a behaviourialapproach
4) Use a scenario based approach

Designmodel

- Design models are the bridge between the requirements andimplementation.
- There are two type of designmodels

1) Static model describe the relationship between the objects.
2)Dynamic model describe the interaction between theobjects

- Object Interface SpecificationIt is concerned with specifying the details of the interfaces toan objects.
- Designevolution

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.

Designing Class-Based Components

1. Component-level Design Principles

• **Open-closed principle** – A module or component should be open for extension but closed for modification
– The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
• **Liskov substitution principle** – Subclasses should be substitutable for their base classes
– A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
• **Dependency inversion principle** – Depend on abstractions (i.e., interfaces); do not depend on concretions
– The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
• **Interface segregation principle** – Many client-specific interfaces are better than one general purpose interface – For a server class, specialized interfaces should be created to serve major categories of clients
– Only those operations that are relevant to a particular category of clients should be specified in the interface
Component Packaging Principles
• **Release reuse equivalency principle** – The granularity of reuse is the granularity of release
– Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
• **Common closure principle** – Classes that change together belong together
– Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
• **Common reuse principle** – Classes that aren't reused together should not be grouped together
– Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

2.Component-Level Design Guidelines

• **Components** – Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
– Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
– Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)

• **Dependencies and inheritance in UML** – Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
– Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency Cohesion

3.Cohesion

It is the "single-mindedness' of a component .
• It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
• The objective is to keep cohesion as high as possible
• The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
– **Functional** • A module performs one and only one computation and then returns a result
– **Layer•** A higher layer component accesses the services of a lower layer component
– **Communicational** • All operations that access the same data are defined within one class
• Kinds of cohesion (continued)
– **Sequential** • Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
– **Procedural** • Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
– **Temporal** • Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
– **Utility** • Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

4. Coupling • As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
• As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
• Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
• The objective is to keep coupling as low as possible
• The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
• Data coupling • Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
• Stamp coupling • A whole data structure or class instantiation is passed as a parameter to an operation
• Control coupling • Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B() • Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
• Common coupling• A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
• Content coupling • One component secretly modifies data that is stored internally in another component • Other kinds of coupling (unranked)
• Subroutine call coupling • When one operation is invoked it invokes another operation within side of it
• Type use coupling • Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration • If/when the type definition changes, every component that declares a variable of that data type must also change
• Inclusion or import coupling • Component A imports or includes the contents of component B
• External coupling • A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
   • These classes are usually not present in the analysis or architectural models
   • These classes include GUI components, operating system components, data management components,

networking components, etc.

3) Elaborate all design classes that are not acquired as reusable components

- Specify message details (i.e., structure) when classes or components collaborate
- Identify appropriate interfaces (e.g., abstract classes) for each component
- Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
- Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

4) Describe persistent data sources (databases and files) and identify the classes required to manage them

5) Develop and elaborate behavioral representations for a class or component

- This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class

6) Elaborate deployment diagrams to provide additional implementation detail

- Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments

7) Factor every component-level design representation and always consider alternatives

- Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
- The final decision can be made by using established design principles and guidelines

USER INTERFACE DESIGN

Interface design focuses on three areas of concern:
(1) the design of interfaces between softwarecomponents,
(2) the design ofinterfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities),and
(3) the design of the interface between a human (i.e., the user) and thecomputer.

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.

1.1 THE GOLDENRULES

Theo Mandel coins three "golden rules":

1. Place the user incontrol.
2. Reduce the user's memoryload.
3. Make the interfaceconsistent.

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 keyboardcommands.
- **Allow user interaction to be interruptible and undoable**. User stop and do something and then resume where left off. Be able to undo anyaction.
- **Streamline interaction as skill levels advance and allow the interaction to be customized**. Perform same actions repeatedly; have macro mechanism so user can customizeinterface.
- **Hide technical internals from the casual user**. Never required to use OS commands; file management functions or other arcane computingtechnology.
- **Design for direct interaction with objects that appear on the screen.** User has feel of control when interact directly with objects; stretch anobject.

Reduce the User's Memory Load:

- ✓ The more a user has to remember, the more error-prone interaction with the system willbe.
- ✓ Good interface design does not tax the user'smemory
- ✓ Systemshouldrememberpertinentdetailsandassisttheuserwithinteractionscenariothat assists userrecall.

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 recallthem.
- **Establish meaningful defaults**. Initial defaults for average user; but specify individual preferences with a resetoption.
- **Define shortcuts that are intuitive**. Use mnemonics likeAlt-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 tomemorize
- **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 optionspresented

Make the Interface Consistent

Interface present and acquire information in a consistent fashion.

1. All visual information is organized to a design standard for all screendisplays
2. Input mechanisms are constrained to limited set used consistentlythroughout theapplication
3. Mechanisms for navigation from task to task are consistently defined andimplemented

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 togo.
- **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 toscaling)

1.2 USER INTERFACEDESIGN

1.2.1 Interface DesignModels

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

- The software engineer creates a *designmodel*,
- a human engineer (or the software engineer) establishes a *usermodel*,
- 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 *implementationmodel*.

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, intermittentusers.
- ✓ Knowledgeable, frequentusers.

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."**

1.2.2 The User Interface Design Process: (steps in interfacedesign)

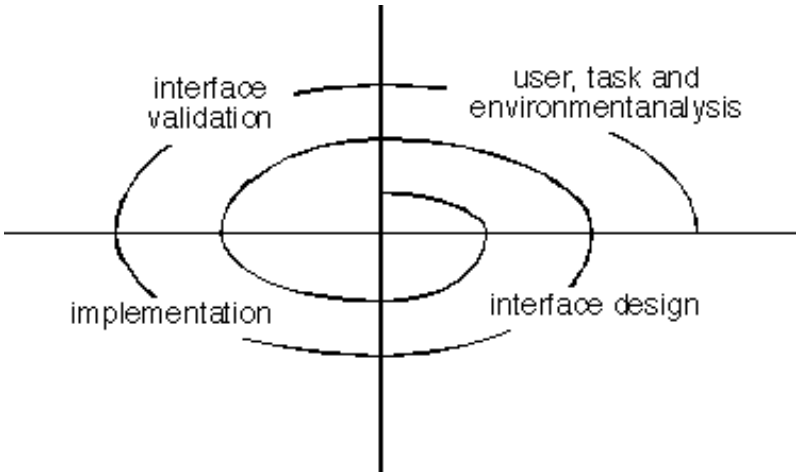
The user interface design process encompasses four distinct framework activities :

1. User, task, and environment analysis andmodeling

2. Interfacedesign

3. Interfaceconstruction

4. Interfacevalidation



User Interface Design Process

(1) User Task and EnvironmentalAnalysis:

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 locatedphysically?
- Will the user be sitting, standing, or performing other tasks unrelated totheinterface?
- Does the interface hardware accommodate space, light, or noiseconstraints?
- Are there special human factors considerations driven by environmentalfactors?

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) InterfaceDesign:
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) InterfaceConstruction(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) InterfaceValidation:
Validation focuses on
(1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general userrequirements;

(2) the degree to which the interface is easy to use and easy to learn;and

(3) the users' acceptance of the interface as a useful tool in theirwork.

12.3 INTERFACEANALYUSIS
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 thatfollow.

12.3.1 Useranalysis
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 theirrequirements.

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 manufacturingworkers?
- What level of formal education does the average userhave?
- Aretheuserscapable oflearningfromwrittenmaterials orhavetheyecpressedadesireofclassroom training?
- Are users expert typists or keyboardphobic?
- What is the age range of the usercommunity?
- Will the users be represented predominately by onegender?
- How are users compensated for the work theyperform?
- Do users work normal office hours, or do they work until the job isdone.
- Isthesoftwaretobeanintegralpart oftheworkusersdo,orwillitbeusedonlyoccasionally?

- What is the primary spoken language amongusers?
- What are the consequences if a user makes a mistake using thesystem?
- Are users experts in the subject matter the is addressed by thesystem?
- Do users want to know about the technology that sits behind theinterface?

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.

12.3.2 Task Analysis andModeling

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

- What work will the user perform in specificcircumstances?
- What specific problem domain objects will the user manipulate as work isperformed?
- What is the sequence of work tasks-theworkflow?
- What is the hierarchy oftasks?

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 userinterface.

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 asystem.

The use-case provides a basic description ofoneimportant work task for the computer-aided designsystem. 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 majortasks.

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 aredefined.

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.

1. Eachuserimplementsdifferenttasksviatheinterface;therefore,thelookandfeeloftheinterfacedesigned for the patient will be different form the one defined for pharmacists orphysicians.

2. 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 thephysician)
3. 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 distributioncenter.

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 identifyinginformation
- Specifyname
- Specifyuserid
- Specify PIN andpassword
- Specify prescriptionnumber
- Specify date refill isrequired

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.

12.3.3 Analysis of DisplayContent

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 allfunctions.
- How will the user request help? Options include a help menu, a special function day, or a HELPcommand.
- 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 screenlocation.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or controlsequence.
- 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 ofhypertext.

Ingeneral,everyerrormessageorwarningproducedbyaninteractivesystemshouldhave thefollowing characteristics:

- The message should describe the problem in language the user canunderstand.
- The message should provide constructive advice for recovering form theerror.
- The message should indicate any negative consequences of the error(e.g., potentiallycorrupted data files)so that the user can check to ensure that they have notoccurred.
- 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 dooccur.

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

- Will every menu option have a correspondingcommand?

- What form will commands take? Options include a control sequence (e.g., alt-p), function keys, ora typedword.
- Howdifficult willit betolearnandrememberthecommands? Whatcanbedoneifacommandis forgotten?
- Can commands be customized or abbreviated by theuser?
- Are menu labels self-explanatory within the context of theinterface?
- Are submenus consistent with the function implied by a master menuitem?

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.

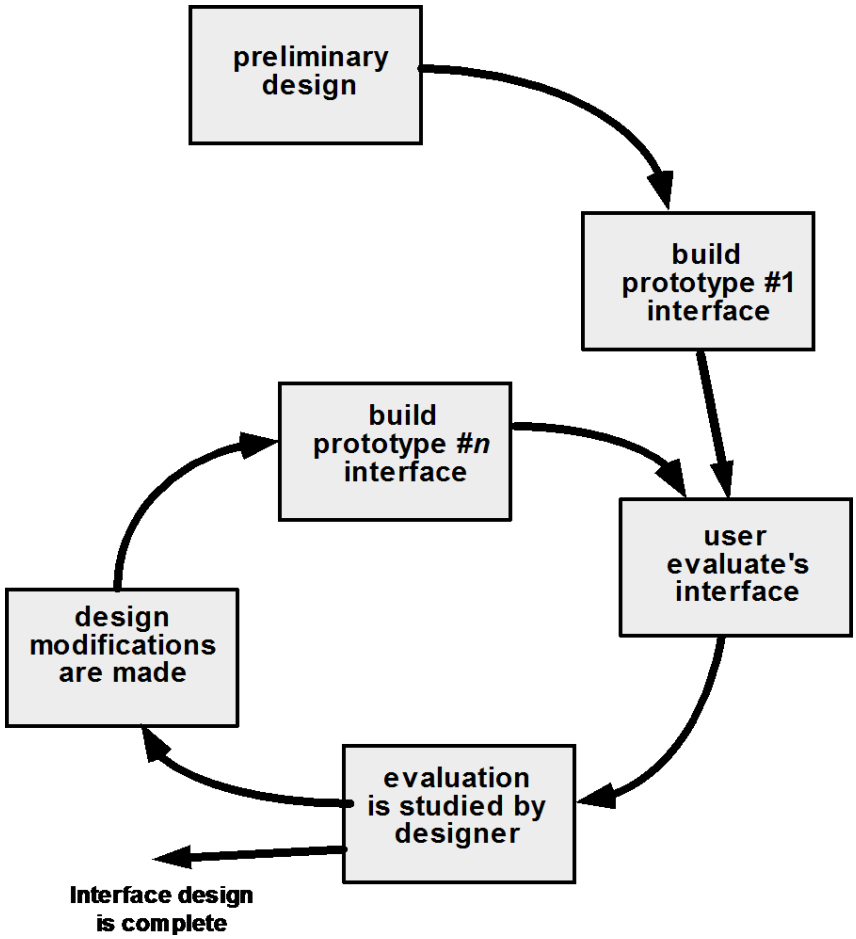
12.5 DESIGN EVALUATION

Afterthedesignmodel hasbeencompleted,afirst-levelprototypeiscreated.Theprototypeisevaluatedbytheuser, whoprovidesthedesignerwithdirectcommentsabouttheefficacyoftheinterface.Inaddition,ifformalevaluation 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 designreviews:

1. Thelengthandcomplexityofthe writtenspecificationofthe systemanditsinterfaceprovideanindication of the amount of learning required by user of thesystem.
2. Thenumberofusertasksspecifiedandtheaverage numberofactionspertaskprovideanindication on interaction time and the overall efficiency of thesystem.
3. Thenumber ofactions,tasks,and systemstatesindicated bythedesignmodelimplythememoryloadon users of thesystem.
4. Interface styles, help facilities, and error handling protocol provide a general indication ofthe complexity of the interface and the degree to which it will be accepted by theuser.

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

UNIT-III
INTRODUCTION TO UML

Introduction to UML: Priciples of modelling,Conceptual model of UML,Class and Object Diagrams: terms,concepts,modelling techniques.
Behavioural Modeling :Interaction diagrams,use case diagrams,activity diagrams,state chart diagrams,component diagrams and deployment diagrams.

Unified Modeling Language (UML) UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

1.Principles of modelling

First principle of modelling:
The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
Choose your models well. The right models will highlight the most nasty development problems. Wrong models will mislead you, causing you to focus on irrelevant issues.
Second principle of modelling:
Every model may be expressed at different levels of precision.
Sometimes, a quick and simple executable model of the user interface is exactly what you need. At other times, you have to get down to complex details such as cross-system interfaces or networking issues etc.
In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is viewing it. An analyst or an end user will want to focus on issues of what and a developer will want to focus on issues of how.
Third principle of modelling:
The best models are connected to reality.
In software, the gap between the analysis model and the system's design model must be less. Failing to bridge this gap causes the system to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one whole.
Fourth principle of modelling:
No single model is sufficient. Every non-trivial system is best approached through a small set of nearly independent models.
In the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.
The same is true of object-oriented systems. To understand the architecture of such a system, you need several complementary and interlocking views: a use case view (exposing the requirements of the system), a design view (capturing the vocabulary of the problem space and the solution space), a process view (modelling the distribution of the system's processes and threads), an implementation view (addressing the physical realization of the system) and a deployment view (focusing on system engineering issues). Depending on the nature of the system, some models may be more important than others.

2.Building blocks of UML (Conceptual model of UML)

These are the fundamental elements in UML. Every diagram can be represented using these building blocks. The building blocks of UML contains three types of elements. They are:

2.1) Things (object oriented parts of uml)
2.2) Relationships (relational parts of uml)
2.3) Diagrams

2.1Things

A diagram can be viewed as a graph containing vertices and edges. In UML, vertices are replaced by things, and the edges are replaced by relationships. There are four types of things in UML. They are:

2.1.1) Structural things (nouns of uml – static parts)
2.1.2) Behavioral things (verbs of uml – dynamic parts)
2.1.3) Grouping things (organizational parts)
2.1.4) Annotational things (explanatory parts)

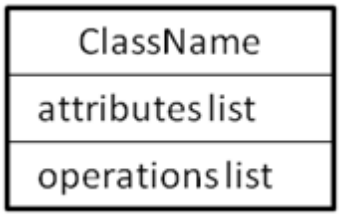
2.1.1.Structural things

Represents the static aspects of a software system. There are seven structural things in UML.

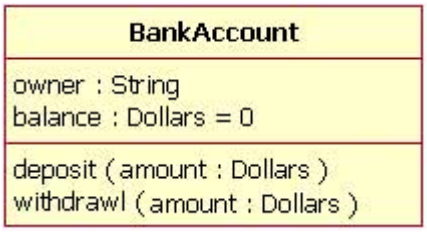
They are:

Class: A class is a collection of similar objects having similar attributes, behavior, relationships and semantics. Graphically class is represented as a rectangle with three compartments.

Graphical representation:



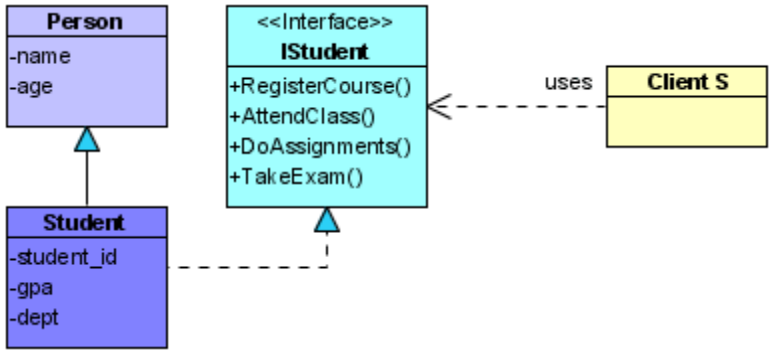
Example:



Interface: An interface is a collection of operation signatures and/or attribute definitions that ideally define a cohesive set of behavior. Graphically interface is represented as a circle or a class symbol stereotyped with interface.

Graphical representation:

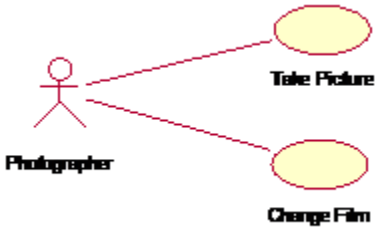
or



Use Case: A use case is a collection of actions, defining the interactions between a role (actor) and the system. Graphically use case is represented as a solid ellipse with its name written inside or below the ellipse.

Graphical representation:

Example:

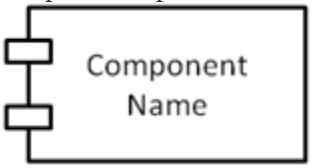


Collaboration: A collaboration is the collection of interactions among objects to achieve a goal. Graphically collaboration is represented as a dashed ellipse. A collaboration can be a collection of classes or other elements.

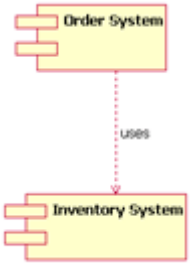
Graphical representation:

Example:

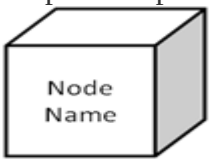
Component: A component is a physical and replaceable part of a system. Graphically component is represented as a tabbed rectangle. Examples of components are executable files, dll files, database tables, files and documents.
Graphical representation:



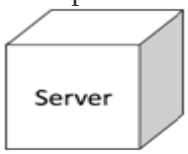
Example:



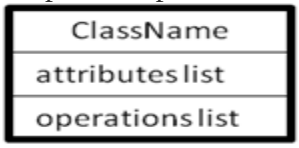
Node: A node is a physical element that exists at run time and represents a computational resource. Graphically node is represented as a cube. Examples of nodes are PCs, laptops, smart phones or any embedded system.
Graphical representation:



Example:



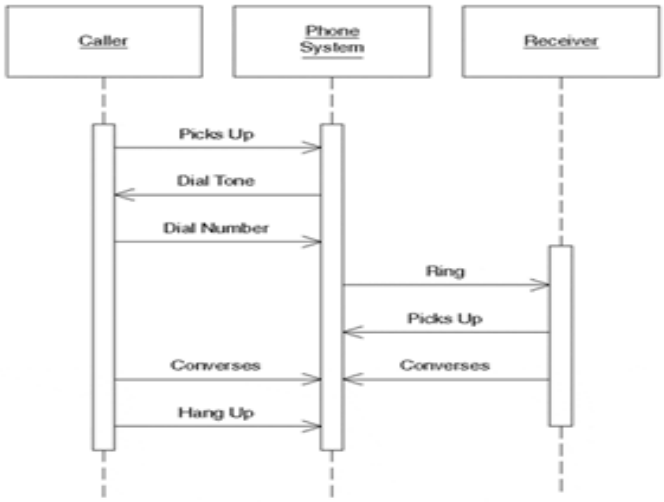
Active Class: A class whose objects can initiate its own flow of control (threads) and work in parallel with other objects. Graphically active class is represented as a rectangle with thick borders.
Graphical representation:



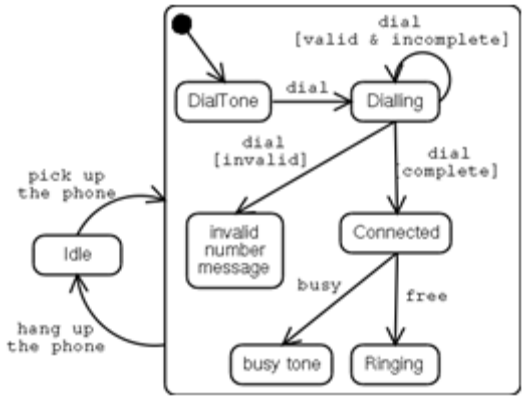
2.2 Behavioral Things

Represents the dynamic aspects of a software system. Behavior of a software system can be modeled as interactions or as a sequence of state changes.

Interaction: A behavior made up of a set of messages exchanged among a set of objects to perform a particular task. A message is represented as a solid arrow. Below is an example of interaction representing a phone conversation:



State Machine: A behavior that specifies the sequences of states an object or interaction goes through during its' lifetime in response to events. A state is represented as a rectangle with rounded corners. Below is an example of state machine representing the states of a phone system:

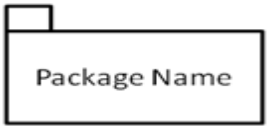


2.3.Grouping Things

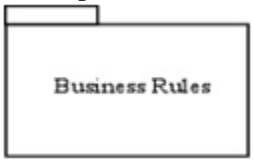
Elements which are used for organizing related things and relationships in models.

Package: A general purpose mechanism for organizing elements into groups. Graphically package is represented as a tabbed folder. When the diagrams become large and cluttered, related are grouped into a package so that the diagram can become less complex and easy to understand.

Graphical representation:



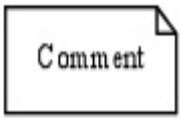
Example:



2.4 Annotational Things

Note: A symbol to display comments. Graphically note is represented as a rectangle with a dog ear at the top right corner.

Graphical representation:

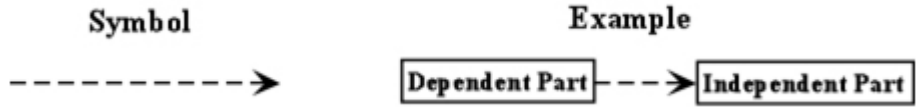


2.2 Relationships

The things in a diagram are connected through relationships. So, a relationship is a connection between two or more things.

Dependency: A semantic relationship, in which a change in one thing (the independent thing) may cause changes in the other thing (the dependent thing). This relationship is also known as "using" relationship. Graphically represented as dashed line with stick arrow head.

Graphical representation:



Example:

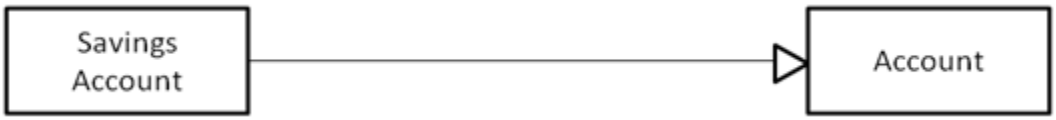
Association: A structural relationship describing connections between two or more things. Graphically represented as a solid line with optional stick arrow representing navigation.

Example:

Generalization: Is a generalization-specialization relationship. Simply put this describes the relationship of a parent class (generalization) to its subclasses (specializations). Also known as "is-a" relationship.

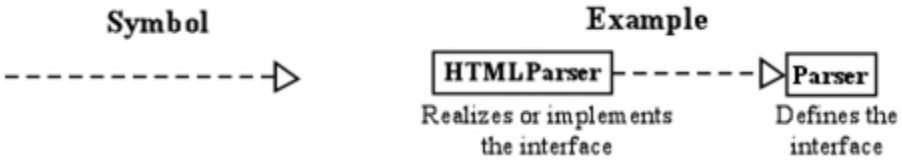
Graphical representation:

Example:



Realization: Defines a semantic relationship in which one class specifies something that another class will perform. Example: The relationship between an interface and the class that realizes or executes that interface.

Graphical representation and Example:



2.3 Diagrams

A diagram is a collection of elements often represented as a graph consisting of vertices and edges joining these vertices. These vertices in UML are things and the edges are relationships. UML includes nine diagrams:

1) Class diagram 2) Object diagram 3) Use case diagram 4) Component diagram 5) Deployment diagram 6) Sequence diagram 7) Collaboration diagram 8) Statechart diagram and 9) Activity diagram.

- A **class diagram** in the Unified Modeling Language (UML) is a type of static structure **diagram** that describes the structure of a system by showing the system's **classes**, their attributes, operations (or methods), and the relationships among objects.
- An **object diagram** shows this relation between the instantiated classes and the defined class, and the relation between these **objects** in the system
- A **use case diagram** can summarize the details of your system's users (also known as actors) and their interactions with the system.
- A **component diagram** is a collection of vertices and arcs and commonly contain **components**, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.
- A **deployment diagram** is a **diagram** that shows the configuration of run time processing nodes and the components that live on them. **Deployment diagrams** is a kind of structure **diagram** used in modeling the physical aspects of an object-oriented system.
- They capture the interaction between objects in the context of a collaboration. **Sequence Diagrams** are time focus and they show the order of the interaction
- **Collaboration diagrams** are used to show how objects interact to perform the behaviour of a particular use case, or a part of a use case.
- **State** machine can be defined as a machine which defines different **states** of an object and these **states** are controlled by external or internal events. Activity diagram is a special kind of a Statechart diagram.
- **Activity diagram** is another important behavioral **diagram** in UML **diagram** to describe dynamic aspects of the system. **Activity diagram** is essentially an advanced version of flow chart that modeling the flow from one **activity** to another **activity**.

Diagrams in UML can be broadly classified as:

1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.

2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

The diagrams are broadly classified into static diagrams and dynamic diagrams.

Static diagrams are

Class diagrams
Object diagrams
Use case diagrams
Component diagrams
Deployment diagrams

Dynamic diagrams are Sequence diagrams
Collaboration diagrams
State chart diagrams
Activity diagrams

Class Diagram

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.

The class diagrams are widely used in the modelling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application,

however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

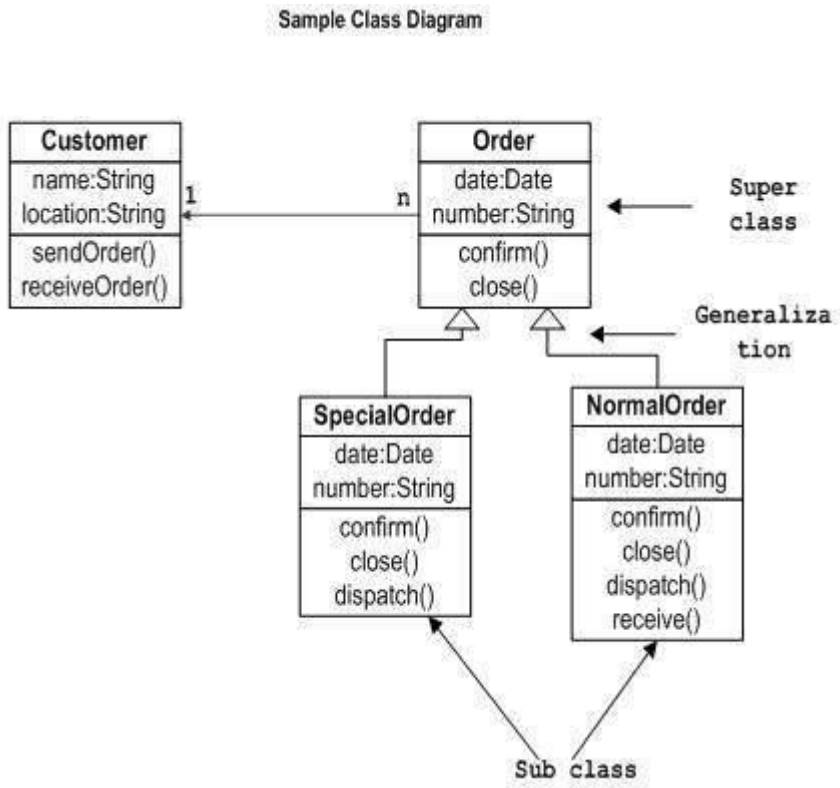
Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

The following points should be remembered while drawing a class diagram –

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified

 For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
 Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
 Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.
The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.
First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
 Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
 The two inherited classes have all the properties as the Order class. In addition, they have additional
functions like dispatch () and receive ().
The following class diagram has been drawn considering all the points mentioned above.



In a nutshell it can be said, class diagrams are used for −
 Describing the static view of the system.
 Showing the collaboration among the elements of the static view.
 Describing the functionalities performed by the system.
 Construction of software applications using object oriented languages.

Object diagram

Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
Object diagrams are used to render a set of objects and their relationships as an instance.

Purpose of Object Diagrams

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.

The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.

It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.

The purpose of the object diagram can be summarized as −

- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective

How to Draw an Object Diagram?

We have already discussed that an object diagram is an instance of a class diagram. It implies that an object diagram consists of instances of things used in a class diagram.

So both diagrams are made of same basic elements but in different form. In class diagram elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

To capture a particular system, numbers of class diagrams are limited. However, if we consider object diagrams then we can have unlimited number of instances, which are unique in nature. Only those instances are considered, which have an impact on the system.

From the above discussion, it is clear that a single object diagram cannot capture all the necessary instances or rather cannot specify all the objects of a system. Hence, the solution is –

- First, analyze the system and decide which instances have important data and association.
- Second, consider only those instances, which will cover the functionality.
- Third, make some optimization as the number of instances are unlimited.

Before drawing an object diagram, the following things should be remembered and understood clearly −

- Object diagrams consist of objects.
- The link in object diagram is used to connect objects.
- Objects and links are the two elements used to construct an object diagram.

After this, the following things are to be decided before starting the construction of the diagram −

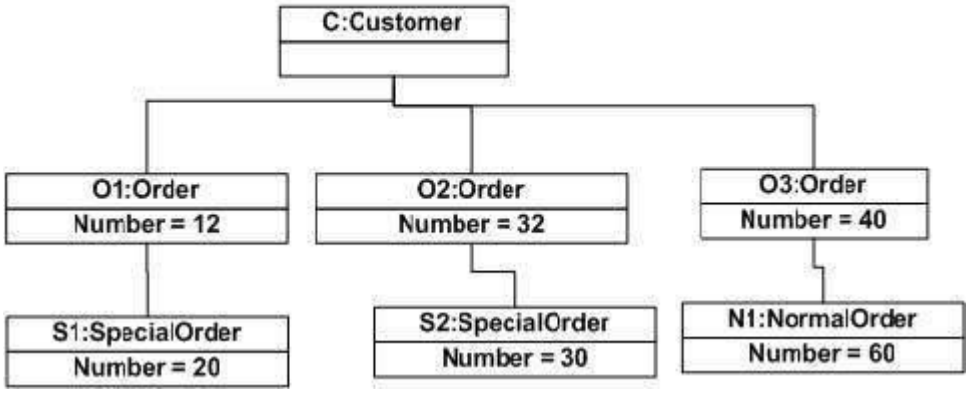
- The object diagram should have a meaningful name to indicate its purpose.
- The most important elements are to be identified.
- The association among objects should be clarified.
- Values of different elements need to be captured to include in the object diagram.
- Add proper notes at points where more clarity is required.

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.
The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.
For orders, the values are 12, 32, and 40 which implies that the objects have these values for a particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured
The same is true for special order and normal order objects which have number of orders as 20, 30, and 60. If a different time of purchase is considered, then these values will change accordingly.
The following object diagram has been drawn considering all the points mentioned above

Object diagram of an order management system



Where to Use Object Diagrams?
Object diagrams can be imagined as the snapshot of a running system at a particular moment. Let us consider an example of a running train
Now, if you take a snap of the running train then you will find a static picture of it having the following −

- A particular state which is running.
- A particular number of passengers. which will change if the snap is taken in a different time

Here, we can imagine the snap of the running train is an object having the above values. And this is true for any real-life simple or complex system.
In a nutshell, it can be said that object diagrams are used for −

- Making the prototype of a system.
- Reverse engineering.
- Modelling complex data structures.
- Understanding the system from practical perspective.

BEHAVIORAL MODELLING

Interaction Diagrams

From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.
This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purpose of both the diagrams are similar.

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages

Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

How to Draw an Interaction Diagram?

As we have already discussed, the purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment

We have two types of interaction diagrams in UML. One is the sequence diagram and the other is the collaboration diagram. The sequence diagram captures the time sequence of the message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

Following things are to be identified clearly before drawing the interaction diagram

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

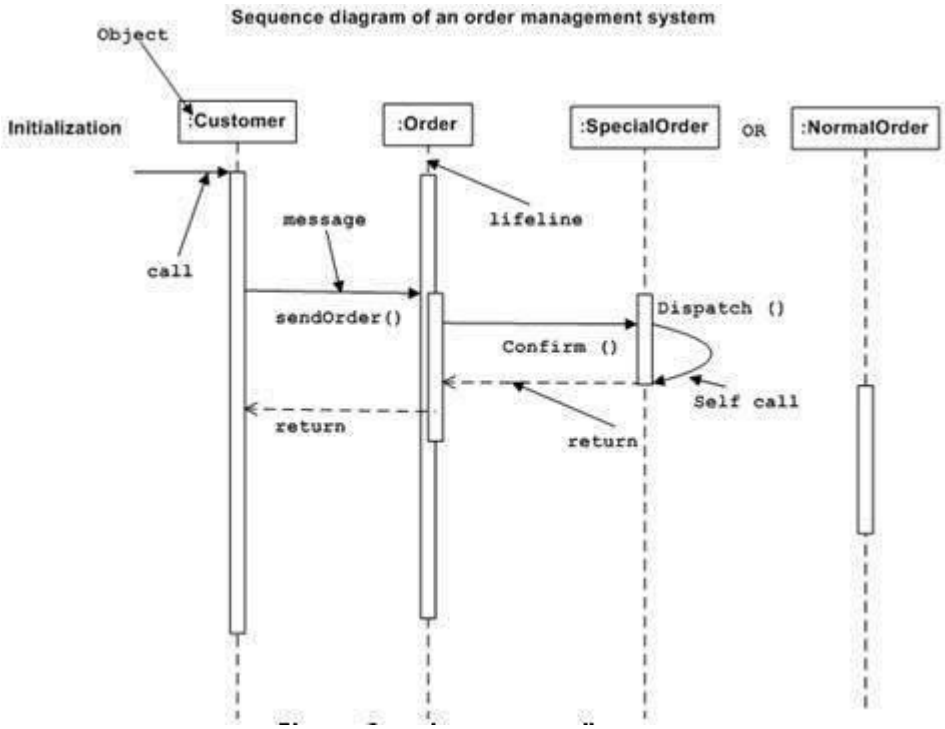
Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.

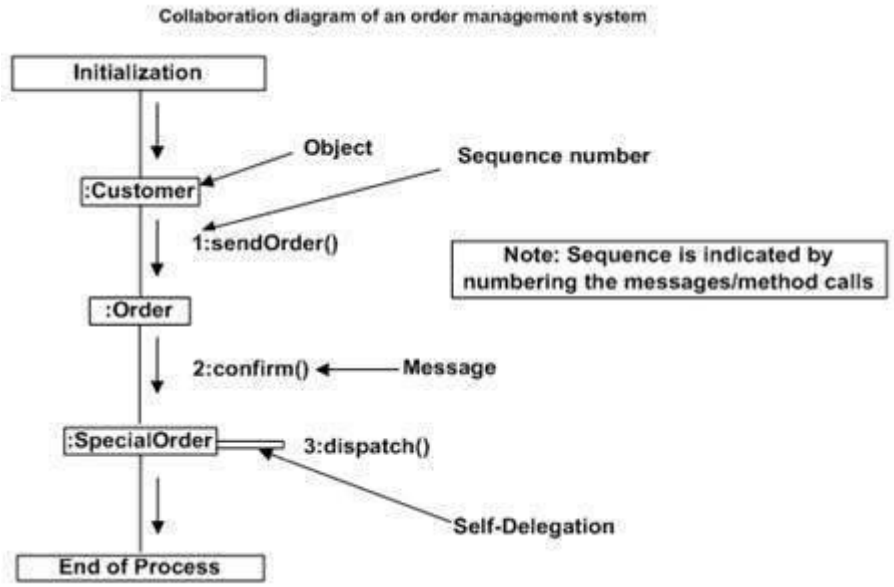


The Collaboration Diagram

The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



Where to Use Interaction Diagrams?

We have already discussed that interaction diagrams are used to describe the dynamic nature of a system. Now, we will look into the practical scenarios where these diagrams are used. To understand the practical application, we need to understand the basic nature of sequence and collaboration diagram.

The main purpose of both the diagrams are similar as they are used to capture the dynamic behavior of a system. However, the specific purpose is more important to clarify and understand.

Sequence diagrams are used to capture the order of messages flowing from one object to another. Collaboration diagrams are used to describe the structural organization of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system, so a set of diagrams are used to capture it as a whole.

Interaction diagrams are used when we want to understand the message flow and the structural organization. Message flow means the sequence of control flow from one object to another. Structural organization means the visual organization of the elements in a system.

Interaction diagrams can be used −

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

Differences between sequence diagram and collaboration diagram

Sequence Diagram	Collaboration Diagram
The sequence diagram represents the UML, which is used to visualize the sequence of calls in a system that is used to perform a specific functionality.	The collaboration diagram also comes under the UML representation which is used to visualize the organization of the objects and their interaction.
The sequence diagram are used to represent the sequence of messages that are flowing from one object to another.	The collaboration diagram are used to represent the structural organization of the system and the messages that are sent and received.
The sequence diagram is used when time sequence is main focus.	The collaboration diagram is used when object organization is main focus.
The sequence diagrams are better suited of analysis activities.	The collaboration diagrams are better suited for depicting simpler interactions of the smaller number of objects.

Use Case Diagrams

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

How to Draw a Use Case Diagram?

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

- Functionalities to be represented as use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

- The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
- Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.
The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.

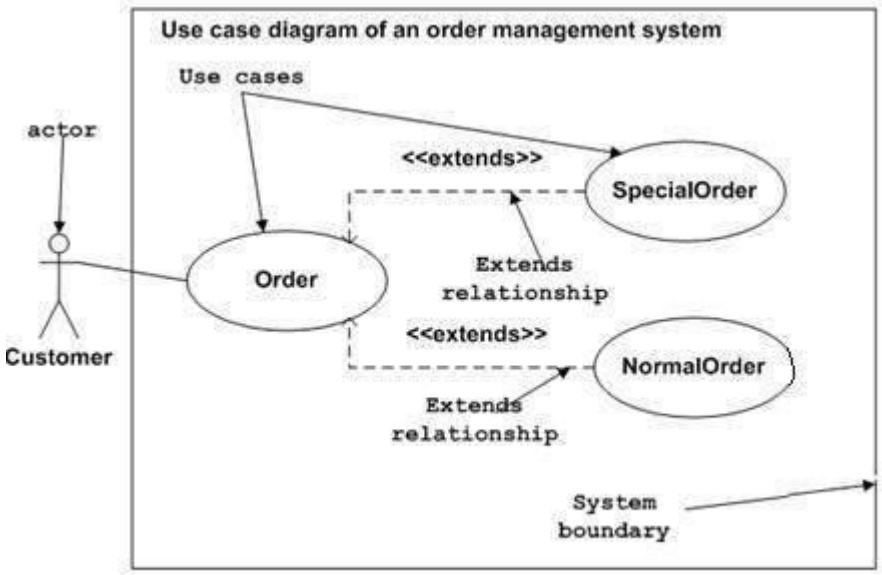


Figure: Sample Use Case diagram

Where to Use a Use Case Diagram?

As we have already discussed there are five diagrams in UML to model the dynamic view of a system. Now each and every model has some specific purpose to use. Actually these specific purposes are different angles of a running system.
To understand the dynamics of a system, we need to use different types of diagrams. Use case diagram is one of them and its specific purpose is to gather system requirements and actors.
Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known.
These diagrams are used at a very high level of design. This high level design is refined again and again to get a complete and practical picture of the system. A well-structured use case also describes the pre-condition, post condition, and exceptions. These extra elements are used to make test cases when performing the testing.
Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering.
In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.
Use case diagrams can be used for −

- Requirement analysis and high level design.
- Model the context of a system.

- Reverse engineering.
- Forward engineering.

Activity Diagrams

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

Purpose of Activity Diagrams

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

How to Draw an Activity Diagram?

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

- Activities
- Association
- Conditions
- Constraints

Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

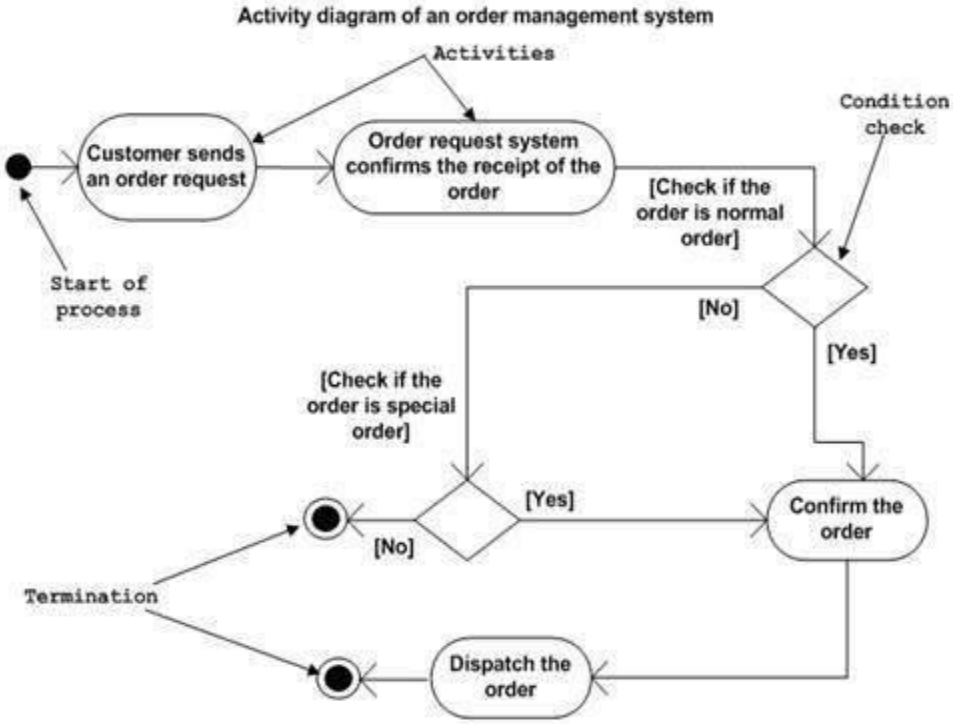
Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

- Send order by the customer
- Receipt of the order
- Confirm the order

- Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process



Where to Use Activity Diagrams?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

We will now look into the practical applications of the activity diagram. From the above discussion, it is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

- Modelling work flow by using activities.
- Modelling business requirements.
- High level understanding of the system's functionalities.
- Investigating business requirements at a later stage.

Statechart Diagrams

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

It is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

Purpose of Statechart Diagrams

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

How to Draw a Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

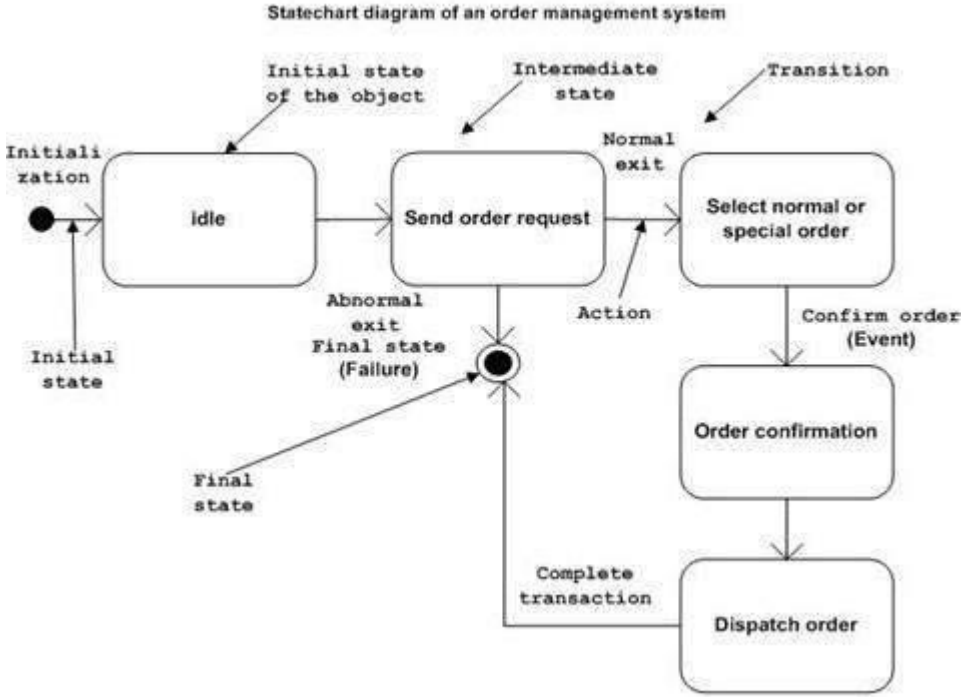
Before drawing a Statechart diagram we should clarify the following points −

- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

Following is an example of a Statechart diagram where the state of Order object is analyzed

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



Where to Use Statechart Diagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as −

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

Difference between Activity diagram and State chart diagram

- Both activity and state chart diagrams model the dynamic behavior of the system. Activity diagram is essentially a flowchart showing flow of control from activity to activity. A state chart diagram shows a state machine emphasizing the flow of control from state to state.
- An activity diagram is a special case of a state chart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state (An activity is an ongoing non-atomic execution within a state machine).

- Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. State chart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, and document the dynamics of an individual object.

Component Diagrams

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to Draw a Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

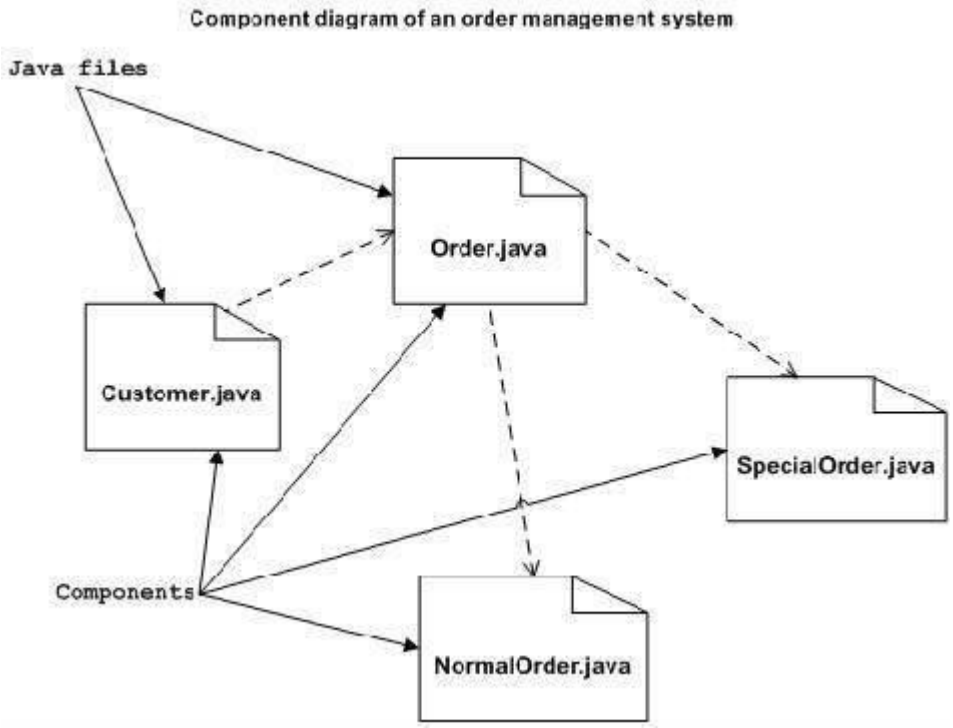
Before drawing a component diagram, the following artifacts are to be identified clearly −

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools.
- Use notes for clarifying important points.

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.
In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.
The following component diagram has been drawn considering all the points mentioned above.



Where to Use Component Diagrams?

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.
These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.
Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.
As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.
Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details
Component diagrams can be used to −

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.

Deployment Diagrams

Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.

Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose of Deployment Diagrams

The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.

Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as −

- Visualize the hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe the runtime processing nodes.

How to Draw a Deployment Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −

- Performance
- Scalability
- Maintainability
- Portability

Before drawing a deployment diagram, the following artifacts should be identified −

- Nodes
- Relationships among nodes

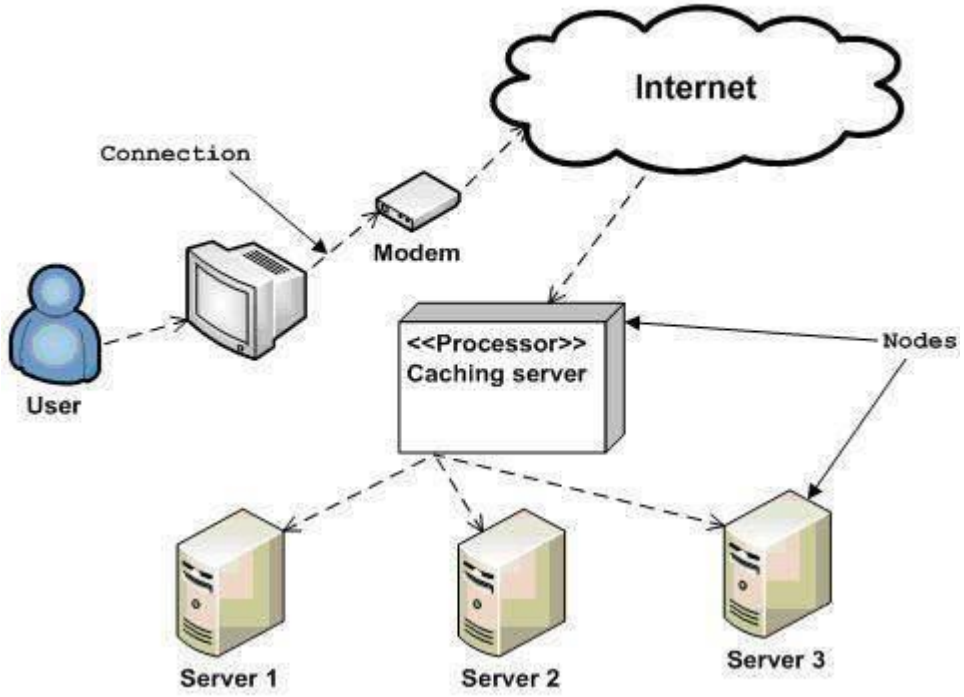
Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −

- Monitor
- Modem
- Caching server
- Server

The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.

The following deployment diagram has been drawn considering all the points mentioned above.

Deployment diagram of an order management system



Where to Use Deployment Diagrams?

Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardware), their distribution, and association.

Deployment diagrams can be visualized as the hardware components/nodes on which the software components reside.

Software applications are developed to model complex business processes. Efficient software applications are not sufficient to meet the business requirements. Business requirements can be described as the need to support the increasing number of users, quick response time, etc.

To meet these types of requirements, hardware components should be designed efficiently and in a cost-effective way.

Now-a-days software applications are very complex in nature. Software applications can be standalone, web-based, distributed, mainframe-based and many more. Hence, it is very important to design the hardware components efficiently.

Deployment diagrams can be used −

- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

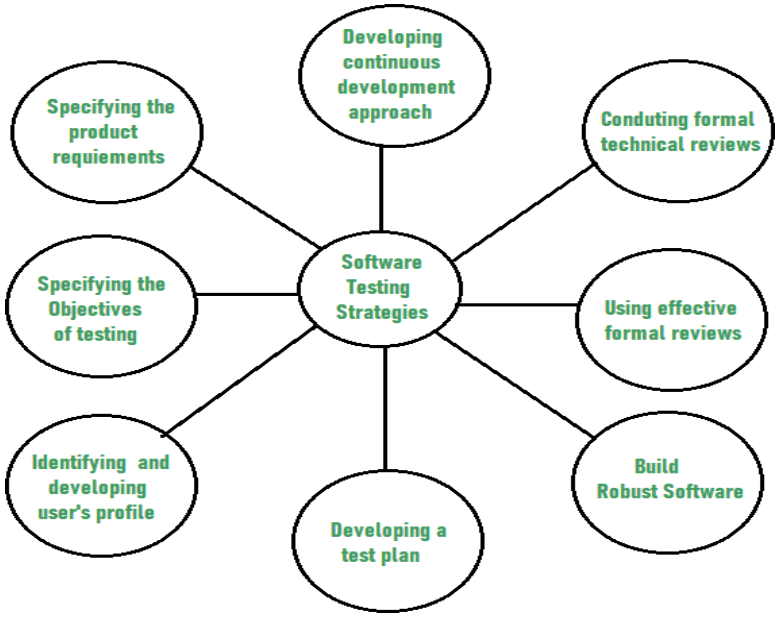
UNIT-IV

Testing Strategies: A strategic approach to software testing, test strategies for conventional software, black-box and white-box testing, validation testing, system testing, the art of debugging.
Product metrics: Software quality, metrics for analysis model, metrics for design model, metrics for source code, metrics for testing, metrics for maintenance.
Metrics for Process and Products: Software measurement, metrics for software quality.

A strategic Approach for Software testing:

Software Testing is a type of investigation to find out if there is any default or error present in the software so that the errors can be reduced or removed to increase the quality of the software and to check whether it fulfills the specifies requirements or not.

The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software. The overall strategy for testing software includes:



1. **Before testing starts, it's necessary to identify and specify the requirements of the product in a quantifiable manner.**
   Different characteristics quality of the software is there such as maintainability that means the ability to update and modify, the probability that means to find and estimate any risk, and usability that means how it can easily be used by the customers or end-users. All these characteristic qualities should be specified in a particular order to obtain clear test results without any error.

2. **Specifying the objectives of testing in a clear and detailed manner.**
   Several objectives of testing are there such as effectiveness that means how effectively the software can achieve the target, any failure that means inability to fulfill the requirements and perform functions, and the cost of defects or errors that mean the cost required to fix the error. All these objectives should be clearly mentioned in the test plan.

3. **For the software, identifying the user's category and developing a profile for each user.**
   Use cases describe the interactions and communication among different classes of users and the system to achieve the target. So as to identify the actual requirement of the users and then testing the actual use of the product.

4. **Developing a test plan to give value and focus on rapid-cycle testing.**
   Rapid Cycle Testing is a type of test that improves quality by identifying and measuring the any changes that need to be required for improving the process of software. Therefore, a test plan is an important and effective document that helps the tester to perform rapid cycle testing.

5. **Robust software is developed that is designed to test itself.**
   The software should be capable of detecting or identifying different classes of errors. Moreover, software design should allow automated and regression testing which tests the software to find out if there is any adverse or side effect on the features of software due to any change in code or program.

6. **Before testing, using effective formal reviews as a filter.**
   Formal technical reviews is technique to identify the errors that are not discovered yet. The effective technical reviews conducted before testing reduces a significant amount of testing efforts and time duration required for testing software so that the overall development time of software is reduced.

7. **Conduct formal technical reviews to evaluate the nature, quality or ability of the test strategy and test cases.**
   The formal technical review helps in detecting any unfilled gap in the testing approach. Hence, it is necessary to evaluate the ability and quality of the test strategy and test cases by technical reviewers to improve the quality of software.

8. **For the testing process, developing a approach for the continuous development.**
   As a part of a statistical process control approach, a test strategy that is already measured should be used for software testing to measure and control the quality during the development of software.

Testing Strategies for Conventional Software

- There are many strategies that can be used to test software.
- At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
    - This approach simply does not work. It will result in buggy software.
- At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.
    - This approach, although less appealing to many, can be very effective.

Types:

1) Unit Testing
2) Integration Testing
3) Validation Testing and
4) System Testing

1) Unit Testing:

Unit testing is a type of software testing where individual units or components of a software are tested. It is concerned with functional correctness of the standalone modules. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

Why Unit Testing?

Unit Testing is important because software developers sometimes try saving time doing minimal unit testing and this is myth because inappropriate unit testing leads to high cost Defect fixing during System Testing, Integration Testing and even Beta Testing after application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Here, are the key reasons to perform unit testing:

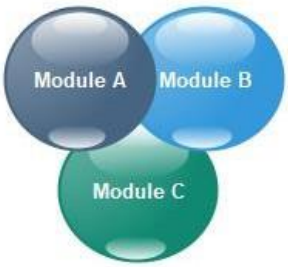
1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.

2) Integration Testing

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.

Tested in Unit Testing



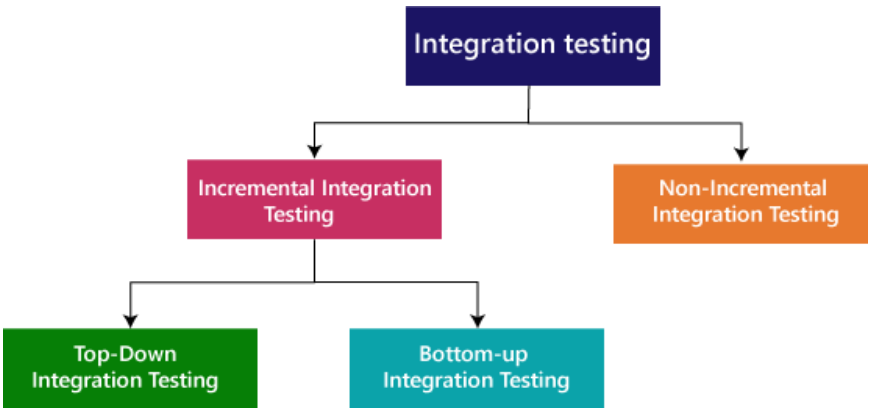
Under Integration Testing

Once all the components or modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Types of Integration Testing

Integration testing can be classified into two parts:

- **Incremental integration testing**
- **Non-incremental integration testing**



Incremental Approach

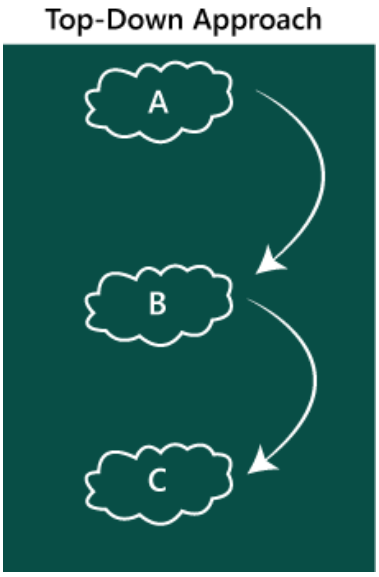
In the Incremental Approach, modules are added in ascending order one by one or according to need. The selected modules must be logically related. Generally, two or more than two modules are added and tested to determine the correctness of functions. The process continues until the successful testing of all the modules.

Incremental integration testing is carried out by further methods:

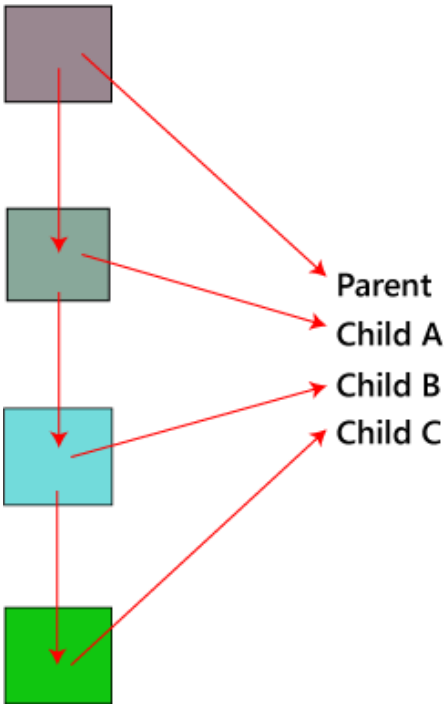
- Top-Down approach
- Bottom-Up approach

Top-Down Approach

The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first. In this type of method, we will add the modules incrementally or one by one and check the data flow in the same order.



In the top-down approach, we will be ensuring that the module we are adding is the **child of the previous one like Child C is a child of Child B** and so on as we can see in the below image:



Advantages:

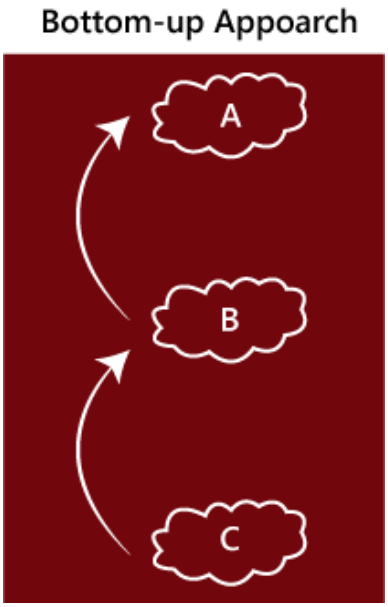
- Identification of defect is difficult.
- An early prototype is possible.

Disadvantages:

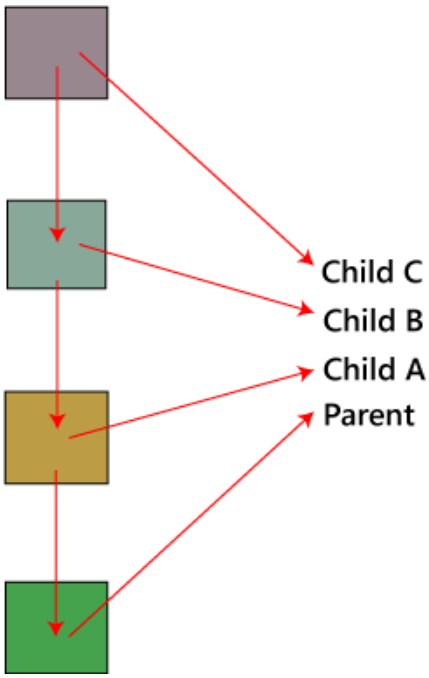
- Due to the high number of stubs, it gets quite complicated.
- Lower level modules are tested inadequately.
- Critical Modules are tested first so that fewer chances of defects.

Bottom-Up Method

The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from **bottom to the top** and check the data flow in the same order.



In the bottom-up method, we will ensure that the modules we are adding **are the parent of the previous one** as we can see in the below image:



Advantages

- Identification of defect is easy.
- Do not need to wait for the development of all the modules as it saves time.

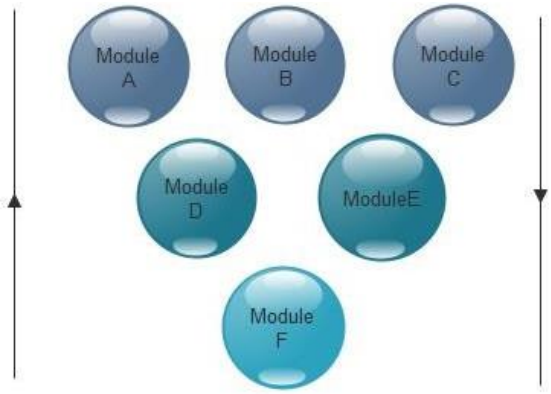
Disadvantages

- Critical modules are tested last due to which the defects can occur.
- There is no possibility of an early prototype.

In this, we have one addition approach which is known as **hybrid testing**.

Hybrid Testing Method

In this approach, both **Top-Down** and **Bottom-Up** approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



Hybrid Method

Advantages

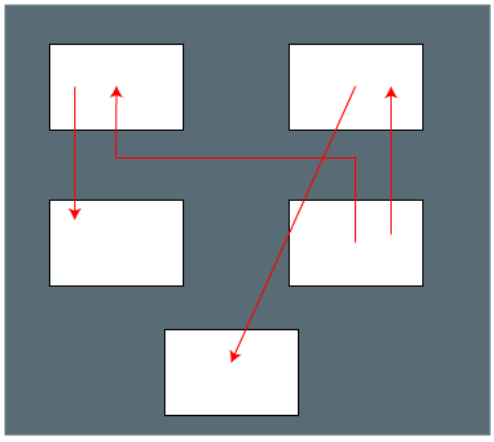
- The hybrid method provides features of both Bottom Up and Top Down methods.
- It is most time reducing method.
- It provides complete testing of all modules.

Disadvantages

- This method needs a higher level of concentration as the process carried out in both directions simultaneously.
- Complicated method.

Non- incremental integration testing

We will go for this method, when the data flow is very complex and when it is difficult to find who is a parent and who is a child. And in such case, we will create the data in any module bang on all other existing modules and check if the data is present. Hence, it is also known as the **Big bang method**.



3. Validation Testing

Verification and Validation Testing

Verification testing

Verification testing includes different activities such as business requirements, system requirements, design review, and code walkthrough while developing a product.

It is also known as static testing, where we are ensuring that "**we are developing the right product or not**". And it also checks that the developed application fulfilling all the requirements given by the client.

Validation testing

Validation testing is testing where tester performed functional and non-functional testing. Here **functional testing** includes Unit Testing (UT), Integration Testing (IT) and System Testing (ST), and **non-functional** testing includes User acceptance testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that **"we have developed the product right."** And it also checks that the software meets the business needs of the client.

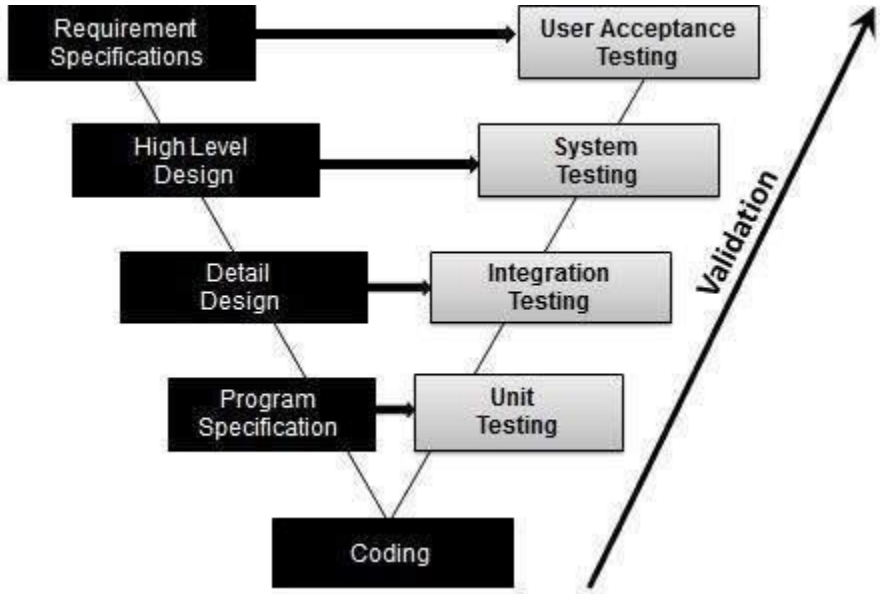
The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

It answers to the question, Are we building the right product?

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



4. System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware. System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

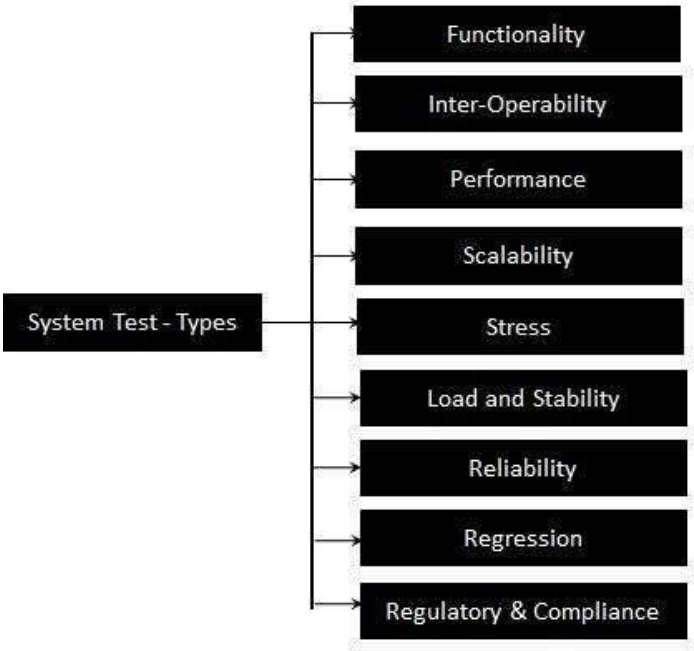
To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system.

It is **end-to-end testing** where the testing environment is similar to the production environment.

System Testing includes the following steps.

- Verification of input functions of the application to test whether it is producing the expected output or not.
- Testing of integrated software by including external peripherals to check the interaction of various components with each other.
- Testing of the whole system for End to End testing.
- Behavior testing of the application via a user's experience

Types of System Tests:



Software Testing

- Two major categories of software testing
- ❖ Black box testing
- ❖ White box testing

Black box testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

How to do Black Box Testing?

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones -

- **Functional testing** - This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** - This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Partitioning:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Analysis:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Equivalence Partitioning Testing

Equivalence Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. also called as equivalence class partitioning. It is abbreviated as ECP. It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.

An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.

Example:

The Below example best describes the equivalence class Partitioning:

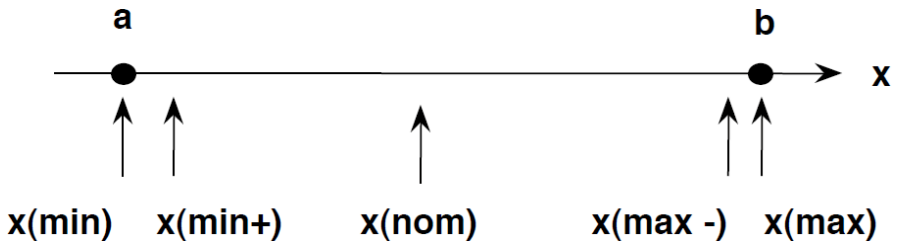
Assume that the application accepts an integer in the range 100 to 999
Valid Equivalence Class partition: 100 to 999 inclusive.
Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

Boundary Value Analysis

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:

1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



Example: Input Box should accept the Number 1 to 10

Here we will see the Boundary Value Test Cases

Test Scenario Description	Expected Outcome
Boundary Value = 0	System should NOT accept
Boundary Value = 1	System should accept
Boundary Value = 2	System should accept

Boundary Value = 9	System should accept
Boundary Value = 10	System should accept
Boundary Value = 11	System should NOT accept

Example 1: Equivalence and Boundary Value

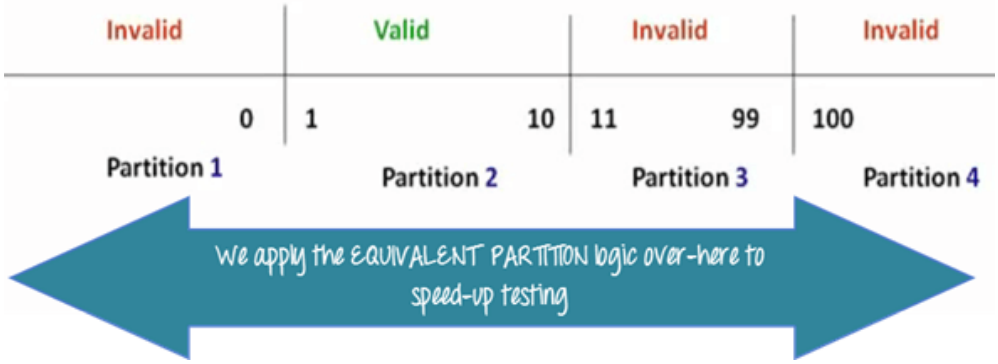
- Let's consider the behavior of Order Pizza Text Box Below
- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, **"Only 10 Pizza can be ordered"**

Order Pizza: [ ] Submit

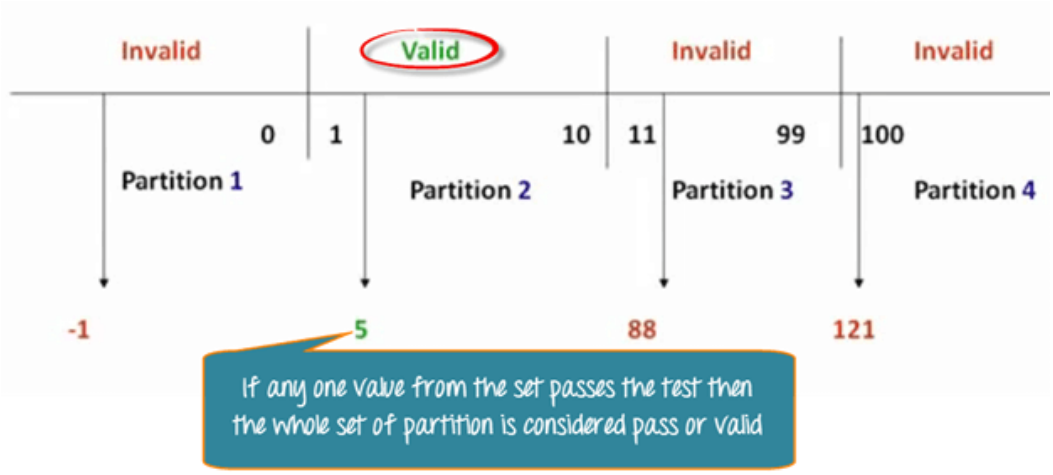
Here is the test condition

1. Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

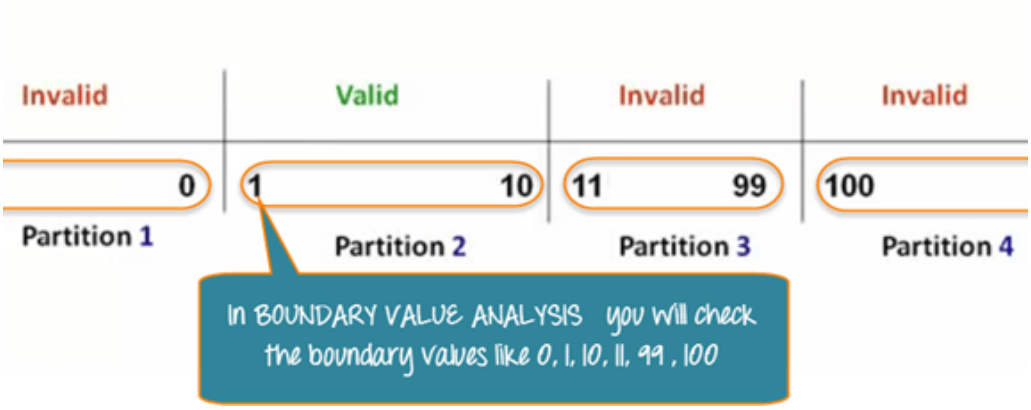
We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.



The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass**. Likewise**, if one condition in a partition fails, all other conditions in that partition will fail**.



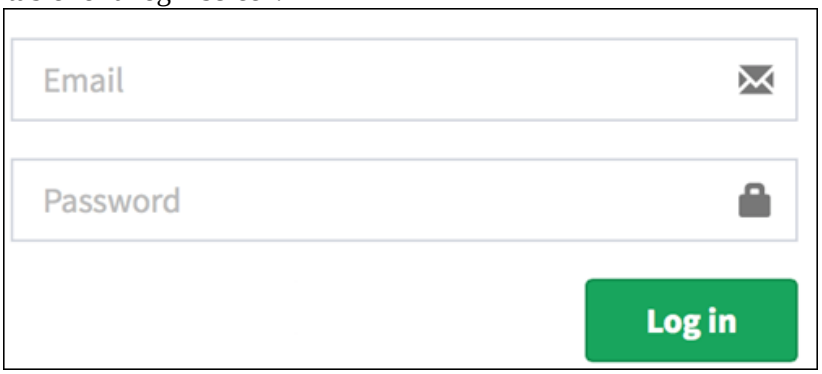
Boundary Value Analysis- in Boundary Value Analysis, you test boundaries between equivalence partitions



In our earlier example instead of checking, one value for each partition you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.
Equivalence partitioning and boundary value analysis(BVA) are closely related and can be used together at all levels of testing.

Decision Table

A **Decision Table** is a tabular representation of inputs versus rules/cases/test conditions. It is a very effective tool used for both complex software testing and requirements management. Decision table helps to check all possible combinations of conditions for testing and testers can also identify missed conditions easily. The conditions are indicated as True(T) and False(F) values.
Example 1: How to make Decision Base Table for Login Screen
Let's create a decision table for a login screen.



The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:

- T – Correct username password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

Interpretation:

- Case 1 – Username and password both were wrong. The user is shown an error message.
- Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
- Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
- Case 4 – Username and password both were correct, and the user navigated to homepage

While converting this to test case, we can create 2 scenarios,

- Enter correct username and correct password and click on login, and the expected result will be the user should be navigated to homepage

And one from the below scenario

- Enter wrong username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter correct username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter wrong username and correct password and click on login, and the expected result will be the user should get an error message

White Box Testing:

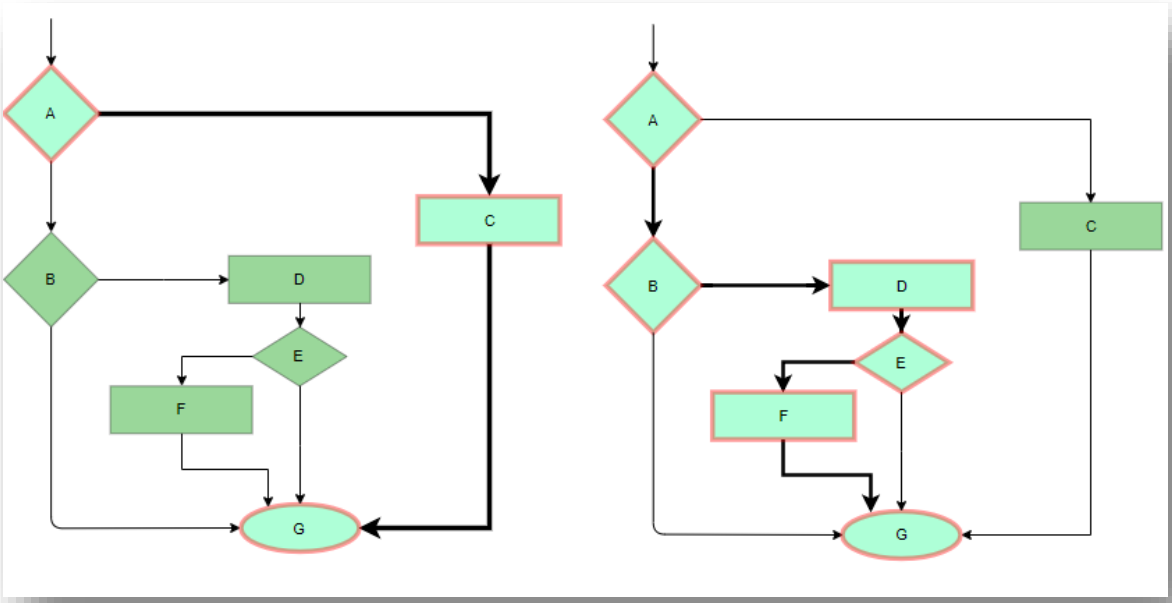
White box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

White Box Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

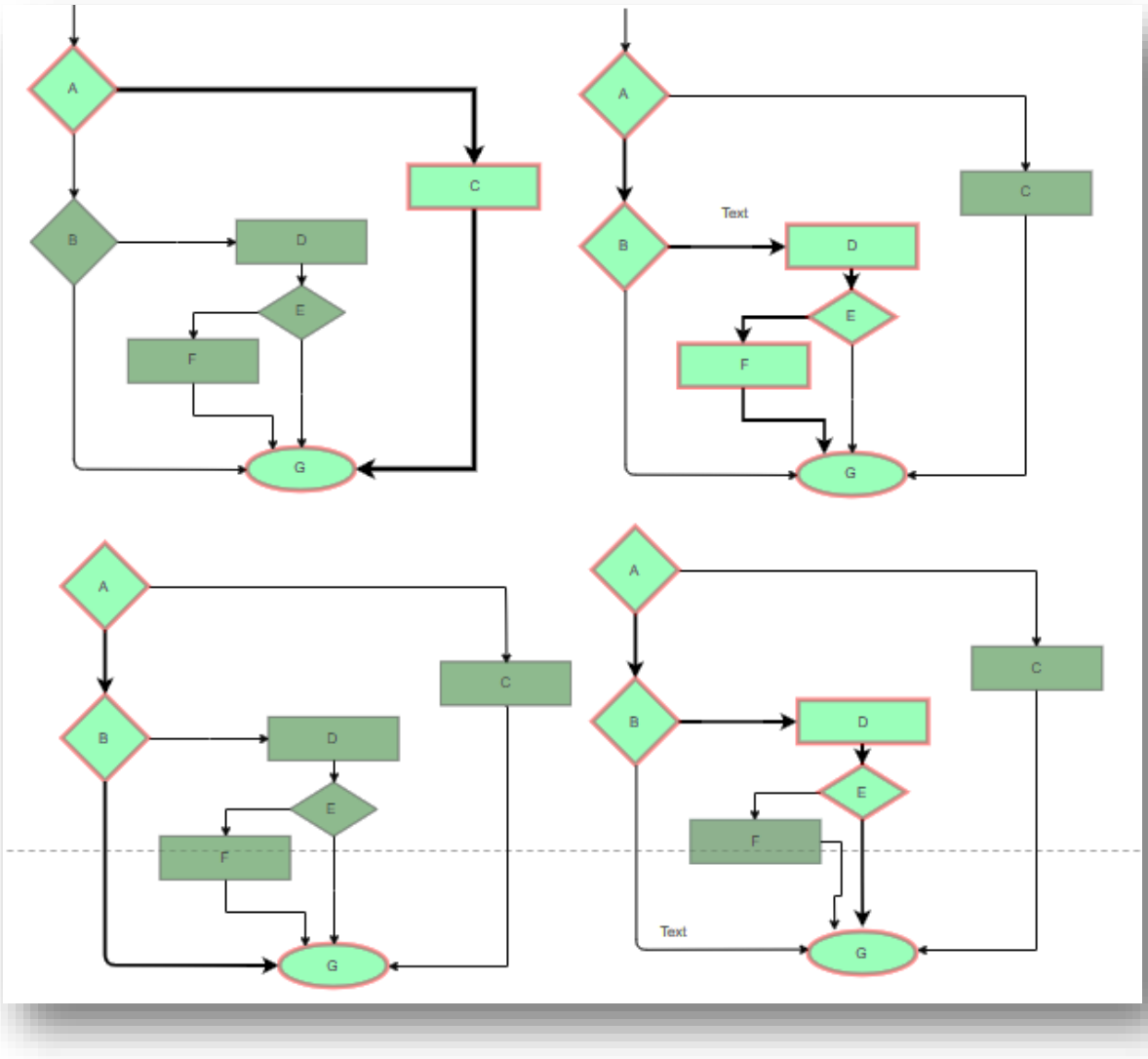
Statement coverage:

In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



Statement Coverage Example

Branch Coverage: In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.



4 test cases required such that all branches of all decisions are covered, i.e, all edges of flowchart are covered

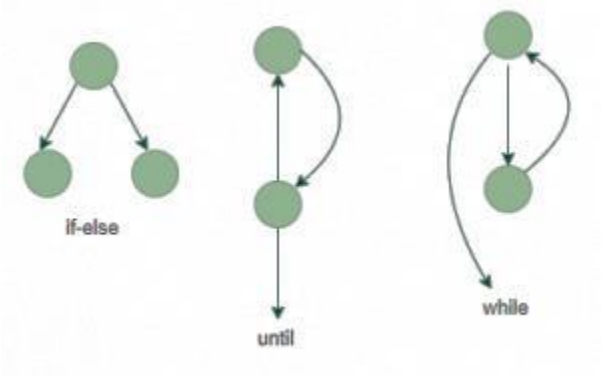
Basis Path Testing: In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

Steps:

1. Make the corresponding control flow graph
2. Calculate the cyclomatic complexity
3. Find the independent paths
4. Design test cases corresponding to each independent path

Flow graph notation: It is a directed graph consisting of nodes and edges. Each node represents a sequence of statements, or a decision point. A predicate node is the one that represents a decision point that

contains a condition after which the graph splits. Regions are bounded by nodes and edges.



Cyclomatic Complexity: It is a measure of the logical complexity of the software and is used to define the number of independent paths. For a graph G, V(G) is its cyclomatic complexity.

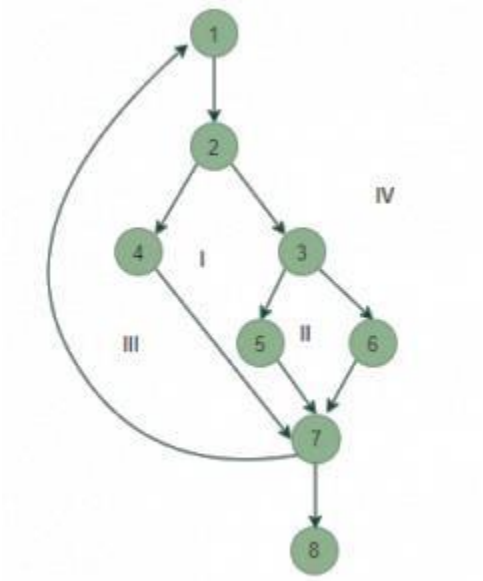
Calculating V(G):

V(G) = P + 1, where P is the number of predicate nodes in the flow graph

V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes

V(G) = Number of non-overlapping regions in the graph

Example:



V(G) = 4 (Using any of the above formulae)

No of independent paths = 4

- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

Loop Testing: Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

Simple loops: For simple loops of size n, test cases are designed that:

- Skip the loop entirely
- Only one pass through the loop
- 2 passes
- m passes, where m < n
- n-1 ans n+1 passes

Nested loops: For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

Concatenated loops: Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

The art of Debugging

Debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

Debugging Process: Steps involved in debugging are:

Problem identification and report preparation.

Assigning the report to software engineer to the defect to verify that it is genuine.

Defect Analysis using modeling, documentations, finding and testing candidate flaws, etc.

Defect Resolution by making required changes to the system.

Validation of corrections.

Debugging Strategies:

1. Study the system for the larger duration in order to understand the system. It helps debugger to construct different representations of systems to be debugging depends on the need. Study of the system is also done actively to find recent changes made to the software.
2. Backwards analysis of the problem which involves tracing the program backward from the location of failure message in order to identify the region of faulty code. A detailed study of the region is conducting to find the cause of defects.
3. Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused to find the defect.
4. Using the past experience of the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

Debugging Tools:

Debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command line interfaces. Examples of automated debugging tools include code based tracers, profilers, interpreters, etc. Some of the widely used debuggers are:

Radare2

WinDbg

Valgrind

Difference Between Debugging and Testing:

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing like unit testing, integration testing, alpha and beta testing, etc.

Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Product metrics

In software development process, a working product is developed at the end of each successful phase. Each product can be measured at any stage of its development. Metrics are developed for these products so that they can indicate whether a product is developed according to the user requirements. If a product does not meet user requirements, then the necessary actions are taken in the respective phase.

Product metrics help software engineer to detect and correct potential problems before they result in catastrophic defects. In addition, product metrics assess the internal product attributes in order to know the efficiency of the following.

- Analysis, design, and code model
- Potency of test cases
- Overall quality of the software under development.

Various metrics formulated for products in the development process are listed below.

- **Metrics for analysis model:** These address various aspects of the analysis model such as system functionality, system size, and so on.
- **Metrics for design model:** These allow software engineers to assess the quality of design and include architectural design metrics, component-level design metrics, and so on.
- **Metrics for source code:** These assess source code complexity, maintainability, and other characteristics.
- **Metrics for testing:** These help to design efficient and effective test cases and also evaluate the effectiveness of testing.
- **Metrics for maintenance:** These assess the stability of the software product.

Metrics for the Analysis Model

There are only a few metrics that have been proposed for the analysis model. However, it is possible to use metrics for project estimation in the context of the analysis model. These metrics are used to examine the analysis model with the objective of predicting the size of the resultant system. Size acts as an indicator of increased coding, integration, and testing effort; sometimes it also acts as an indicator of complexity involved in the software design. Function point and lines of code are the commonly used methods for size estimation.

Function Point (FP) Metric

The function point metric, which was proposed by A.J Albrecht, is used to measure the functionality delivered by the system, estimate the effort, predict the number of errors, and estimate the number of components in the system.

Function point is derived by using a relationship between the complexity of software and the information domain value. Information domain values used in function point include the number of external inputs, external outputs, external inquires, internal logical files, and the number of external interface files.

Lines of Code (LOC)

Lines of code (LOC) is one of the most widely used methods for size estimation. LOC can be defined as the number of delivered lines of code, excluding comments and blank lines. It is highly dependent on the programming language used as code writing varies from one programming language to another. Fur example, lines of code written (for a large program) in assembly language are more than lines of code written in C++. From LOC, simple size-oriented metrics can be derived such as errors per KLOC

(thousand lines of code), defects per KLOC, cost per KLOC, and so on. LOC has also been used to predict program complexity, development effort, programmer performance, and so on. For example, Hasltead proposed a number of metrics, which are used to calculate program length, program volume, program difficulty, and development effort.

Metrics for Specification Quality

To evaluate the quality of analysis model and requirements specification, a set of characteristics has been proposed. These characteristics include specificity, completeness, correctness, understandability, verifiability, internal and external consistency, &achievability, concision, traceability, modifiability, precision, and reusability. Most of the characteristics listed above are qualitative in nature. However, each of these characteristics can be represented by using one or more metrics. For example, if there are n_r requirements in a specification, then n_r can be calculated by the following equation.

$$n_r = n_f + n_{rf}$$

Where

n_f = number of functional requirements

n_{nf} = number of non-functional requirements.

In order to determine the specificity of requirements, a metric based on the consistency of the reviewer's understanding of each requirement has been proposed. This metric is represented by the following equation.

$$Q_1 = n_{ui}/n_r$$

Where

n_{ui} = number of requirements for which reviewers have same understanding Q_1 = specificity.

Ambiguity of the specification depends on the value of Q. If the value of Q is close to 1 then the probability of having any ambiguity is less.

Completeness of the functional requirements can be calculated by the following equation.

$$Q_2 = n_u / [n_j * n_s]$$

Where

n_u = number of unique function requirements

n_i = number of inputs defined by the specification n_s = number of specified state.

Q_2 in the above equation considers only functional requirements and ignores non-functional requirements. In order to consider non-functional requirements, it is necessary to consider the degree to which requirements have been validated. This can be represented by the following equation.

$$Q_3 = n_c / [n_c + n_{nv}]$$

Where

n_c= number of requirements validated as correct

n_{nv}= number of requirements, which are yet to be validated.

Metrics for Software Design

The success of a software project depends largely on the quality and effectiveness of the software design. Hence, it is important to develop software metrics from which meaningful indicators can be derived. With the help of these indicators, necessary steps are taken to design the software according to the user requirements. Various design metrics such as architectural design metrics, component-level design metrics, user-interface design metrics, and metrics for object-oriented design are used to indicate the complexity, quality, and so on of the software design.

Architectural Design Metrics

These metrics focus on the features of the program architecture with stress on architectural structure and effectiveness of components (or modules) within the architecture. In architectural design metrics, three software design complexity measures are defined, namely, structural complexity, data complexity, and system complexity. In hierarchical architectures (call and return architecture), say module 'j', structural complexity is calculated by the following equation.

$S(j) = f^2_{out}(j)$

Where

$f_{out}(j)$ = fan-out of module 'j' [Here, fan-out means number of modules that are subordinating module j]. Complexity in the internal interface for a module 'j' is indicated with the help of data complexity, which is calculated by the following equation.

$D(j) = V(j) / [f_{out}(j)+l]$

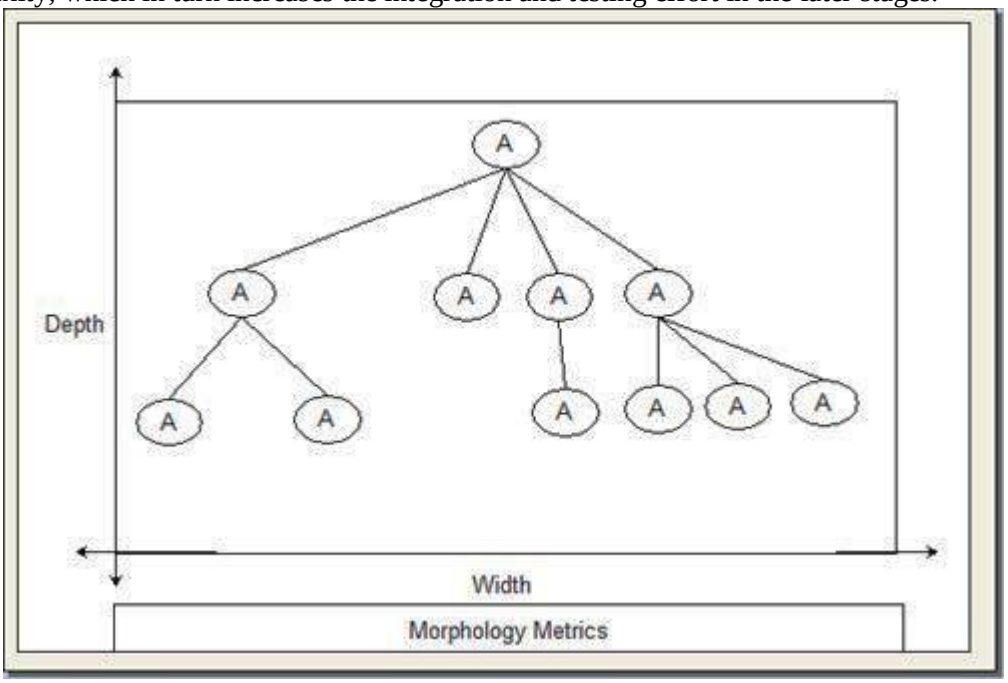
Where

V(j) = number of input and output variables passed to and from module 'j'.

System complexity is the sum of structural complexity and data complexity and is calculated by the following equation.

$C(j) = S(j) + D(j)$

The complexity of a system increases with increase in structural complexity, data complexity, and system complexity, which in turn increases the integration and testing effort in the later stages.



In addition, various other metrics like simple morphology metrics are also used. These metrics allow comparison of different program architecture using a set of straightforward dimensions. A metric can be developed by referring to call and return architecture. This metric can be defined by the following equation.

$$Size = n+a$$

Where

n = number of nodes a= number of arcs.

For example, there are 11 nodes and 10 arcs. Here, Size can be calculated by the following equation.

$$Size = n+a = 11+10+21.$$

Depth is defined as the longest path from the top node (root) to the leaf node and width is defined as the maximum number of nodes at any one level.

Coupling of the architecture is indicated by arc-to-node ratio. This ratio also measures the connectivity density of the architecture and is calculated by the following equation.

$$r=a/n$$

Quality of software design also plays an important role in determining the overall quality of the software. Many software quality indicators that are based on measurable design characteristics of a computer program have been

proposed. One of them is Design Structural Quality Index (DSQI), which is derived from the information obtained from data and architectural design. To calculate DSQI, a number of steps are followed, which are listedbelow.

1. To calculate DSQI, the following values must be determined.

- Number of components in program architecture (S_1)
- Number of components whose correct function is determined by the Source of input data (S_2)
- Number of components whose correct function· depends on previous processing (S_3)
- Number of database items (S_4)
- Number of different database items (S_5)
- Number of database segments (S_6)
- Number of components having single entry and exit (S_7).

2. Once all the values from S_1 to S_7 are known, some intermediate values are calculated, which are listed below. **Program structure (D_1):** If discrete methods are used for developing architectural design then $D_1 = 1$, else $D_1 = 0$
Module independence (D_2): $D_2 = 1-(S_2/S_1)$
Modules not dependent on prior processing (D_3): $D_3 = 1-(S_3/S_1)$
Database size (D_4): $D_4 = 1-(S_5/S_4)$
Database compartmentalization (D_5): $D_5 = 1-(S_6/S_4)$
Module entrance/exit characteristic (D_6): $D_6 = 1-(S_7/S_1)$
3. Once all the intermediate values are calculated, OSQI is calculated by the following equation.

$$DSQI = \sum W_i D_i$$

Where
$i = 1$ to 6
$\sum W_i = 1$ (W_i is the weighting of the importance of intermediate values).
In conventional software, the focus of component – level design metrics is on the internal characteristics of the software components; The software engineer can judge the quality of the component-level design by measuring module cohesion, coupling and complexity; Component-level design metrics are applied after procedural design is final. Various metrics developed for component-level design are listed below.

- **Cohesion metrics:** Cohesiveness of a module can be indicated by the definitions of the following five concepts and measures.
- **Data slice:** Defined as a backward walk through a module, which looks for values of data that affect the state of the module as the walk starts
- **Data tokens:** Defined as a set of variables defined for a module
- **Glue tokens:** Defined as a set of data tokens, which lies on one or more data slice
- **Superglue tokens:** Defined as tokens, which are present in every data slice in the module
- **Stickiness:** Defined as the stickiness of the glue token, which depends on the number of data slices that it binds.
- **Coupling Metrics:** This metric indicates the degree to which a module is connected to other modules, global data and the outside environment. A metric for module coupling has been proposed, which includes data and control flow coupling, global coupling, and environmental coupling.
    - Measures defined for data and control flow coupling are

listed below. d_i = total number of input data parameters
c_i = total number of input control
parameters d_o= total number of
output data parameters c_o= total
number of output control
parameters

\$1§ Measures defined for global coupling are

listed below. g_d= number of global variables utilized as data
g_c = number of global variables utilized as control

\$1§ Measures defined for environmental coupling are

listed below. w = number of modules called
r = number of modules calling the modules under consideration
By using the above mentioned measures, module-coupling indicator (m_c) is calculated by using the

following equation.

$$m_c = K/M$$

Where
K = proportionality constant

$$M = d_i + (a*c_i) + d_o + (b*c_o) + g_d + (c*g_c) + w + r.$$

Note that K, a, b, and c are empirically derived. The values of m_c and overall module coupling are inversely proportional to each other. In other words, as the value of m_c increases, the overall module coupling decreases.

Complexity Metrics: Different types of software metrics can be calculated to ascertain the complexity of program control flow. One of the most widely used complexity metrics for ascertaining the complexity of the program is cyclomatic complexity.

Many metrics have been proposed for user interface design. However, layout appropriateness metric and cohesion metric for user interface design are the commonly used metrics. Layout Appropriateness (LA) metric is an important metric for user interface design. A typical Graphical User Interface (GUI) uses many layout entities such as icons, text, menus, windows, and so on. These layout entities help the users in completing their tasks easily. In to complete a given task with the help of GUI, the user moves from one layout entity to another.

Appropriateness of the interface can be shown by absolute and relative positions of each layout entities, frequency with which layout entity is used, and the cost of changeover from one layout entity to another.

Cohesion metric for user interface measures the connection among the onscreen contents. Cohesion for user interface becomes high when content presented on the screen is from a single major data object (defined in the analysis model). On the other hand, if content presented on the screen is from different data objects, then cohesion for user interface is low.

In addition to these metrics, the direct measure of user interface interaction focuses on activities like measurement of time required in completing specific activity, time required in recovering from an error condition, counts of specific operation, text density, and text size. Once all these measures are collected, they are organized to form meaningful user interface metrics, which can help in improving the quality of the user interface.

Metrics for Object-oriented Design

In order to develop metrics for object-oriented (OO) design, nine distinct and measurable characteristics of OO design are considered, which are listed below.

- **Complexity:** Determined by assessing how classes are related to each other
- **Coupling:** Defined as the physical connection between OO design elements
- **Sufficiency:** Defined as the degree to which an abstraction possesses the features required of it
- **Cohesion:** Determined by analyzing the degree to which a set of properties that the class possesses is part of the problem domain or design domain
- **Primitiveness:** Indicates the degree to which the operation is atomic
- **Similarity:** Indicates similarity between two or more classes in terms of their structure, function, behavior, or purpose
- **Volatility:** Defined as the probability of occurrence of change in the OO design
- **Size:** Defined with the help of four different views, namely, population, volume, length, and functionality. Population is measured by calculating the total number of OO entities, which can be in the form of classes or operations. Volume measures are collected dynamically at any given point of time. Length is a measure of interconnected designs such as depth of inheritance tree. Functionality indicates the value rendered to the user by the OO application.

Metrics for Coding

Halstead proposed the first analytic laws for computer science by using a set of primitive measures, which can be derived once the design phase is complete and code is generated. These measures are listed below.

n_1 = number of distinct operators in a program n_2 = number of distinct operands in a program N_1 = total number of operators

N_2= total number of operands.
By using these measures, Halstead developed an expression for overall program length, program volume, program difficulty, development effort, and so on.
Program length (N) can be calculated by using the following equation. $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$.
Program volume (V) can be calculated by using the following equation. $V = N \log_2 (n_1+n_2)$.
Note that program volume depends on the programming language used and represents the volume of information (in bits) required to specify a program. Volume ratio (L) can be calculated by using the following equation.

$$L = \frac{\text{Volume of the most compact form of a program}}{\text{Volume of the actual program}}$$

Where, value of L must be less than 1. Volume ratio can also be calculated by using the following equation.

$$L = (2/n_1) * (n_2/N_2).$$

Program difficulty level (D) and effort (E)can be calculated by using the following equations.

$$D = (n_1/2) * (N_2/n_2).$$

$$E = D * V.$$

Metrics for Software Testing

Majority of the metrics used for testing focus on testing process rather than the technical characteristics of test. Generally, testers use metrics for analysis, design, and coding to guide them in design and execution of test cases. Function point can be effectively used to estimate testing effort. Various characteristics like errors discovered, number of test cases needed, testing effort, and so on can be determined by estimating the number of function points in the current project and comparing them with any previous project.
Metrics used for architectural design can be used to indicate how integration testing can be carried out. In addition, cyclomatic complexity can be used effectively as a metric in the basis-path testing to determine the number of test cases needed.
Halstead measures can be used to derive metrics for testing effort. By using program volume (V) and program level (PL),Halstead effort (e)can be calculated by the following equations.

$$e = V/ PL$$

Where

$$PL = 1/ [(n_1/2) * (N_2/n_2)] \quad \ldots (1)$$

For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation.
Percentage of testing effort (z) = $e(z)/\sum e(i)$
Where, e(z) is calculated for module z with the help of equation (1). Summation in the denominator is the sum of Halstead effort (e) in all the modules of the system.
For developing metrics for object-oriented (OO) testing, different types of design metrics that have a direct impact on the testability of object-oriented system are considered. While developing metrics for OO testing, inheritance and encapsulation are also considered. A set of metrics proposed for OO testing is listed below.

- **Lack of cohesion in methods (LCOM):** This indicates the number of states to be tested. LCOM indicates the number of methods that access one or more same attributes. The value of LCOM is 0, if no methods access the same attributes. As the value of LCOM increases, more states need to be tested.
- **Percent public and protected (PAP):** This shows the number of class attributes, which are public or protected. Probability of adverse effects among classes increases with increase in value of PAP as public and protected attributes lead to potentially higher coupling.
- **Public access to data members (PAD):** This shows the number of classes that can access attributes of another class. Adverse effects among classes increase as the value of PAD increases.
- **Number of root classes (NOR):** This specifies the number of different class hierarchies, which are described in the design model. Testing effort increases with increase in NOR.

- **Fan-in (FIN):** This indicates multiple inheritances. If value of FIN is greater than 1, it indicates that the class inherits its attributes and operations from many root classes. Note that this situation (where FIN> 1) should be avoided.

Metrics for Software Maintenance

For the maintenance activities, metrics have been designed explicitly. IEEE have proposed Software Maturity Index (SMI), which provides indications relating to the stability of software product. For calculating SMI, following parameters are considered.

- Number of modules in current release (M_T)
- Number of modules that have been changed in the current release (F_e)
- Number of modules that have been added in the current release (F_a)
- Number of modules that have been deleted from the current release (F_d)

Once all the parameters are known, SMI can be calculated by using the following equation.

$$SMI = [M_T – (F_a + F_e + F_d)]/M_T.$$

Note that a product begins to stabilize as SMI reaches 1.0. SMI can also be used as a metric for planning software maintenance activities by developing empirical models in order to know the effort required for maintenance.

Metrics for Process and Products

Software Measurement: A measurement is a manifestation of the size, quantity, amount or dimension of a particular attributes of a product or process.

It is an authority within software engineering. Software measurement process is defined and governed by ISO Standard.

Need of Software Measurement:

Software is measured to:

Create the quality of the current product or process.

Anticipate future qualities of the product or process.

Enhance the quality of a product or process.

Regulate the state of the project in relation to budget and schedule.

Classification of Software Measurement:

There are 2 types of software measurement:

Direct Measurement:

In direct measurement the product, process or thing is measured directly using standard scale.

Indirect Measurement:

In indirect measurement the quantity or quality to be measured is measured using related parameter i.e. by use of reference.

Metrics:

A metrics is a measurement of the level that any impute belongs to a system product or process. There are 4 functions related to software metrics:

Planning

Organizing

Controlling

Improving

Characteristics of software Metrics:

Quantitative:

Metrics must possess quantitative nature. It means metrics can be expressed in values.

Understandable:

Metric computation should be easily understood, the method of computing metric should be clearly defined.

Applicability:

Metrics should be applicable in the initial phases of development of the software.

Repeatable:

The metric values should be same when measured repeatedly and consistent in nature.

Economical:

Computation of metric should be economical.
Language Independent:
Metrics should not depend on any programming language.

Metrics for software quality:

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.

Software quality metrics can be further divided into three categories −

Product quality metrics
In-process quality metrics
Maintenance quality metrics

Product Quality Metrics

This metrics include the following −

Mean Time to Failure
Defect Density
Customer Problems
Customer Satisfaction

- **Mean Time to Failure**

It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

- **Defect Density**

It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

- **Customer Problems**

It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

The problems metric is usually expressed in terms of Problems per User-Month (PUM). PUM = Total Problems that customers reported (true defect and non-defect oriented problems) for a time period + Total number of license months of the software during
the period Where,

Number of license-month of the software = Number of install license of the software × Number of months in the calculation period

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

- **Customer Satisfaction**

Customer satisfaction is often measured by customer survey data through the five-point scale −

Very satisfied
Satisfied
Neutral
Dissatisfied
Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −

Percent of completely satisfied customers
Percent of satisfied customers
Percent of dis-satisfied customers
Percent of non-satisfied customers Usually, this percent satisfaction is used.

In-process Quality Metrics

In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

Defect density during machine testing
Defect arrival pattern during machine testing
Phase-based defect removal pattern

Defect removal effectiveness

Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.

This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing

The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following −

- The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.
- The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.
- The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern

This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.

Defect removal effectiveness

It can be defined as follows –

$$DRE = \frac{\text{Defect removed during a development phase}}{\text{Defects latent in the product}} \times 100\%$$

This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

Maintenance Quality Metrics

Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog and backlog management index

Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

$$BMI = \frac{Number\ of\ problems\ closed\ during\ the\ month}{Number\ of\ problems\ arrived\ during\ the\ month} \times 100\%$$

If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

Fix response time and fix responsiveness

The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fix response time leads to customer satisfaction.

The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.

Percent delinquent fixes

It is calculated as follows −

$$Percent\ Delinquent\ Fixes = \frac{Number\ of\ fixes\ that\ exceeded\ the\ response\ time\ criteria\ by\ ceverity\ level}{Number\ of\ fixes\ delivered\ in\ a\ specified\ time} \times 100\%$$

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.

Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

UNIT-V

Risk management: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM, RMMM plan.

Quality Management: Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000 quality standards.

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

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

(1) Building a excellent product or system that no one really wants (market risk),

(2) Building a product that no longer fits into the overall business strategy for the company (strategic risk),

(3) Building a product that the sales force doesn't understand how tosell,

(4) Losing the support of senior management due to a change in focus or a change in people (management risk), and

(5) Losing budgetary or personnel commitment (budget risks).

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.

2) RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan. There are two distinct types of risks.

1) Generic risks and
2) 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 ormodified.
- ***Business impact***—risks associated with constraints imposed by management or the marketplace.
- ***Customer characteristics***—risks associated with the sophistication of the customer andthe 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.

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be Implemented?
10. Is the number of people on the project team adequate to do the job?
11. 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.

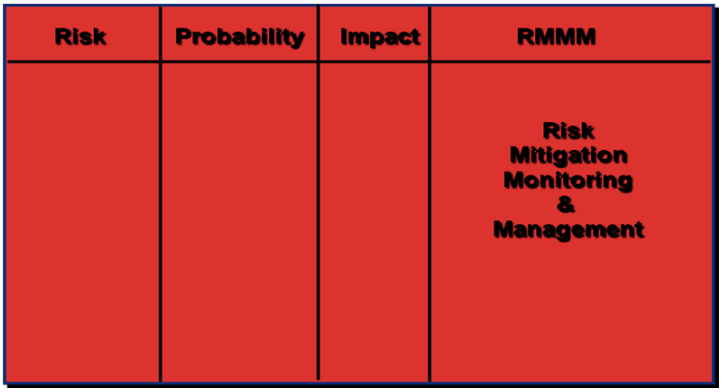
3) RISK PROJECTION

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:

(1) establish a scale that reflects the perceived likelihood of a risk,
(2) delineate the consequences of the risk,
(3) estimate the impact of the risk on the project and the product, and
(4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table Building a Risk

- 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 overalldistribution.
- ✓ 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.

4) 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.

5) 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, and 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.

6) 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:

1) to assess whether predicted risks do, in fact, occur;
2) to ensure that risk aversion steps defined for the risk are being properly applied; and
3) to collect information that can be used for future risk analysis.

QUALITY MANAGEMENT

1) QUALITY CONCEPTS:

Quality management encompasses

(1) a quality management approach,
(2) effective software engineering technology (methods and tools),
(3) formal technical reviews that are applied throughout the softwareprocess,
(4) a multitiered testing strategy,
(5) control of software documentation and the changes made to it,
(6) a procedure to ensure compliance with software development standards (when applicable), and
(7) 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:

1) Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2) Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surelyresult.

3) 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 byall 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.

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)*

4) 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

(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verify that the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and
(5) 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
(1) accept the product without further modification,
(2) reject the product due to severe errors (once corrected, another review must be performed), or
(3) 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:
1. What was reviewed?
2. Who reviewed it?
3. 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:

1. ***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.
2. ***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.
3. ***Limit debate and rebuttal***. When an issue is raised by a reviewer, there may not be universal agreement on its impact.
4. ***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.
5. ***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 isrecorded.
6. ***Limit the number of participants and insist upon advance preparation***. Keep the number of people involved to the necessary minimum.
7. ***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.
8. ***Allocate resources and schedule time for FTRs***. For reviews to be effective, they should be scheduled as a task during the software engineering process
9. ***Conduct meaningful training for all reviewers.*** To be effective all review participants should receive some formal training.
10. ***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:

1. Information about software defects is collected and categorized.
2. 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).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the 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.

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".

7.1 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.

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

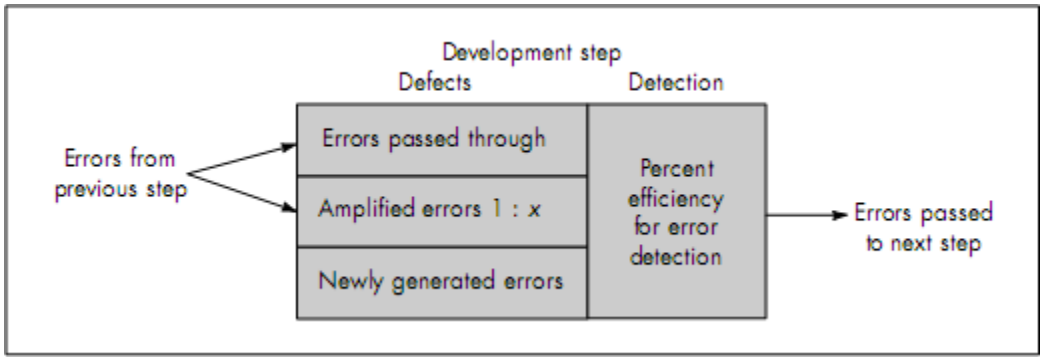
- causes uncontrolled acceleration that cannot be stopped
- does not respond to depression of brake pedal (by turning off)
- does not engage when switch is activated
- slowly loses or gains speed

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

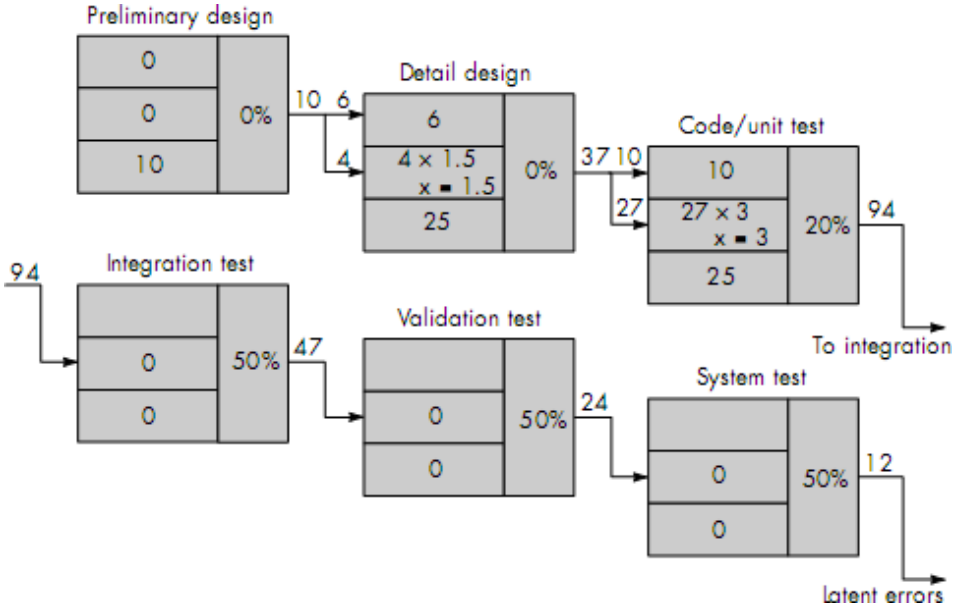
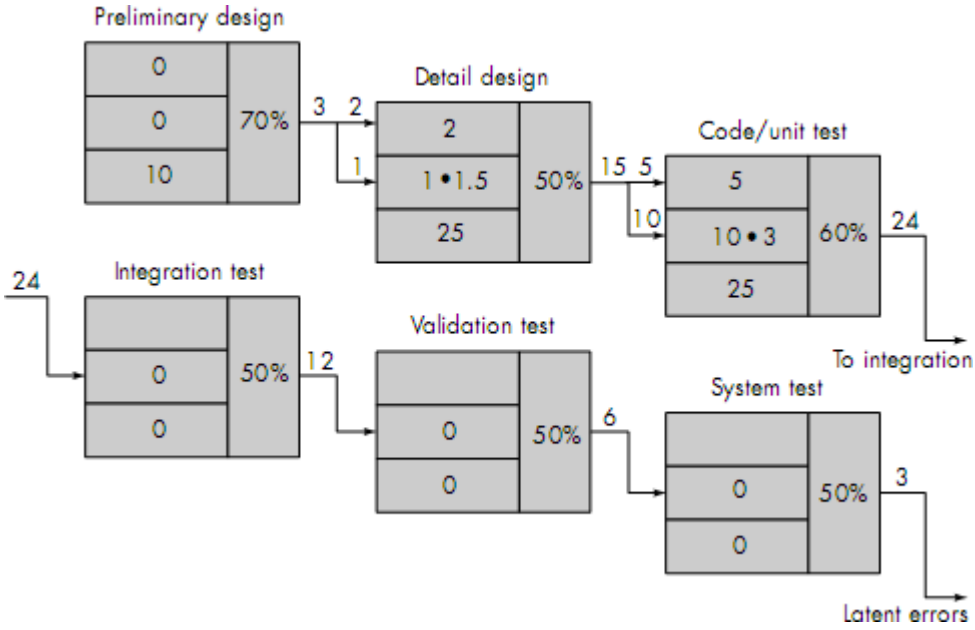


FIGURE 8.4
Defect amplification, reviews conducted



ISO 9000 Certification:

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards

The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production, then good quality products are bound to follow automatically. The types of industries to which the various ISO standards apply are as follows.

1. **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 Certification?

An organization determines to obtain ISO 9000 certification applies to ISO registrar office for registration. The process consists of the following stages:

1. **Application:** Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
3. **Document review and Adequacy of Audit:** During this stage, the registrar reviews the document submitted by the organization and suggest an improvement.
4. **Compliance Audit:** During this stage, the registrar checks whether the organization has compiled the suggestion made by it during the review or not.
5. **Registration:** The Registrar awards the ISO certification after the successful completion of allthe phases.
6. **Continued Inspection:** The registrar continued to monitor the organization time by time.