(Affilated to Purbanchal University)

Khwopa Engineering College

Libali-2, Bhaktapur

A Complete Manual Of

Software Engineering

BE Computer (Seventh Semester) New Syllabus

2014

Prepared By:

Er. Ganesh Ram Suwal

Lecturer

Khwopa Engineering College

Software Engineering

BEG 472CO

Year VI Semester: I

COURSE OBJECTIVES:

This course is intended to provide an introduction to SE concepts and practices focusing on industrial software development characteristics and processes, development models, and the software life cycle for mid-scale system.

Provide students a comprehensive introduction to software engineering. provide the students the kinds of activities that are necessary for developing a software system

Study the important phases of software development

UNIT I: (4 hrs)

Introduction to Software Engineering:

- Definition of Software engineering
- The evolving role of software
- Changing Nature of Software
- Characteristics of Software
- A Generic view of software Engineering
- Software engineering- layered technology.

UNIT II: (5 hrs)

Process models:

- The waterfall model
- Prototyping Model
- RAD Model
- Spiral Model.

UNIT III: (8 hrs)

Software Project Management

- Meaning of People, Product, Process, Project in Software Project Management
- Activities of Project Planning
- Project Estimation techniques
- COCOMO
- Risk Management
- Project Scheduling
- Staffing
- Software Configuration Management (SCM)

UNIT IV: (7 hrs)

Software Requirements and Specification

- Functional and non-functional requirements,
- Requirements engineering process
- (Feasibility studies, Requirements elicitation and analysis, Requirements validation, Requirements management)
- Data Modeling and flow diagram
- Software Prototyping Techniques
- Requirement definition and specification.

UNIT V: (7 hrs)

Software Design

- Introduction to Software Design
- Characteristics of a good Software Design
- Design Principal
- Design concepts
- Design Strategy
- Design process and Design quality
- Software Architecture and its types

UNIT VI: (7 hrs)

Software Testing

- Software testing Process
- Principal of Testing
- Test Case design
- Black-Box Testing(Boundary Value Analysis, Equivalence class Partitioning)
- White-Box testing(Statement Coverage, Path coverage, Cyclomatic complexity)
- Software Verification and Validation.

UNIT VII: (5 hrs)

Metrics for Process and Products

- Software Measurement
- Metrics for software quality
- Software Quality Assurance
- Software reliability
- The ISO 9000 quality standards.

UNIT VIII: (2hrs)

Introduction to Engineering Software Trends and Technology

- Agile Development
- Extreme Programming
- Cloud Computing and Grid Computing
- Enterprise Mobility
- Business Intelligent and Approaches
    - ERP, Supply Chain Management, Service Oriented Architecture and web services
    - Enterprise Portals and Content Management
- Introduction to OOSE

Case Studies

Students are encouraged to perform the case study to implement concepts of above- mentioned topics.

References:

1. Software Engineering, A practitioner’s Approach-Roger S. Pressman, 6th edition.McGrawHill International Edition.

2. Software Engineering- Sommerville, 7th edition, Pearson education.2004

3. Software Engineering (Latest Edition), Udit Agrawal

4. Fundamentals of Software Engineering (Latest Edition), Rajib Malla

5. Software Engineering – A precise Approach (Latest Edition), Pankaj Jalote

CHAPTER: 1 Introduction of Software

Computer software is the product that software engineers design and build. It encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text but also includes representations of pictorial, video, and audio information.

- Many people equate the term software with computer programs.
- Programs are developed by individuals for their personal use.
- They are generally smaller in size and have limited functionality.

Since the author of a program himself uses and maintains his programs, they usually don't have good user interface and lack proper documentation.

- **Software = programs + good user interface + operating procedures + documentations**
- Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operating.
- It consists of a no of separate programs, configuration files, system documentation which describes the structure of the system and user documentation which explains how to use the system.
- Software products have multiple users, therefore, should have good UI, proper operating procedures and good documentation support.

Software engineers are concerned with developing software products.

- Any program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared.
- Program=source code + object code
- Documentation= analysis (formal specification, CFD, DFD) , design (flow charts, ER diagrams), implementation (source code listing), testing (test data, test results)
- Operating procedures consists of instructions to setup and use the system

Operating procedures= user manuals (system overview, beginner's guide, reference guide), operational manuals (installation guide, system administration guide)

Software = Program + Associated Documents + Operating Procedure

Different types of **Documents** includes

1. Analysis / Specification
2. Design
3. Implementation
4. Testing

Analysis / Specification

- Formal Specification
- Context Diagram
- Data Flow Diagram

Design

- Flow Charts
- Entity-Relationship (E-R) Diagram

Implementation

- Source Code Listings
- Cross-Reference Listing

Testing

- Test Data
- Test Result

Types of software products

1. Generic products

- Stand alone systems which are produced by a development organization and sold on the open market to any customer who is able to buy them.
- Also called shrink-wrapped software
- Eg: word processors, drawing packages, project management tools
- The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.

2. Bespoke (customized) products

- Software that is commissioned by a specific customer to meet their own needs.
- Eg: air traffic control software, traffic monitoring systems.
- The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

Software engineering

- Engineering discipline which is concerned with all aspects of software production from early stage of system specification through a maintaining the system after it has gone into use.
- Engineering discipline, all aspects of software
- Engineers make things work. They apply theories, methods and tools where they are appropriate in a best manner
- Is not just concerned with technical aspects but also project management, development tools, theories and methods to support software production.

Small wall: use common sense bringing bricks, cements. Building a small wall and building a large building are entirely different. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering principles. The Small walls are very simple to build but multi storyed building is very difficult. That means the complexity is getting increased. Software Engineering is about the managing the complexity in Software.

Some Definition of Software Engineering

- **IEEE (1981):** Application of a systematic, disciplined and quantifiable approach to development, operation and maintenance of software.
- **Sommerville (1995):** Software engineering is concerned with the theories, methods and tools that are needed to develop the software products in a cost effective way.
- **Peer Torngren (1997):** Software engineering is what software engineers do.
- **Stephen Schach (1999):** Software engineering is a discipline whose aim is the production of fault-free software that satisfies the user's needs and that is delivered on time and within budget.

Why we need software engineering?

➔ We need software Engineering due to Change in nature & complexity of software

As per the IBM report, “31% of the project gets cancelled before they are completed, 53% overrun their cost estimates by an average of 189% and for every 100 projects, and there are 94 restarts”.

Advantages of software engineering

1. Improved requirement specification
2. Improved cost and scheduled estimates
3. Improved quality
4. Better use of automated tools and techniques
5. Less defects in final product
6. Better maintenance of delivered software
7. Well defined processes
8. Improved reliability
9. Improved productivity

Software engineering vs computer science

- Computer Science is concerned with the theories and methods which underlie computers and software systems whereas software engineering is concerned with the practical problems of producing software
- Some knowledge of CS is essential for software engineers in the same way some knowledge of physics is essential for electrical engineers
- All software engineering should be underpinned by theories of CS but in reality this is not the case.
- Software engineers must often use ad hoc approaches to develop the software.
- Elegant theories of CS can’t always be applied to real, complex problems which require a software solution

System Engineering

System Engineering concerned with all aspects of computer-based systems development including Hardware development, policy and process design and system deployment as well as software engineering. **System engineers** are involved in system **specification**, **architectural design**, **integration** and **deployment.** System

Engineering is about the services that the system provides the constraints under which the system must be built and operated and the interaction of the system with its environment.

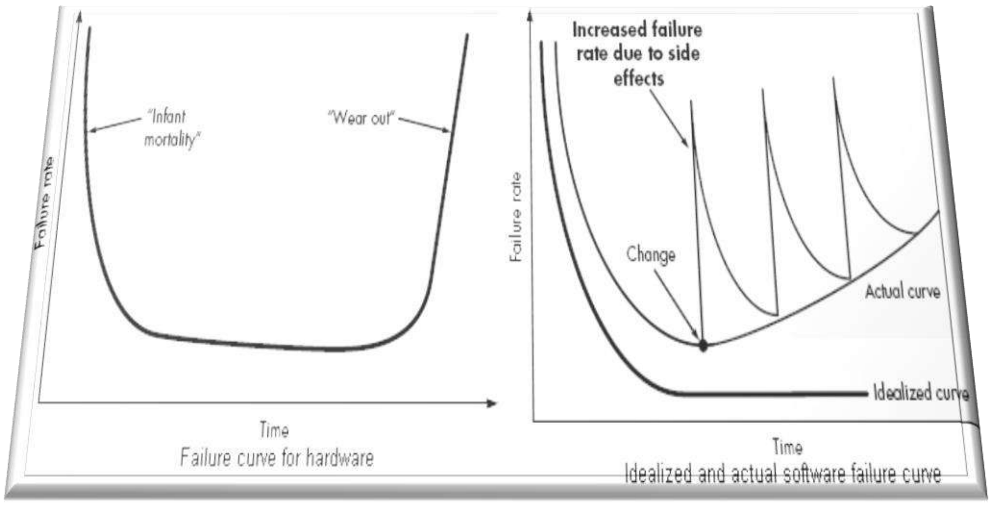
Software Characteristics

1. **Software does not wear out**

Well understood with the help of bath tub curve. There are three phases in the life of a h/w product **Initial phase** is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing problems, failure intensity will come down initially.

The second phase is the useful life phase where the failure intensity is approx. constant and called the useful life of a product.

Third Phase: After few years, again failure intensity will increase due to wearing out of components, called wear out phase. H/w components suffer from the cumulative effects of dust, vibration, abuse, temperature, environmental maladies.



Software does not wear out but it does deteriorate…

- Software is not susceptive to environmental maladies that cause hardware to wear out.
- In theory, failure rate curve for software should take the form of the "idealized curve".
- Undiscovered defects will cause high failure rate early in the life of a program.
- However, these are corrected (hopefully without introducing the other errors) so failure curve flattens.

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

2. Software is developed or engineered, it is not manufactured in the classical sense

The life of the software begins with the requirement and analysis phase and ends up with the retirement of a software product. Development of software is on time effort but it requires a continuous maintenance effort in order to keep it operational. However, making 100 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs us due to raw materials and other processing expenses.

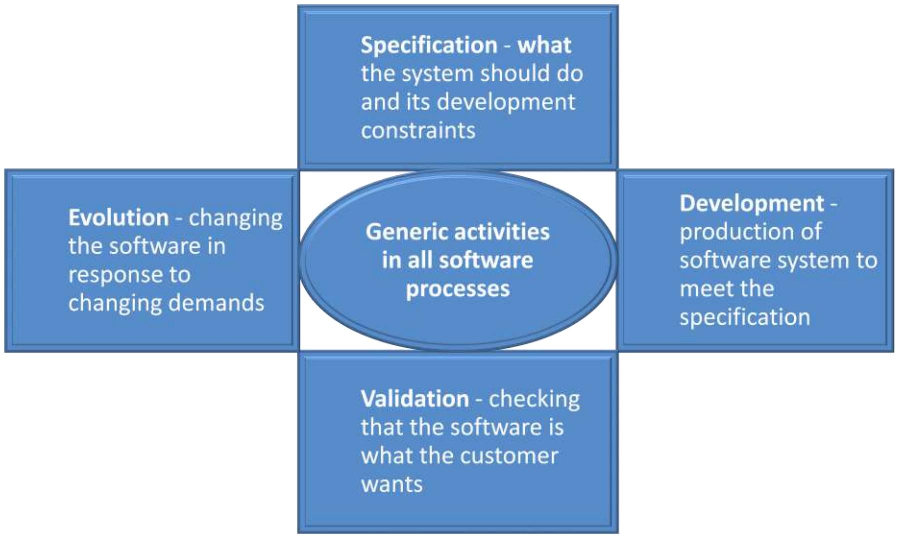
Q1. Software maintenance involves considerably more complexity than hardware maintenance

When a hardware component wears out, it is replaced by a spare part. There is no software spare part. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. You can’t easily detect the exact point where the errors have been occurred!!!!

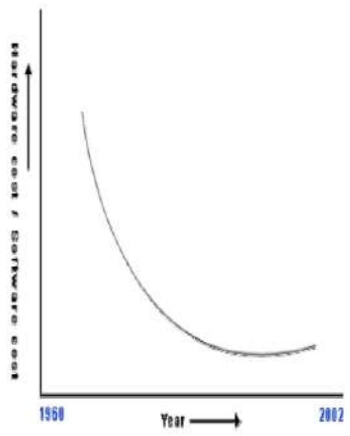
Software Process

Set of activities and associated results that produce the software product. Software process is the way in which we produce software. These activities are carried out by SEs. Four fundamental process activities are common to all software process

1. Specification
2. Development
3. Validation
4. Evolution


Specification - what the system should do and its development constraints
Evolution - changing the software in response to changing demands
Generic activities in all software processes
Development - production of software system to meet the specification
Validation - checking that the software is what the customer wants

Some of the Software failure and Software crisis



The term software crisis refers to a set of problems encountered in the development of computer software during 1960s. Within this period the software industry unsuccessfully attempted to build larger and larger software systems by simply scaling up existing development scheme.

- Poor quality software was produced
- Late Delivery of Software, Deadline time not meet
- Organizations spending larger portion of their budget on s/w
- s/w are turning out to be more expensive than h/w
- use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late
- Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis

1. Ariane 5

In june 1996, Anane 5 launcher broke up and exploded after 40 seconds of take off at an attitude of less than 4 km. the total loss was $500 million.

Reason

It was found that the error was due to overflow in the conversion from a 64 bit floating point number to 16 bit signed integer.

2. Y2K problem

It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year.

Reason

The 4-digit date format, like 1964, was shortened to 2-digit format, like 64.

3. The Patriot Missile

First time used in Gulf war. It is used as a defense from Iraqi Scud Missiles but failed several times including one that killed 28 US soldiers in Dhahran, Saudi Arabia

Reasons:

A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

Reasons for s/w crisis

Software crisis is characterized by an inability to develop s/w on time, within budget, and within requirements.

- Lack of communication between s/w developers and users
- Increase in size of software
- Increased complexity of problem area
- Project management problem
- Lack of understanding of problem and its environment
- High optimistic estimates regarding s/w development time and cost.

Attributes of good software?

The software should deliver the required functionality and performance to the user. The various no of attributes associated with a software decides whether it is good or bad. These attributes reflect the quality of a software product. The specific set of attributes which one expects from a software system depends on its application. A banking system must be secure, a telephone system must be reliable, an interactive game must be responsive. The general attributes of good software are:

1. Maintainability

Software must evolve to meet changing needs. This is a critical attribute coz software change is an inevitable consequence of a changing business environment.

2. Dependability

Software must be trustworthy. It has a range of sub attributes as reliability, security and safety. Dependable software should not cause physical or economical damage in the event of system failure.

3. Efficiency

Software should not make wasteful use of system resources such as memory, processor and storage. So it includes responsiveness, processing time, and memory utilization.

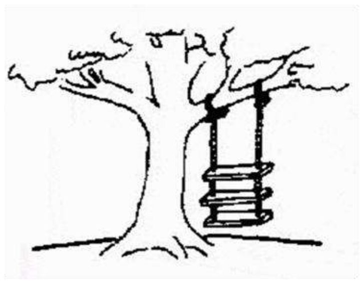
4. Usability

Software becomes usable if it does not call for extra effort to be learnt. Usability increases with good documentation and an appropriate user interface.

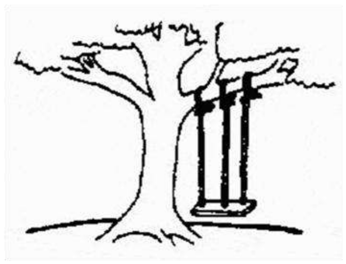
Software Application Types

1. System software (compilers, drivers)
2. Real time software
3. Business software (payroll, MIS)
4. Engineering and scientific software (simulation, CAD)
5. Embedded software
6. Personal computer software (word processing, multimedia, graphics, DBMS, entertainment)
7. Web based software
8. Artificial intelligence software (game playing, expert systems, pattern recognition)

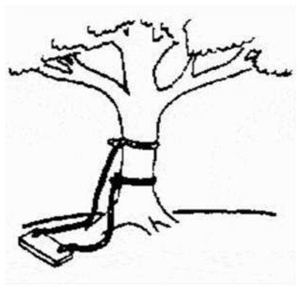
How Programs Are Usually Written



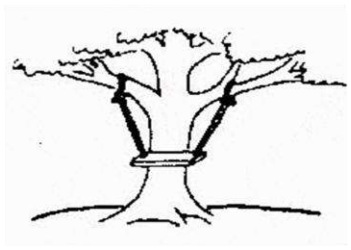
Requirements specification was defined like this



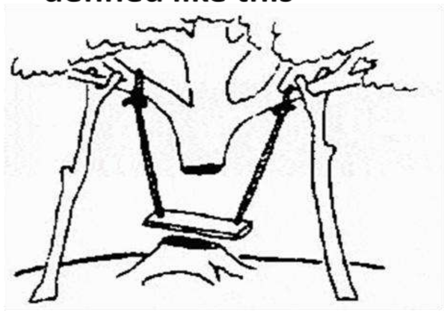
The developers understood it in that way



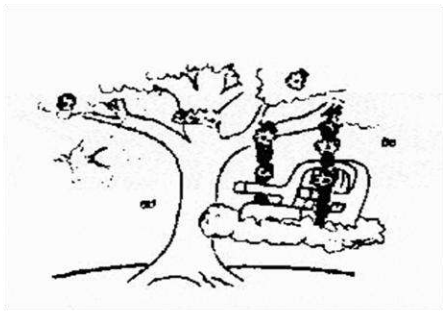
This is how problem was solved before.



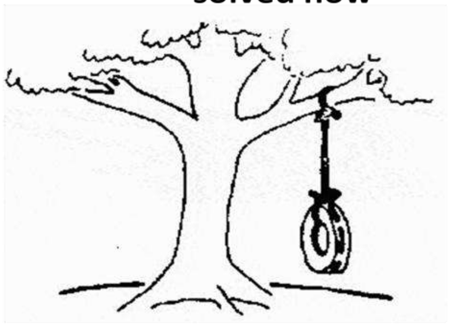
This is how problem is solved now



That is program after debugging



This is how program is described by marketing department



This, in fact, is what the customer wanted … ;-)

CHAPTER: 2 Process Model

The goal of Software Engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable.

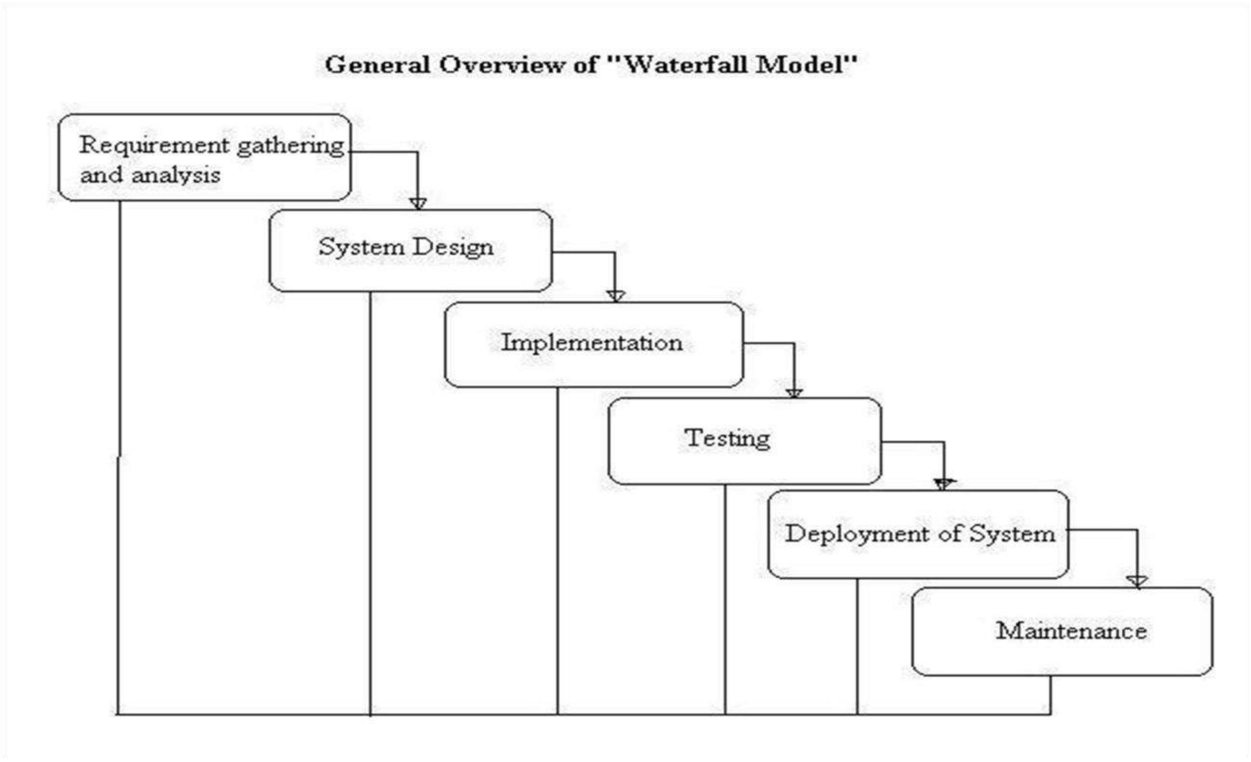
"The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase".

Types of software process models

1. **Linear Sequential Model (Water Fall Model)**
2. **Incremental Process Model**
   - **2.1. Incremental Model**
   - **2.2. The RAD Model**
3. **Evolutionary Model**
   - **3.1 Prototyping Model**
   - **3.2 Spiral Model**
   - **3.3 Win-Win Spiral Model**
   - **3.4 Component Based Development**
4. **Concurrent Development Model**
5. **Formal Methods Models**
6. **Fourth Generation Technology Models**

1. Linear Sequential Model (Water Fall Model)

This model is named "waterfall model" because its diagrammatic representation resembles a cascade of waterfalls. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In waterfall model phases do not overlap.


General Overview of "Waterfall Model"
Requirement gathering and analysis
System Design
Implementation
Testing
Deployment of System
Maintenance

Problems of waterfall model

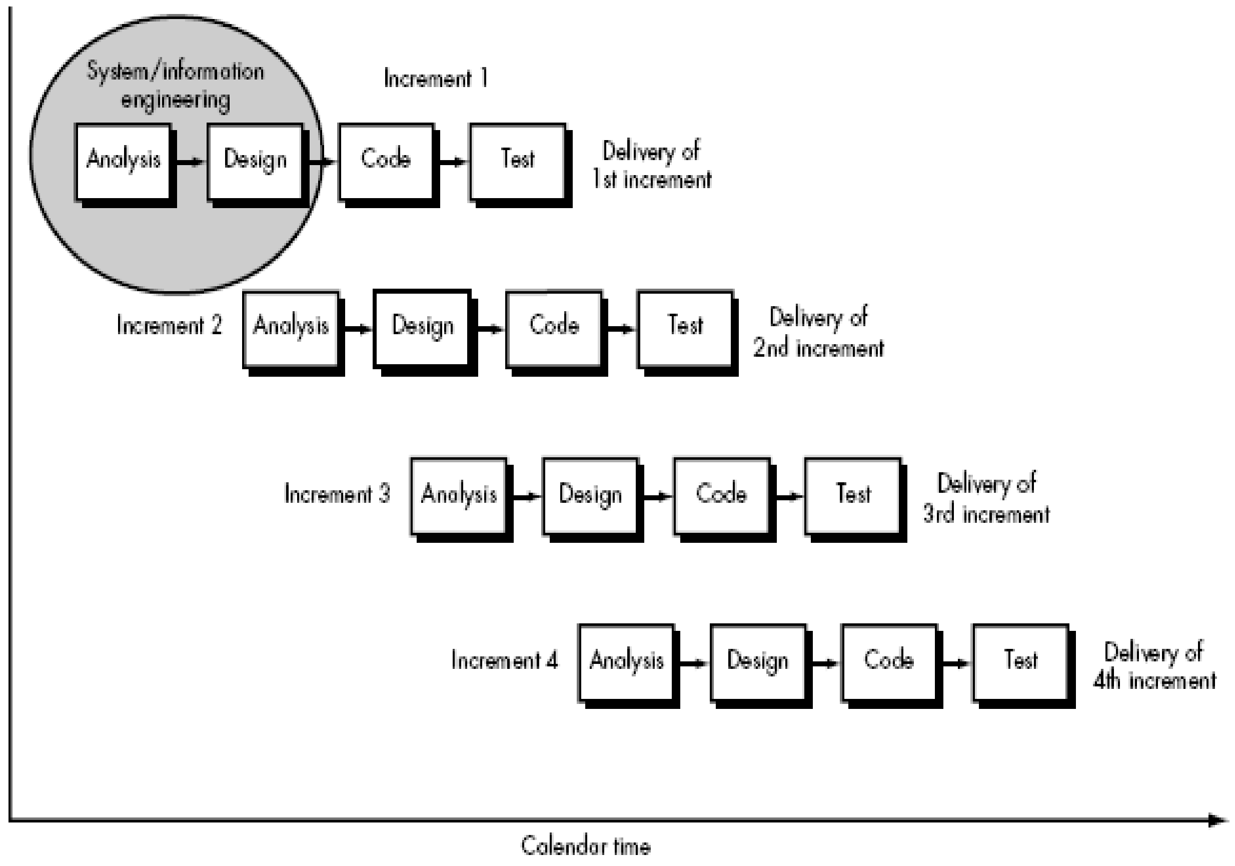
1. It is difficult to define all requirements at the beginning of a project
2. This model is not suitable for accommodating any change
3. A working version of the system is not seen until late in the project's life
4. It does not scale up well to large projects.
5. Real projects are rarely sequential.

2. Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but they may be conducted in several cycles. Useable product is released at the end of the each cycle, with each release providing additional functionality.

- Customers and developers specify as many requirements as possible and prepare a SRS document.
- Developers and customers then prioritize these requirements
- Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities.

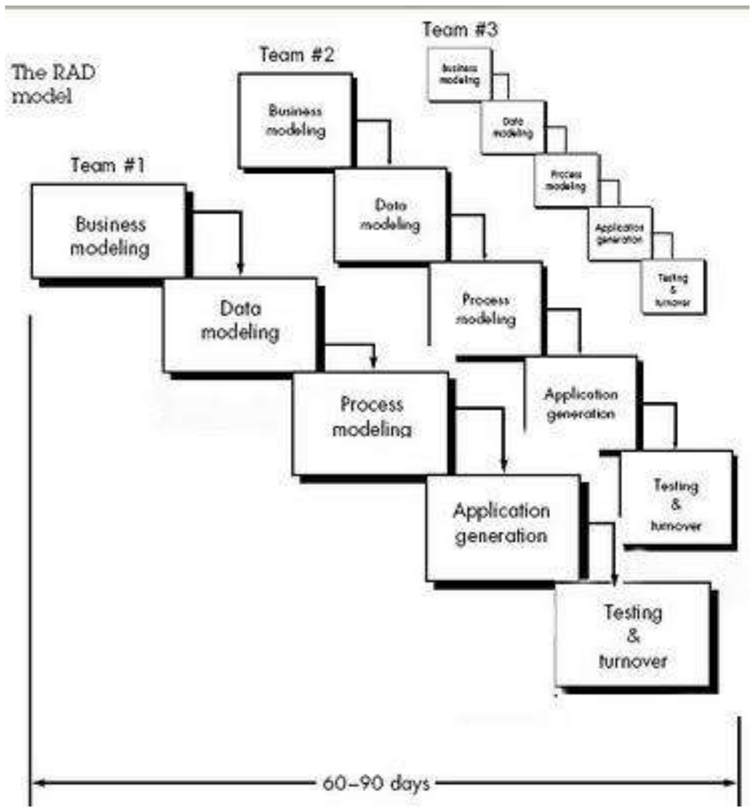
Iteration Model


System/information engineering
Increment 1
Analysis
Design
Code
Test
Delivery of 1st increment
Increment 2
Delivery of 2nd increment
Increment 3
Delivery of 3rd increment
Increment 4
Delivery of 4th increment
Calendar time

2.2 RAD Model

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

The rapid application development model emphasizes on delivering projects in small pieces. If the project is large, it is divided into a series of smaller projects. Each of these smaller projects is planned and delivered individually. Thus, with a series of smaller projects, the final project is delivered quickly and in a less structured manner. The major characteristic of the RAD model is that it focuses on the reuse of code, processes, templates, and tools.



The phases of RAD model are listed below.

1. **Planning:** Inthis phase, the tasks and activities are planned. The derivables produced from this phase are project definition, project management procedures, and a work plan. **Project definition** determines and describes the project to be developed. **Project management procedure** describes processes for managing issues, scope, risk, communication, quality, and so on. **Work plan** describes the activities required for completing the project.
2. **Analysis:** The requirements are gathered at a high level instead of at the precise set of detailed requirements level. In-case the user changes the requirements, RAD allows changing these requirements over a period of time. This phase determines plans for testing, training and implementation processes. Generally, the RAD projects are small in size, due to which high-level strategy documents are avoided.
3. **Prototyping:** The requirements defined in the analysis phase are used to develop a prototype of the application. A final system is then developed with the help of the prototype. For this, it is essential to make decisions regarding technology and the tools required to develop the final system.

4. **Repeat analysis and prototyping as necessary:** When the prototype is developed, it is sent to the user for evaluating its functioning. After the modified requirements are available, the prototype is updated according to the new set of requirements and is again sent to the user for analysis.
5. **Conclusion of prototyping:** As a prototype is an iterative process, the project manager and user agree on a fixed number of processes. Ideally, three iterations are considered. After the third iteration, additional tasks for developing the software are performed and then tested. Last of all, the tested software is implemented.
6. **Implementation:** The developed software, which is fully functioning, is deployed at the user's end.

Table Advantages and Disadvantages of RAD Model

Advantages	**Disadvantages**
1. Deliverables are easier to transfer as high-level abstractions, scripts, and intermediate codes are used. **2.** Provides greater flexibility as redesign is done according to the developer. **3.** Results in reduction of manual coding due to code generators and code reuse. **4.** Encourages user involvement. **5.** Possibility of lesser defects due to prototyping in nature.	**1.** Useful for only larger projects **2.** RAD projects fail if there is no commitment by the developers or the users to get software completed on time. **3.** Not appropriate when technical risks are high. This occurs when the new application utilizes new technology or when new software requires a high degree of interoperability with existing system. **4.** As the interests of users and developers can diverge from single iteration to next, requirements may not converge in RAD model.

3. Evolutionary Model

Evolutionary models are inherently iterative in nature. It helps to develop increasingly more complete versions of the target software

Why Evolutionary Model

- Business **and** product **requirement changes as the development proceeds Straight path to end product becomes** unrealistic
- Tight **market deadline** Limited version **to be introduced**
- System requirements **are well understood but details of** product extensions **or** system extensions **are not known**

Evolutionary Models are

1. Prototyping Model
2. Spiral Model
3. Win-Win Spiral Model
4. Concurrent development Model

3.1 Prototyping Model

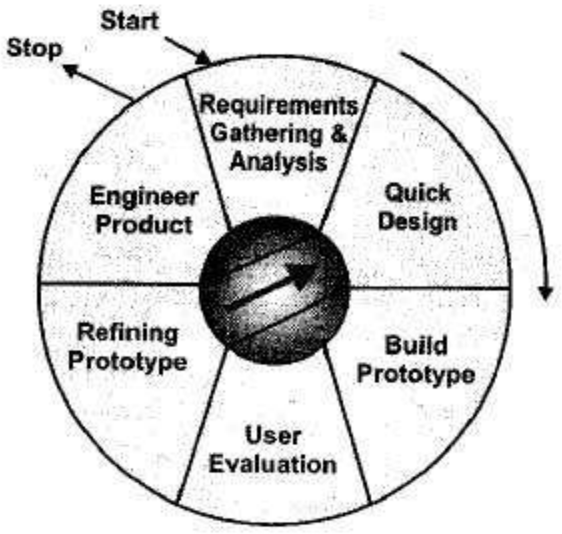
The **prototyping model** is applied when detailed information related to input and output requirements of the system is not available. In this model, it is assumed that all the requirements may not be known at the start of the development of the system. It is usually used when a system does not exist or in case of a large and complex system where there is no manual process to determine the requirements. This model allows the users to interact and experiment with a working model of the system known as **prototype.** The prototype gives the user an actual feel of the system.

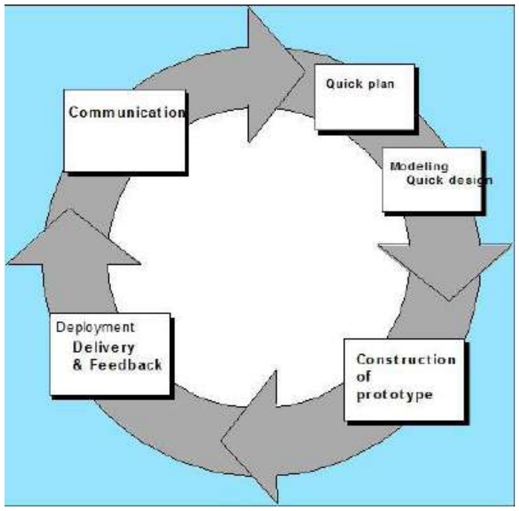
At any stage, if the user is not satisfied with the prototype, it can be discarded and an entirely new system can be developed. Generally, prototype can be prepared by the approaches listed below.

1. By creating main user interfaces without any substantial coding so that users can get a feel of how the actual system will appear

2. By abbreviating a version of the system that will perform limited subsets of functions

3. By using system components to illustrate the functions that will be included in the system to be developed





1. Requirements gathering and analysis: A prototyping model begins with requirements analysis and the requirements of the system are defined in detail. The user is interviewed in order to know the requirements of the system.

2. Quick design: When requirements are known, a preliminary design or quick design for the system is created. It is not a detailed design and includes only the important aspects of the system, which gives an idea of the system to the user. A quick design helps in developing the prototype.

3. Build prototype: Information gathered from quick design is modified to form the first prototype, which represents the working model of the required system.

4. User evaluation: Next, the proposed system is presented to the user for thorough evaluation of the prototype to recognize its strengths and weaknesses such as what is to be added or removed. Comments and suggestions are collected from the users and provided to the developer.

5. Refining prototype: Once the user evaluates the prototype and if he is not satisfied, the current prototype is refined according to the requirements. That is, a new prototype is developed with the additional information provided by the user. The new prototype is evaluated just like the previous prototype. This process continues until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed on the basis of the final prototype.

6. Engineer product: Once the requirements are completely met, the user accepts the final prototype. The final system is evaluated thoroughly followed by the routine maintenance on regular basis for preventing large-scale failures and minimizing downtime.

Advantages and Disadvantages

Advantages	**Disadvantages**
1. Provides a working model to the user early in the process, enabling early assessment and increasing user's confidence. 2. The developer gains experience and insight by developing a prototype there by resulting in better implementation of requirements. 3. The prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between the developers and users. 4. There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent. 5. Helps in reducing risks associated with the software.	1. If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive. 2. The developer loses focus of the real purpose of prototype and hence, may compromise with the quality of the software. For example, developers may use some inefficient algorithms or inappropriate programming languages while developing the prototype. 3. Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is finished when it is not. 4. The primary goal of prototyping is speedy development, thus, the system design can suffer as it is developed in series without considering integration of all other components.

3.2 Spiral Model

Spiral Model, Originally Proposed by Dr. Berry Boehm is an evolutionary Software Process model in the year 1980s. It combines the feature of prototyping model and waterfall model. This evolutionary software process model combines the iterative nature of prototyping model, the control and systematic aspect of linear sequential model

It has potential for rapid development of incremental versions of the software

Software is developed in a series of incremental releases

- **Early stage increments:** paper model or prototype
- **Subsequent stage releases:** more complete version of required software

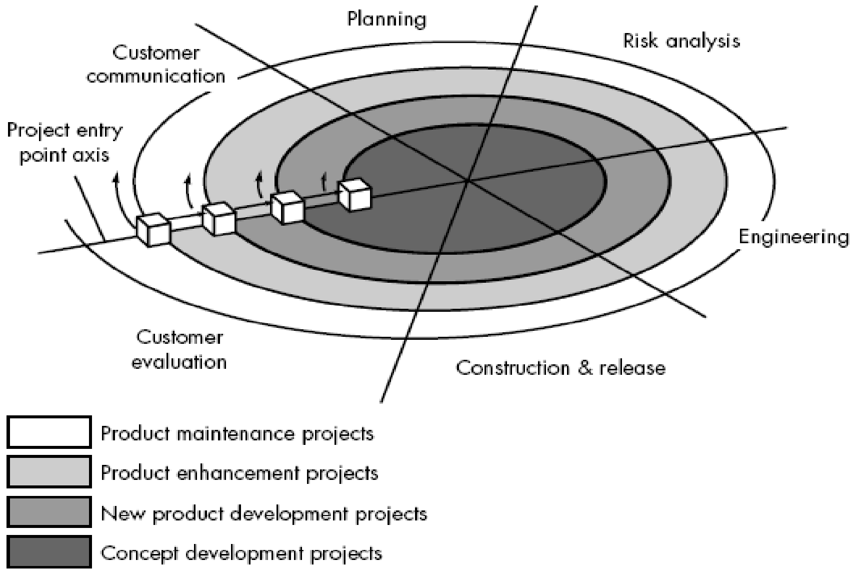
Spiral Model is divided into a number of Frame Work Activities or Task Regions. The angular dimension represents the progress made in completing each cycle. Each loop of the spiral from X-axis clockwise through 360^o represents one phase.

There are six task regions or frame work activities:

1. **Customer Communication:** Task for effective communication between customer and developer
2. **Planning:** Task is to define resources, timeline and other project related information
3. **Risk analysis:** the task is to assess technical and management risks
4. **Engineering:** task required to build one or more representations of the application
5. **Construction and release:** task to construct, test, install and provide support (documentation, training etc)
6. **Customer evaluation:** task is to obtain customer feedback or evaluation (engineering stage versus implementation stage)

Spiral model handles the software development process in phase manner, each phase being treated as a project work

Spiral model divides the development process into **four projects**:



Each region is populated by a series of tasks specific to the nature of the project. In all cases, umbrella activities are applied (SCM and SQA)

All the stages **iterative in nature:**

First Iteration: Results in production of product specification

Second Iteration: Results in production of product prototype

Next Iteration: Results in production of progressively more sophisticated versions of the software

- Each pass thru the planning region two results in adjustment to the project plan
- Cost and schedule adjusted on the basis of customer evaluation
- Project manager adjusts the number of iterations to complete the software
- Classical model ends when the software is delivered. Spiral model can be applied thru out the life of the software
- Each project in the Spiral model has a starting point in the project entry point axis, which represents the start of a different type of project

Spiral model remains active until the **software retires**

- Spiral model is a realistic approach to development of large scale projects
- Spiral model uses Prototyping as a Risk Reduction mechanism. Prototyping is applied at any stage of the product
- It incorporated systematic approach as suggested by classical life cycle of software in an iterative way (frame-work)
- It demands direct consideration of technical risk
- Discussions:
- Difficult to convince customer that evolutionary approach is controllable
- High expertise is required to assess considerable risk
- This is a new model not used widely as linear sequential development approach

It will take number of years before the effectiveness of this model is known

Table Advantages and Disadvantages of Spiral Model

Advantages	**Disadvantages**

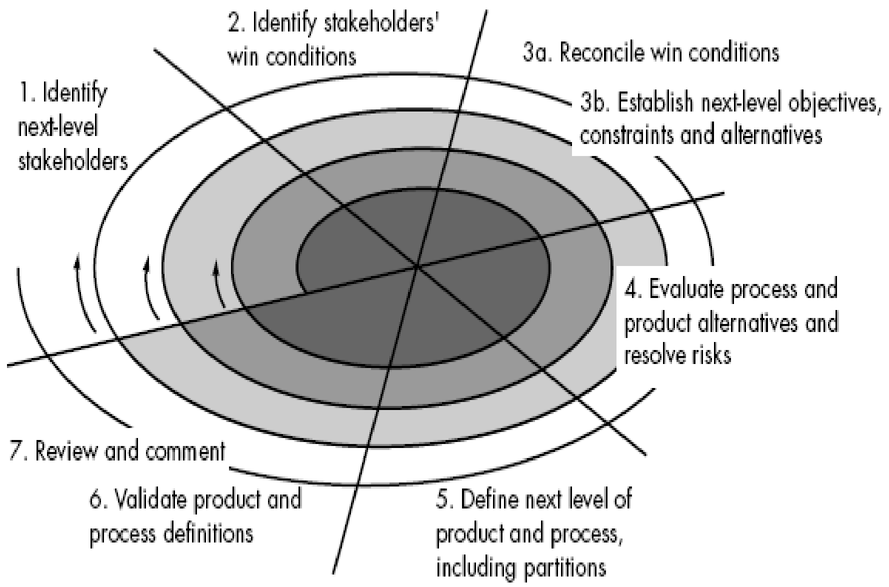
1. Avoids the problems resulting in risk-driven approach in the software 2. Specifies a mechanism for software quality assurance activities 3. Is utilized by complex and dynamic projects 4. Re-evaluation after each step allows changes in user perspectives, technology advances, or financial perspectives. 5. Estimation of budget and schedule gets realistic as the work progresses.	1. Assessment of project risks and its resolution is not an easy task. 2. Difficult to estimate budget and schedule in the beginning as some of the analysis is not done until the design of the software is developed.

3.3 Win-Win Spiral Model

The Spiral model suggests customer communication to decide upon project requirements form customer and the Developer asks what is required and customer provides necessary details. In reality, developer negotiates with customer for functionality, performance, and other product / system features against cost and time to market. Negotiation is successful at **Win-Win state:**

Customer wins by getting a product /system that satisfies majority of his requirements

Developer wins by deadline target and achievable budget


2. Identify stakeholders' win conditions
3a. Reconcile win conditions
1. Identify next-level stakeholders
3b. Establish next-level objectives, constraints and alternatives
4. Evaluate process and product alternatives and resolve risks
7. Review and comment
6. Validate product and process definitions
5. Define next level of product and process, including partitions

Negotiation takes place at the beginning of each pass around the spiral, involving the **following activities**:

- Identification of the key stake holders of the system/sub systems
  - Determination of the stake holder's win condition

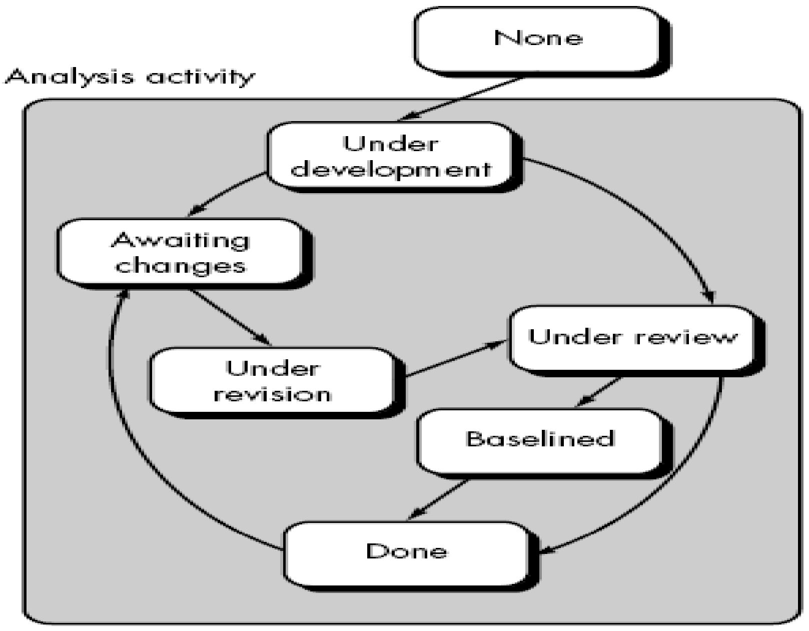
- Negotiation of the stake holder's win condition to fit into a set of Win-Win condition
- for all concerned (including software development project team)

4 Concurrent Development Model (by Davis and Sitaram)

Project managers tracking status of major phases of a project have no idea of project status since personnel are associated with more than one activity - might be writing SRS, doing design, coding, testing etc. all simultaneously

This shows existence of concurrency of activities occurring in any one phase (requirements change during late development) which can be represented by notations to represent the state of a process (state chart)

Existence of concurrency of activities affects the time bound nature of software development process



Representation:

- Any state of a concurrent process model can be represented schematically as a series of major technical activities, tasks and associated states e.g. analysis, activity can be represented as shown
- All activities resides concurrently but resides in different states

For a **Spiral Model:**

- When customer communication activity completed the first iteration and is in state three the analysis activity makes a transition from state one to state two.

- Now as a part of customer communication activity , the customer signals a change in requirement , analysis activity makes a move to state three
- Concurrent process model defines a series of events that will trigger transition from state to state for each software engineering activity

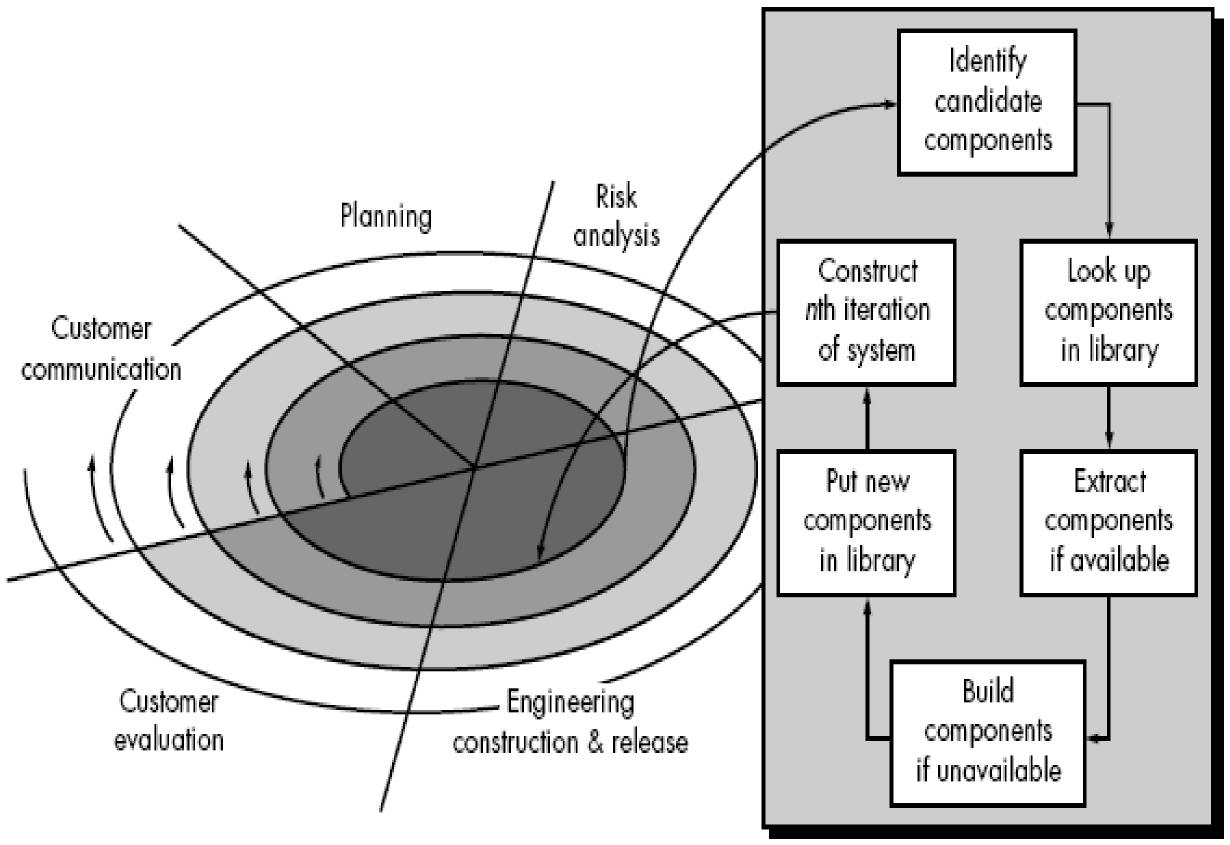
In general this model is used as a paradigm for client server applications which comprises of a set of functional components

- Concurrent development model defines client/server applications in two dimensions:
- System Dimensions: involves three activities (design, assembly, use)
- Component dimensions: involves two activities (design and realization)
- Concurrency is achieved in two ways:
  1. System and component activities can be concurrently taking place (a state oriented approach)
  2. Design and realization of many components can take place concurrently
- Comments:
- Applicable to all types of software developments
- Helps to figure out the actual picture of the state of the project
- Instead of showing software engineering activities as a sequence of tasks it defines a network of activities existing simultaneous with other activities
- Events generated in one activity may trigger a state transition of an activity

5 Component Assembly Model

- Component Assembly model is based on the principle of Object Oriented Technology
- OOT emphasizes creation of class that encapsulates both data and the algorithm (code) that are used to manipulate the data
- OOT provides very high level of reusability
- Component assembly model resembles spiral model
- It is evolutionary in nature and uses iterative approach to develop software

It composes applications from prepackaged software components (class)


Planning
Risk analysis
Customer communication
Customer evaluation
Engineering construction & release
Identify candidate components
Construct nth iteration of system
Look up components in library
Put new components in library
Extract components if available
Build components if unavailable

- Accomplished by examining the data that are to be manipulated by the application and the algorithms that will be applied to accomplish manipulation these data and code are packaged to form a software component or a class
- Old classes are stored in class libraries
- This model leads to software reuse
- Reusability provides measurable benefits to software engineering processes
- QSM Inc. report indicates:
- 70% reduction in development cycle time
- 84% reduction in project cost

26.2% productivity index versus 16.9 as industry norm

Q1. What is SRS? List & Describe Various Characteristics of an SRS

Software requirement specification (SRS) is a document that completely describes what the proposed software should do without describing how software will do it. The basic goal of the requirement phase is to produce the SRS, Which describes the complete behavior of the proposed software. SRS is also helping the clients to understand their own needs.

Advantages

Software SRS establishes the basic for agreement between the client and the supplier on what the software product will do.

1. A SRS provides a reference for validation of the final product.
2. A high-quality SRS is a prerequisite to high-quality software.
3. A high-quality SRS reduces the development cost.

Characteristics of an SRS

1. Correct
2. Complete
3. Unambiguous
4. Verifiable
5. Consistent
6. Ranked for importance and/or stability
7. Modifiable
8. Traceable

CHAPTER: 3 Software Project Management

Software Project Managements is an umbrella activity within Software Engineering. Project Management involves the Planning, Monitoring and Control of People, Process and Events that occur as Software evolves from preliminary concept to an operational implementation. Project Management begins before any technical activity and continues throughout the Definition, Development and Support of Computer Software.

Project

A unique process, consisting of a set of coordinated and controlled activities with start and finish dates, undertaken to achieve an objective conforming to specific requirements including constraints of time, cost and resources

-(Lockyer and Gordon, 1996)

A Project is

- Unique process
- Coordinated and controlled activities
- Start and finish dates
- To achieve an objective
- Specific requirements
- Constraints of time, cost and resources

<u>WHO DOES IT</u> ? Project Managers

Project Managers Plan, Monitor and Control the work of Team of Software Engineers.

<u>WHY IS IT IMPORTANT</u> ?

Building a Computer Software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why Software Projects need to be managed.

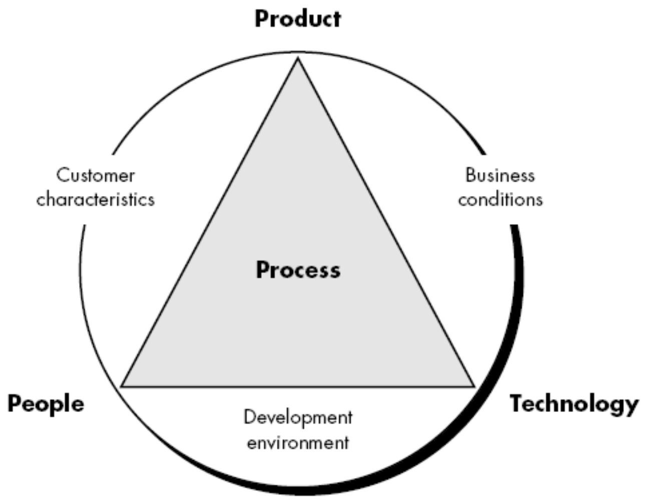
<u>WHAT IS THE WORK PRODUCT</u> ? ''A PROJECT PLAN''

Project Plan defines the Process and Tasks to be conducted, the People who will do the work and the mechanism to assessing Risks, Controlling Change and Evaluating Quality.

Project Plan is produced as Management activities commences

Effective Project Management focuses on 4 P's

People, Product, Process, Project



THE PEOPLE

- The People Factor is so important that Software Engineering institution has developed a People Management Capability Maturity Model. (PM-CMM).
- PM-CMM Defines the following key practice areas for Software People:-

  -Recruiting

  - Selection

  - Performance Management

  - Training

  - Compensation

  - Career Development

  - Organization and work Design

  - Team Culture Development

Organizations that achieve high level of Maturity in People Management area have a higher likelihood of implementing effective Software Engineering Practices.

THE PRODUCT (Software Application)

Before a Product can be Planned:-

- Product Objectives and Scope should be planned

- Alternative solutions should be considered
- Technical and Management Constraints should be identified.

Without the PRODUCT Plan it is not possible to define:-

- Reasonable and accurate Estimates of Cost and Effective Risk Management)
- Realistic breakdown of Project tasks (WBS)
- Project Schedule that provides a meaningful indication of progress.

THE PROCESS

- Software Process provides the Framework from which a comprehensive Plan for Software Development can be established.
- A small number of Framework activities are applicable to all Software Projects regardless of Project size or complexity.

A number of Different Task set however, such as:-

- Tasks
- Milestones
- Work Product (Deliverables)
- Quality Assurance points enable the Framework activities to be adapted to the characteristics of the Software Project and the requirements of the Project team.

THE PROJECT

- Software Project Development is conducted in a Planned and Controlled way since it is only known way to manage complexity. And yet we still struggle.
- In 1998 industry data indicated that 26% of Project failed outright and 46% experienced Cost and Schedule overruns.
  Although Project success rate for Projects has improved, yet Project failure rate remains higher than it should be!

WHY PROJECTS FAIL

- An unrealistic Deadlines is established

- Changing Customer Requirements
- An honest underestimate of effort
- Predictable and/ or unpredictable risks
- Miscommunication among project staff
- Failure in Project Management practice

THE PEOPLE AS PROJECT PLAYERS (STAKEHOLDERS)

The Software Process is populated by People as Players or otherwise called as Stakeholders. Who can be categories into one of the Five constituencies:-

- Senior Mangers
- Project Managers
- Software Engineers (Practitioners)
- Customers or Clients
- End-users

Since the Software Process is populated by People. The Project Team must be organized in a way to maximize each person's skills and ability. The Organization of the Project Team is the Job of Project Team Leader may be called Project Manager.
Companies that organizes and Manages their people wisely. Prosper in the long run!!!

COCOMO Model

The Constructive cost model (COCOMO) was developed by Boehm. This model also estimates the total effort in terms of person-months of the technical project staff. The effort estimate includes development, management, and support tasks but does not include the cost of the secretarial and other staff that might be needed in an organization. The basic steps in this model are: -
The **COstructive COst MOdel** (COCOMO) is the most widely used software estimation model in the world. It
The COCOMO model predicts the **effort** and **duration** of a project based on inputs relating to the size of the resulting systems and a number of "**cost drives**" that affect productivity.

Software Cost Estimation

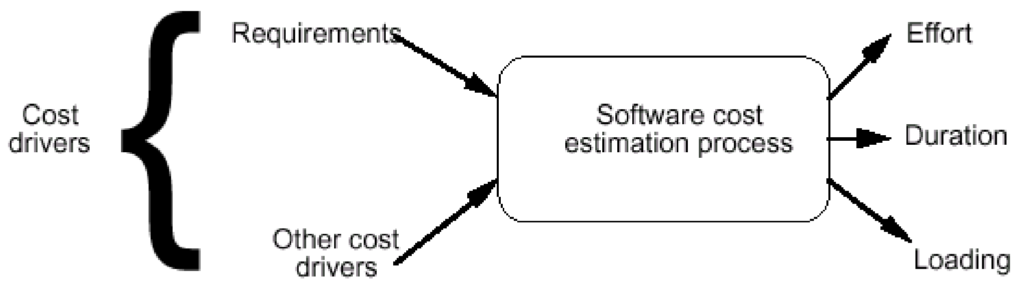


Figure 1. Classical view of software estimation process.

Effort:

- Effort Equation
  - **PM = C * $(\text{KDSI})^n$ (**person-months)
    - where **PM** = number of person-month (=152 working hours),
    - **C** = a constant,
    - **KDSI** = thousands of "delivered source instructions" (DSI) and
    - **n** = a constant.

Productivity:

- Productivity equation
  - **(DSI) / (PM)**
    - where **PM** = number of person-month (=152 working hours),
    - **DSI** = "delivered source instructions"

Schedule:

- Schedule equation
  - **TDEV = C * $(\text{PM})^n$ (**months)
  - Where TDEV = number of months estimated for software development.

Average Staffing:

- Average Staffing Equation
  - **(PM) / (TDEV)** **(**FSP)
  - Where FSP means Full-time-equivalent Software Personnel.

COCOMO Models

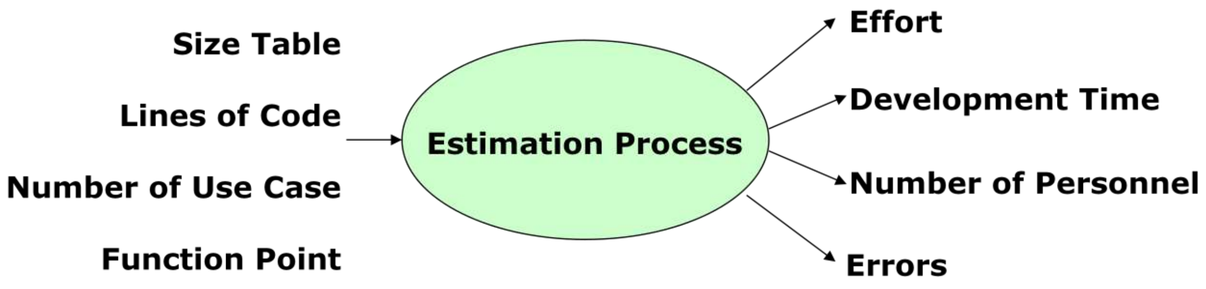
COCOMO is defined in terms of three different models:

1. **Basic model**
2. **Intermediate model**
3. **Detailed model**.

- The more complex models account for more factors that influence software projects, and make more accurate estimates.

Cost Estimation Process

Cost = SizeOfTheProject x Productivity



Risk Management

Any large project involves certain risks, and that is true for software projects. Risk management is an emerging area that aims to address the problem of identifying and managing the risks associated with a software project.

Risk is a project of the possibility that the defined goals are not met. The basic motivation of having risk management is to avoid disasters and heavy losses. The current interest in risk management is due to the fact that the history of software development projects is full of major and minor failures. A large percentage of projects have run considerably over budget and behind schedule, and many of these have been abandoned midway. It is now argued that many of these failures were due to the fact that the risks were not identified and managed properly.

Risk management is an important area, particularly for large projects. Like any management activity, proper planning of that activity is central to success. Here we discuss various aspects of risk management and planning.

Risk Management Overview

1. Risk is defined as an exposure to the chance of injury of loss. That is, risk implies that there is a possibility that negative may happen. In the context of software projects, negative implies that here is an adverse effect on cost, quality, or schedule. Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimum.
2. Like configuration management, which minimizes the impact of change, risk management minimizes the impact of risks. However, risk management is generally done by the project management. For this reason we have not considered risk management as a separate process (through it can validly be considered one) but have considered such activities as part of project management.
3. Risk management can be considered as dealing with the possibility and actual occurrence of those events that are not "regular" or commonly expected. Normally project management handles the commonly expected events, such as people going on leave or some requirements changing. It deals with events that are infrequent, somewhat out of the control of the project management, and are large enough (i.e. can have a major impact on the project) to justify special attention.

The Principles of Risk Management

1. Global Perspective: In this we look at the larger system definitions, design and implementation. We look at the opportunity and the impact the risk is going to have.

2. Forward Looking View: Looking at the possible uncertainties that might creep up. We also think for the possible solutions for those risks that might occur in the future.

3. Open Communication: This is to enable the free flow of communication between in the customers and the team members so that they have clarity about the risks.

4. Integrated management: In this phase risk management is made an integral part of project management.

5. Continuous process: In this phase the risks are tracked continuously throughout the risk management paradigm.

Risk management paradigm

1. Identify: Search for the risks before they create a major problem

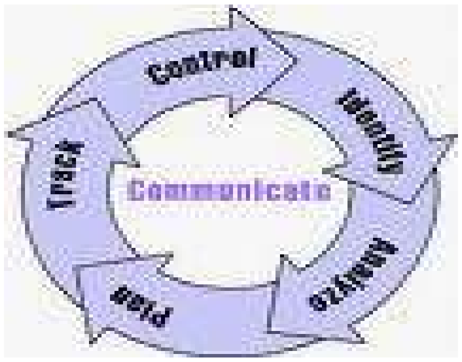
2. Analyze: understand the nature, kind of risk and gather information about the risk.

3. Plan: convert them into actions and implement them.

4. Track: we need to monitor the necessary actions.

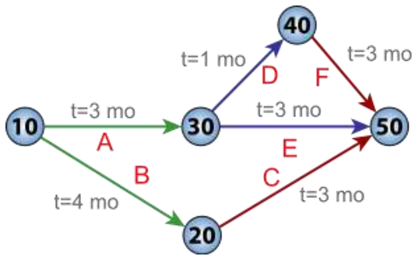
5. Control: Correct the deviation and make any necessary amendments.

6. Communicate: Discuss about the emerging risks and the current risks and the plans to be undertaken.

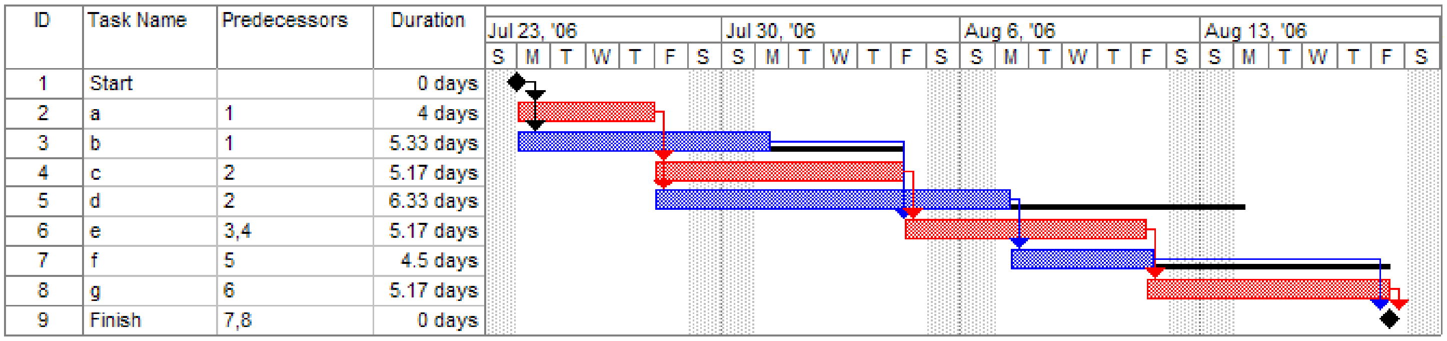


PERT Chart: Activity Network

Program Evaluation and Review Technique is designed to analyze and represent the tasks involved in completing a given project.



Gantt chart: Activity Timeline



ID	Task Name	Predecessors	Duration
1	Start		0 days
2	a	1	4 days
3	b	1	5.33 days
4	c	2	5.17 days
5	d	2	6.33 days
6	e	3,4	5.17 days
7	f	5	4.5 days
8	g	6	5.17 days
9	Finish	7,8	0 days

Software configuration management (SCM)

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of software engineering becomes part of a software configuration. The configuration is organized in a manner that enables orderly control of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items that are produced as a result of some software engineering activity. In addition to documents, programs, and data, the development environment that is used to create software can also be placed under configuration control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and composite objects form an object pool from which variants and versions are created. Version control is the set of procedures and tools for managing the use of these objects. Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

Software configuration management (SCM) is the discipline for systematically controlling the changes that take place during development. Software configuration management is a process independent of the development process largely because most development models cannot accommodate change at any time during development. SCM can be considered as having three major components:

1. Software configuration identification
2. Change control
3. Status accounting and auditing

1. Configuration identification:

The first requirement for any change management is to have clearly agreed-on basis for change. That is, when a change is done, it should be clear to what changes has been applied. This requires baselines to be established. A baseline change is the changing of the established baseline, which is controlled by SCM.

After baseline changes the state of the software is defined by the most recent baseline and the changes that were made. Some of the common baselines are functional or requirements baseline, design baseline, and product or system baseline. Functional or requirement baseline is generally the requirements document that specifies the functional requirements for the software. Design baseline consists of the different components in the software and their designs. Product or system baseline represents the developed system.

It should be clear that a baseline is established only after the product is relatively stable. Though the goal of SCM is to control the establishment and changes to these baselines, treating each baseline as a single unit for the purpose of change is undesirable, as the change may be limited to a very small portion of the baseline.

2. Change control:

Most of the decisions regarding the change are generally taken by the configuration control board (CCB), which is a group of people responsible for configuration management, headed by the configuration manager. For smaller projects, the CCB might consist of just one person. A change is initiated by a change request.

The reason for change can be anything. However, the most common reasons are requirement changes, changes due to bugs, platform changes, and enhancement changes. The CR for change generally consists of three parts. The first part describes the change, reason for change, the SCIs that are affected, the priority of the change, etc.

The second part, filled by the CM, describes the decision taken by the CCB on this CR, the action the CM feels need to be done to implement this change and any other comments the CM may have. The third part is filled by the implementer, which later implements the change.

3. Status accounting and auditing:

For status accounting, the main source of information is the CRs and FRs themselves. Generally, a field in the CR/FR is added that specifies its current status. The status could be active, complete, or not scheduled. Information about dates and efforts can also be added to the CR, the information from the CRs/FRs can be used to prepare a summary, which can be used by the project manager and the CCB to track all the changes.

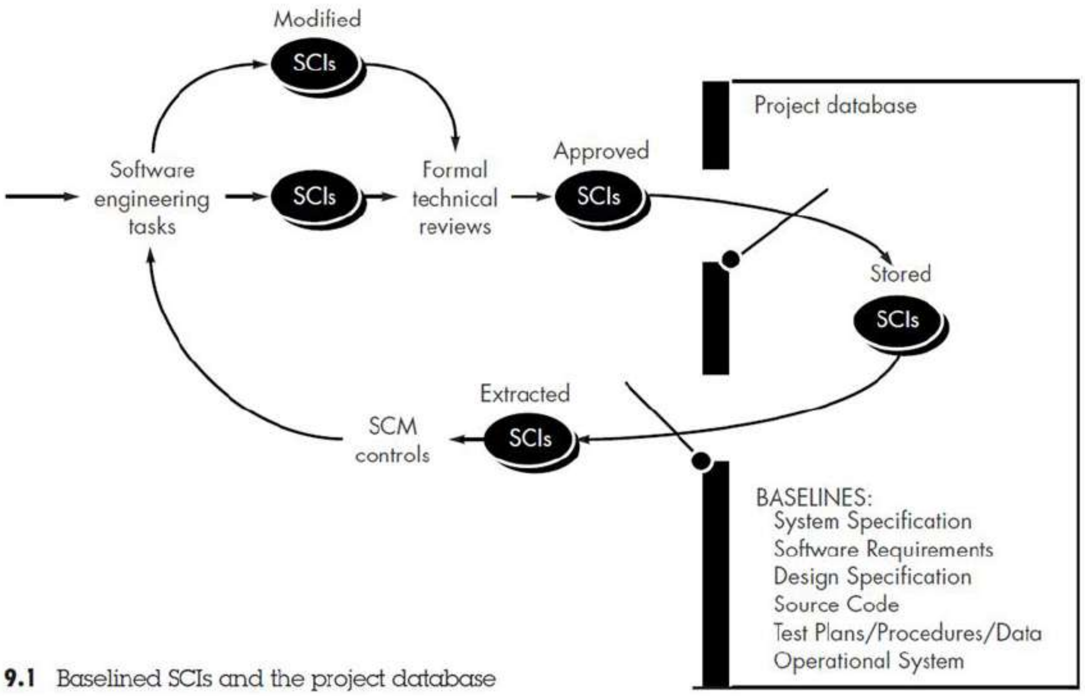


FIGURE 9.1 Baselined SCIs and the project database

CHAPTER 4: Software Requirement and Specification

4.1. Functional and Non Functional Requirement

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

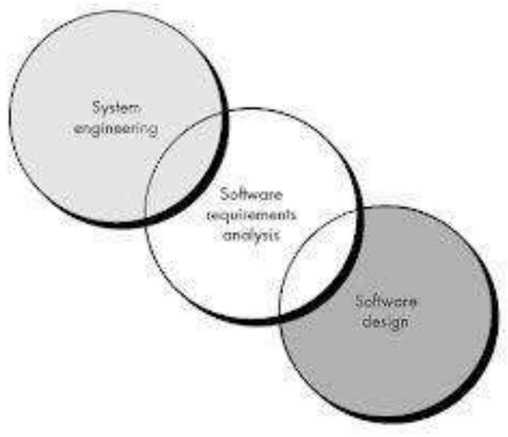


Figure: The analysis model as a bridge between System and design model

Requirements set out what the system should do and define constraints on its operation and implementation. The Requirement engineering occurs during the customer communication and modelling activities that we have defined for the generic software process. Seven distinct requirement engineering functions are

1. **Inception** – How does the software project get started?
2. **Elicitation** – Ask the customer, the user what the objectives of the system or a product?

Why requirements elicitation is difficult:

Problems of scope: define the boundary of the problem

Problem of understanding: The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

Problems of volatility: The requirements change over time.

3. **Elaboration** – The information obtained from inception and elicitation is expanded and refine during elaboration.
4. **Negotiation** – The business resources are limited. So May arise conflict and the system must reconcile through a process of negotiation.
5. **Specification** – It is different for different people. Specification is a written document, a set of graphical models, a formal mathematical model, a collection of usage scenario, a prototype.
6. **Validation** – The requirement validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omission, and error have been detected and Corrected.
7. **Management** – It is the set of activities that helps project team identify, control and track requirements and change to requirement at any time as the project proceed.

Functional Requirement:

- ➢ Functional requirements set out services the system should provide. (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 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.

Non-functional requirements

- Non-functional requirements constrain the system being developed or the development process.
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional classifications

1. Product requirements
   - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
2. Organisational requirements
   - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
3. External requirements
   - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

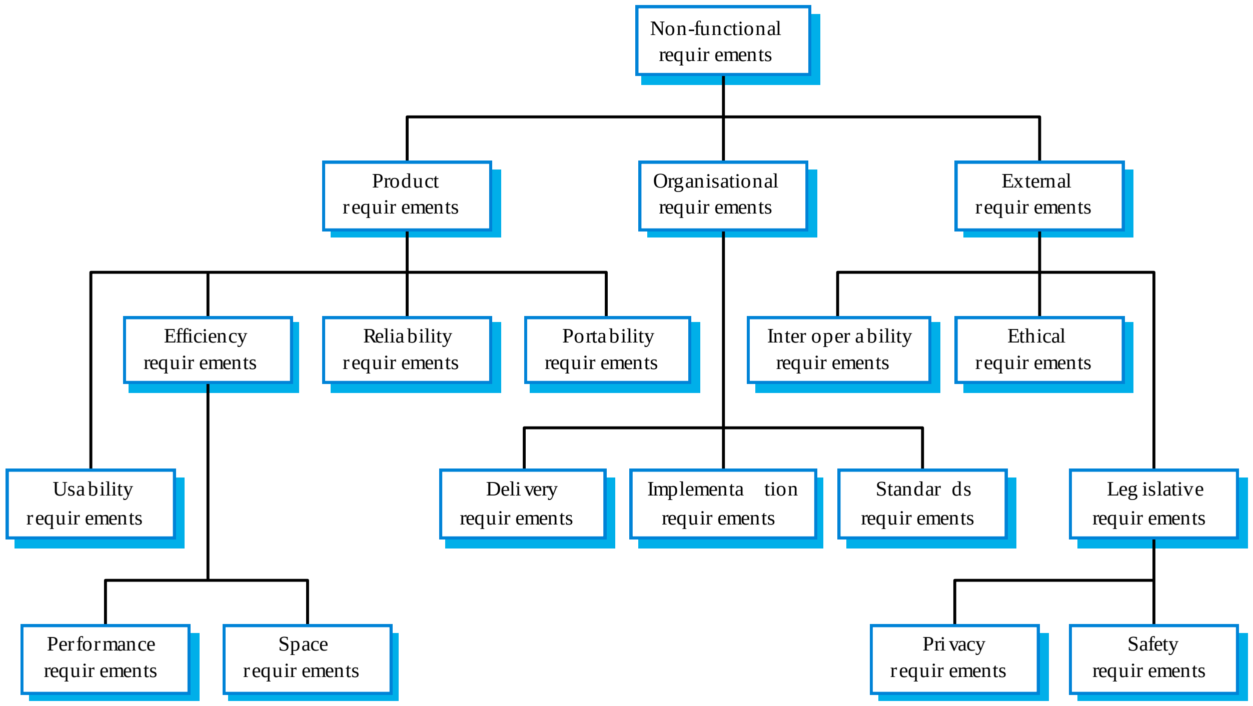

Non-functional requir ements
Product requir ements
Organisational requir ements
External requir ements
Efficiency requir ements
Relia bility requir ements
Porta bility requir ements
Inter oper a bility requir ements
Ethical requir ements
Usa bility requir ements
Deli very requir ements
Implementa tion requir ements
Standar ds requir ements
Leg islative requir ements
Per for mance requir ements
Space requir ements
Pri vacy requir ements
Safety requir ements

Fig: Types of non functional Requirement

User Requirements

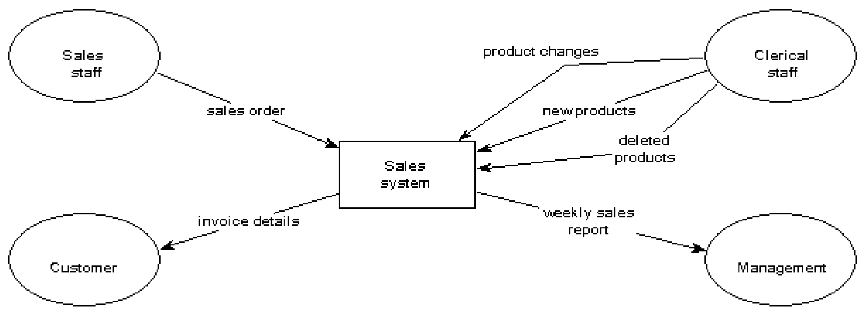
User requirements are high-level statements of what the system should do. User requirements should be written using natural language, tables and diagrams. User requirement are written for the users and include functional and non functional requirement.

System requirements

System requirements are intended to communicate the functions that the system should provide. System requirement are derived from user requirement. The user system requirements are the parts of software requirement and specification (SRS) document.

2.3 Data Model

A **data model** organizes data elements and standardizes how the data elements relate to one another. Since data elements document real life people, places and things and the events between them, the data model represents reality, for example a house has many windows or a cat has two eyes. Computers are used for the accounting of these real life things and events and therefore the data model is a necessary standard to ensure exact communication between human beings. Data models are often used as an aid to communication



between the business people defining the requirements for a computer system and the technical people defining the design in response to those requirements. They are used to show the data needed and created by business processes.

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

Data objects. A *data object* is a representation of almost any composite information that must be understood by software.

Context diagram

The context diagram helps to define our system boundary to show what is included in, and what is excluded from, our system.

Figure: An example context diagram

Data Flow Diagram

Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system.

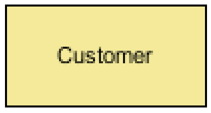
The notation used in these models represents

- *External Entity:* External to the System

- *rounded rectangles:* functional processing
- *rectangles:* data stores
- *Labeled arrows:* Data movements between functions

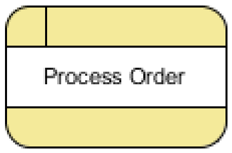
External Entity

An external entity can represent a human, system or subsystem. It is where certain data comes from or goes to. It is external to the system we study, in terms of the business process. For this reason, people use to draw external entities on the edge of a diagram.



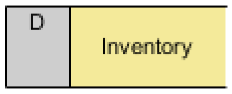
Process

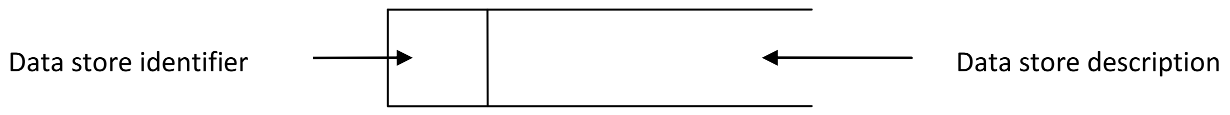
a process is a business activity or function where the manipulation and transformation of data takes place. A process can be decomposed to finer level of details, for representing how data is being processed within the process.

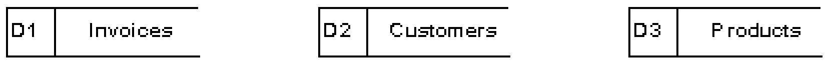


Data Store

A data store represents the storage of persistent data required and/or produced by the process. Here are some examples of data stores: membership forms, database table, etc.



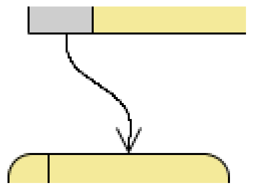




Data Flow

a data flow represents the flow of information, with its direction represented by an arrow head that shows at the end(s) of flow connector.

Applicant's name
Customer details
Employee record
Payment



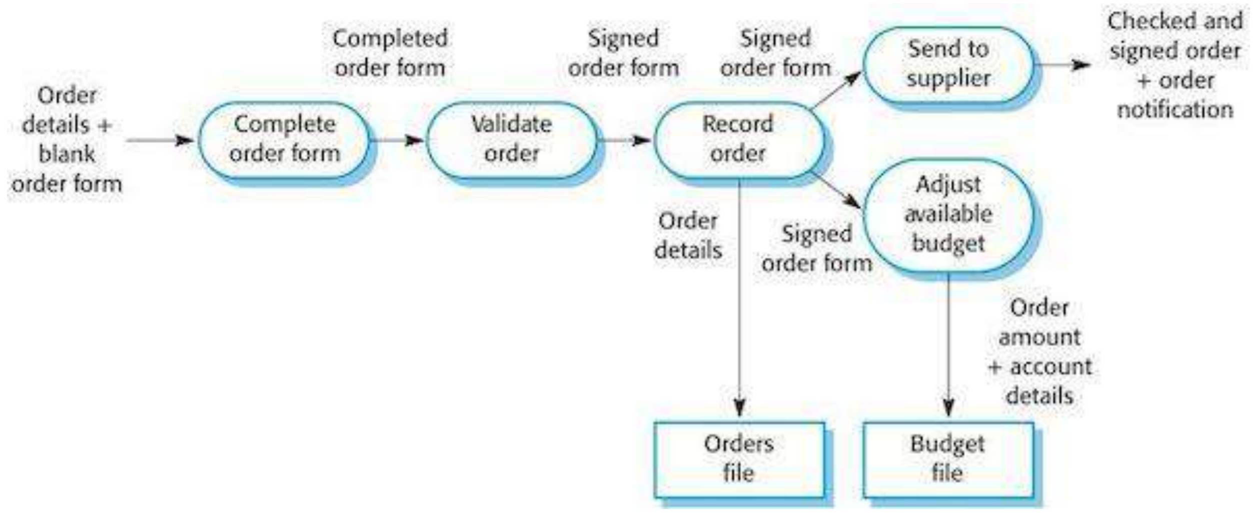
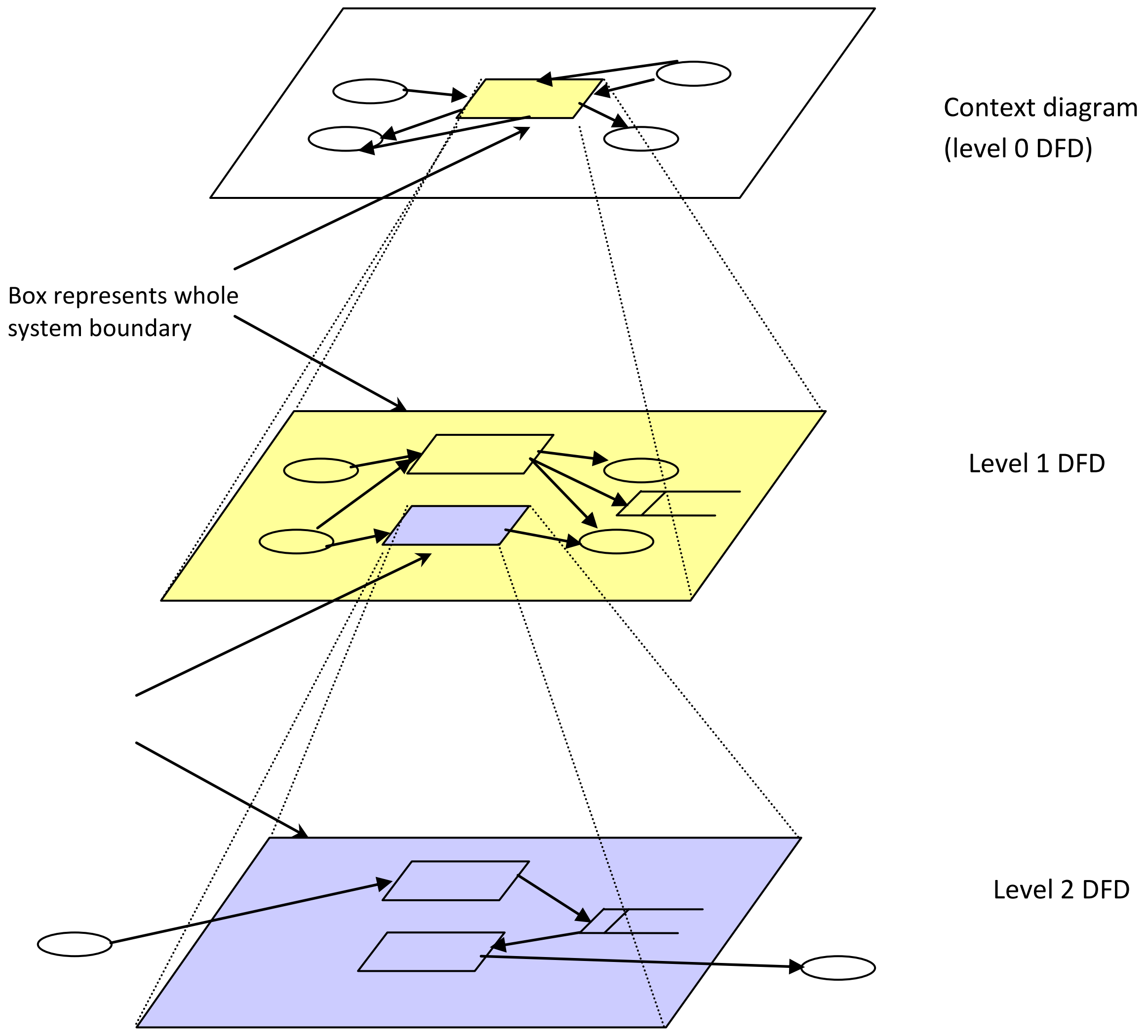


Figure: Data flow diagram of order processing

Data-flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system helps analysts understand what is going on. Data-flow diagrams have the advantage that, unlike some other modeling notations, they are simple and intuitive. It is usually possible to explain them to potential system users who can therefore participate in validating the analysis.

The development of models such as data-flow models should be a 'top-down' process. In this example, this would imply that you should start by analyzing the overall procurement process. You then move on to the analysis of sub-processes such as ordering. In practice, analysis is never like that. You learn about several different levels at the same time. Lower-level models may be developed first then abstracted to create a more general model.



Data flow diagrams usually occur in sets. The set consist of different levels.

In this DFD (level 0 DFD) no processes or data stores are shown.

Another diagram (level 1 DFD) is now developed which shows the processes taking place to convert the inputs shown in the context diagram to the outputs. In this DFD detail is given to show which processes are responsible for accepting the different inputs and producing the different outputs. Any process shown that is complicated by a number of data flows and requires further refinement is shown on another diagram (level 2 DFD) where sub-processes are shown together with any necessary extra data stores.

UML

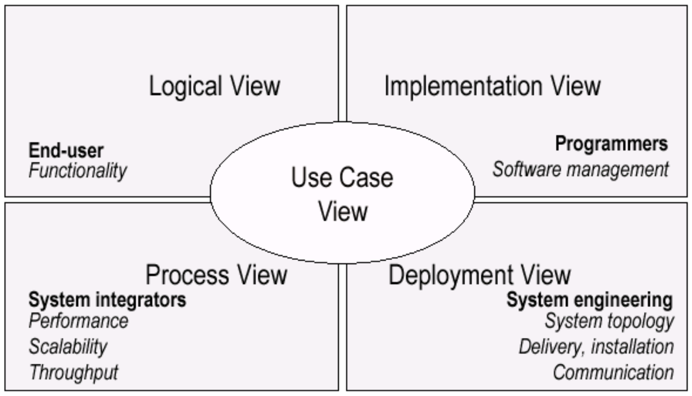
The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software intensive system.

UML provides a wide array of diagrams that can be used for analysis and design at both the system and software level. Developed by the "Three Amigos": Grady Booch, Jim Rumbaugh, Ivar Jacobson

The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

Even though it is expressive, the UML is not difficult to understand and to use. Learning to apply the UML effectively starts with forming a conceptual model of the language, which requires learning three major elements: the UML's basic building blocks, the rules that dictate how these building blocks may be put together, and some common mechanisms that apply throughout the language.

The UML is only a language and so is just one part of a software development method. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.



The UML is a language for

· **Visualizing** – UML is not just a collection of symbols. Behind each symbol is some semantics.

· **Specifying** – the building models are precise, unambiguous and complete. UML addresses the specification of all the important analysis, design and implementation decisions that must be made in developing and deploying a software intensive system.

· **Constructing –** Model are map using programming language like JAVA, C++ or VB6 etc

· **Documenting** – UML provides a language for expressing requirements and for tests. UML also provides a language for modeling the activities of project planning and release management.

A *modeling* language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for software blueprints.

The goal of UML is to provide a standard notation that can be used by all object oriented methods. Actually system development focuses on three different models

1. Functional model : Use case diagram
2. Object model : class diagram
3. Dynamic model: interaction diagram, statechart diagrams and activity diagrams.

Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for such domains as

· Enterprise information systems

· Banking and financial services

· Telecommunications

· Transportation

· Defense/aerospace

· Retail

· Medical electronics

· Scientific

· Distributed Web-based services

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. **Things** – Things are the abstractions that you use to write well formed models. They are the basic object oriented building blocks of UML.
2. **Relationships**
3. **Diagrams**

Relationships in the UML

There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

1. *Dependencies*

A *dependency* is a semantic relationship between two things in which a change to one thing (The independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label,

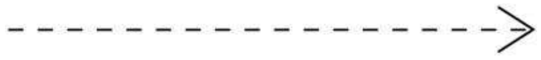
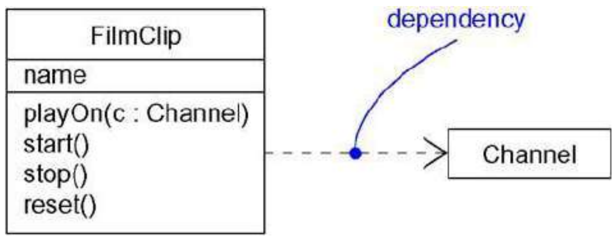


Fig Showing Dependencies



2. *Associations*

An *association* is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names,

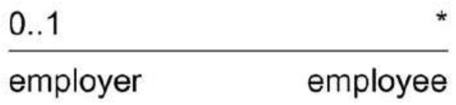
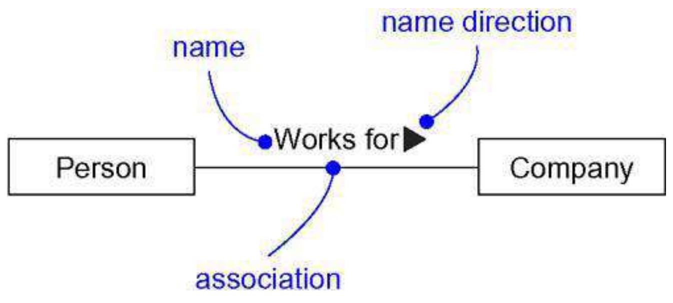


Fig: Association



3. *Generalizations*

A *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

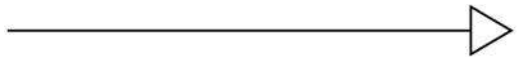
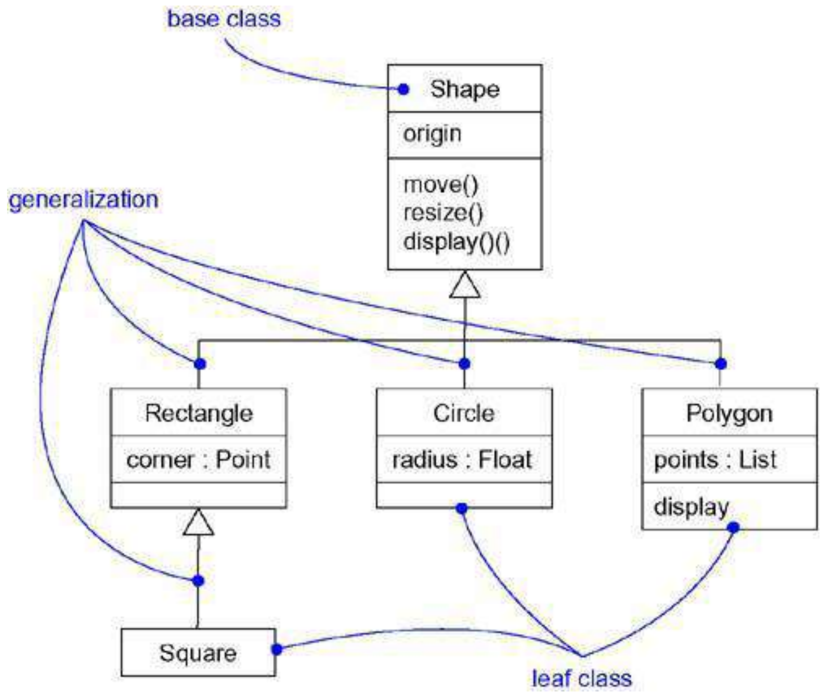


Fig: generalization



4. *Realizations*

A *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship.

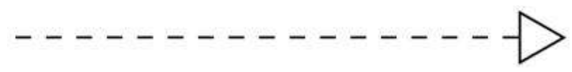


Fig: realization

Things in the UML

There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

1. Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things. Structural things are

1.1 Class

1.2 Interface

1.3 Collaboration

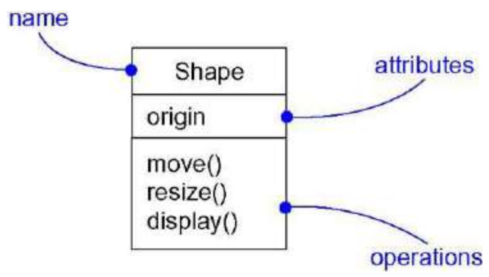
1.4 Use Case

1.5 Active Class

1.6 Component

1.7 Node

1.1 Class: It is a description of set of objects that share the same attributes, operation, relationship and semantics.



1.2 Interface: it is collection of operations that specify a service of a class or component. It describe an externally visible behavior of that element. It may represent the complete behavior of a class or component or only a part of that behavior, it defines a set of operation specifications (i.e their signatures) but never a set of operation implementation.

1.3 Collaboration: it defines and interaction. It is a society of roles and other element that work together to provide some cooperative behavior that is bigger than the sum of all the elements. So collaboration diagram have structural behavior as well as behavioral dimensions. A given class might be participating in several collaborations.

1.4 Use Case: it is a description of set of sequence of actions that a system performs that yield an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by collaboration.

1.5 Active Class: it is a class whose object own one or more processes or threads and thus can initiate control activity.

1.6 Component: it is physical and replaceable part of a system that conforms to and provides the realization of a set of interface. It is physical packaging of other logical elements like classes, interfaces and collaborations.

1.7 Node: it is a physical element that represents computational resources that has some memory and some processing capability.

2. Behavioral Things

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

2.1 Interaction

2.2 State Machine

2.1 interactions: it is a behavior that comprises of a set of message exchange among a set of objects within a particular context to accomplish a specific purpose. It involves a number of other elements like messages, action sequences and links.

2.2 State Machine: it is a behavior that specifies the sequences of states an object or interaction goes through during its life time in response to events together with its responses to those events. A state machine involves a number of other elements including states, transitions, events and activities.

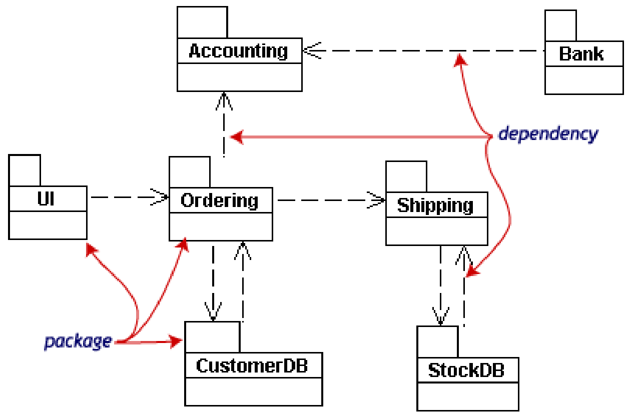
3. Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

3.1 Package

3.1 Package: A *package* is a general-purpose mechanism for organizing elements into groups.

Package = Structural things + behavioral things + annotational things



4. Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

4.1 Notes

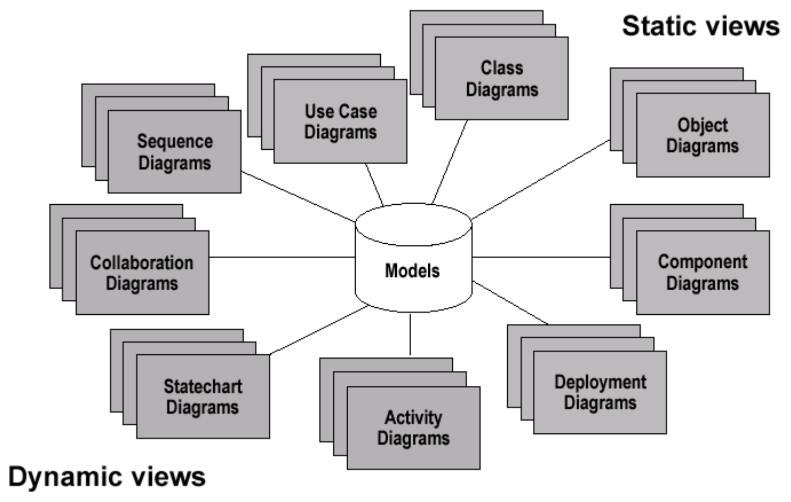
Diagrams in UML

It is a graphical presentation of a set of elements. It is also connected graph vertices (things) and arcs (relationship). The diagrams are used to visualize the system so diagrams are the projection of a system.

UML includes nine such diagrams:

a. Structural Diagram or static diagram
1. Class diagram
2. Object diagram
3. Component diagram
4. Sequence diagram
5. Deployment diagram

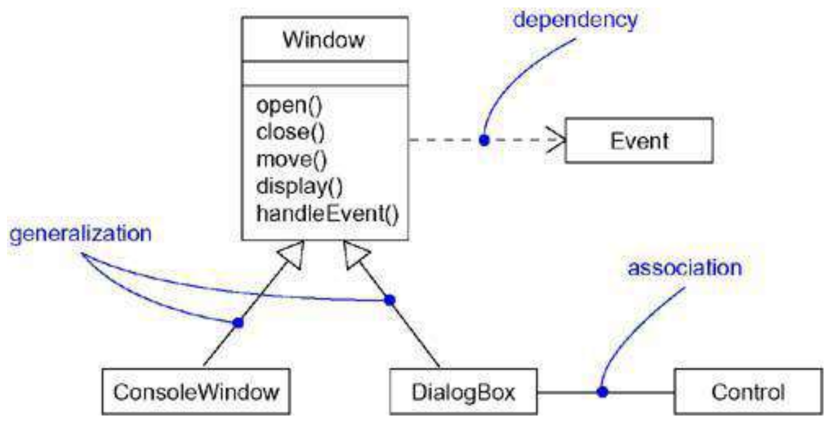
b. Behavioral Diagrams or dynamic diagrams
   1. Statechart diagram
   2. Activity diagram
   3. Use case diagram
   4. Collaboration diagram

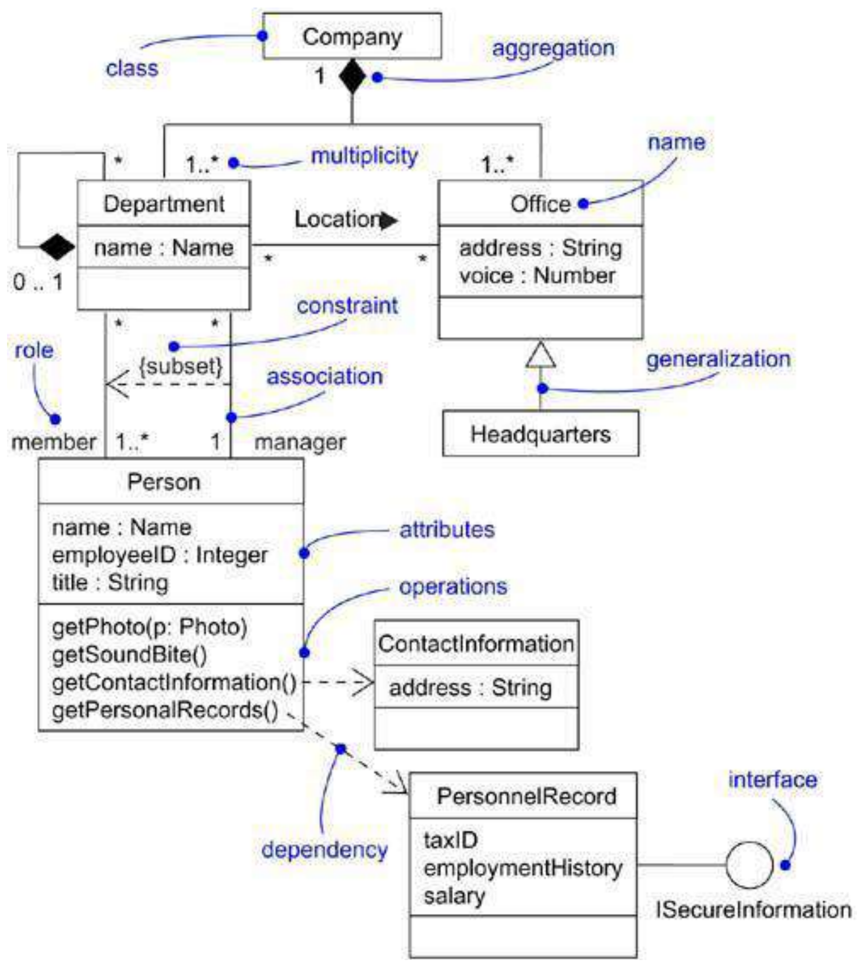


1. *Class diagrams*

A *class diagram* shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

Classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. *A class name must be unique within its enclosing package,*





With the UML, you use class diagrams to visualize the static aspects of these building blocks and their relationships and to specify their details for construction. Graphically, a class diagram is a collection of vertices and arcs.

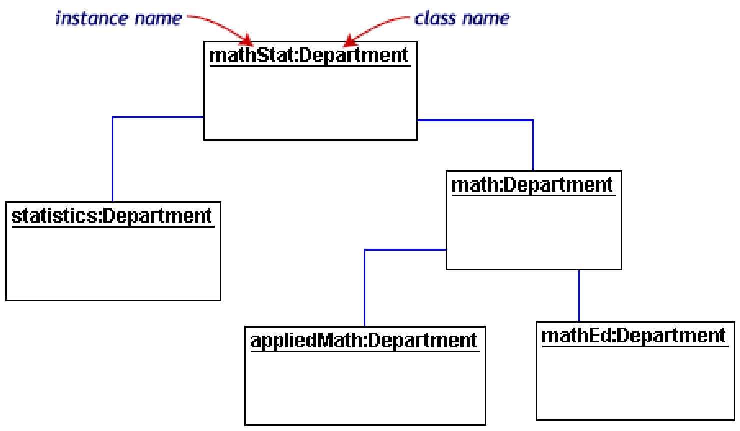
Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Like all other diagrams, class diagrams may contain notes and constraints. Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your class diagrams, as well, especially when you want to visualize the (possibly dynamic) type of an instance.

2. *Object diagrams*

An *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.



3. *Use case diagrams*

A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Use case diagram is created during requirements elicitation process. Use case diagram represents what happens when actor interacts with a system. Use case diagram captures functional and operational aspect of the system. Actors appear outside the rectangle. Use cases within rectangle providing functionality. Relationship association is a solid line between actor & use cases. Use cases should not be used to capture all the details of the system. It represents only the significant aspects of the required functionality.

, use-cases should achieve the following ***Objectives:***

• To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.

• To provide a clear and unambiguous description of how the end-user and the system interact with one another.

• To provide a basis for validation testing.

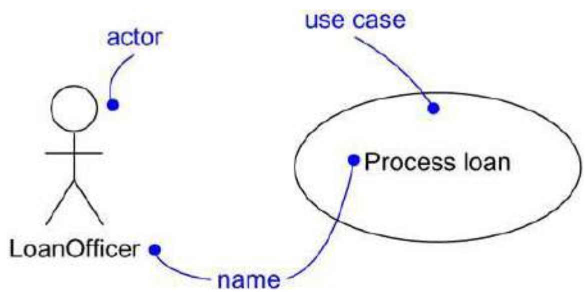
-- represents what happens when actor interacts with a system.

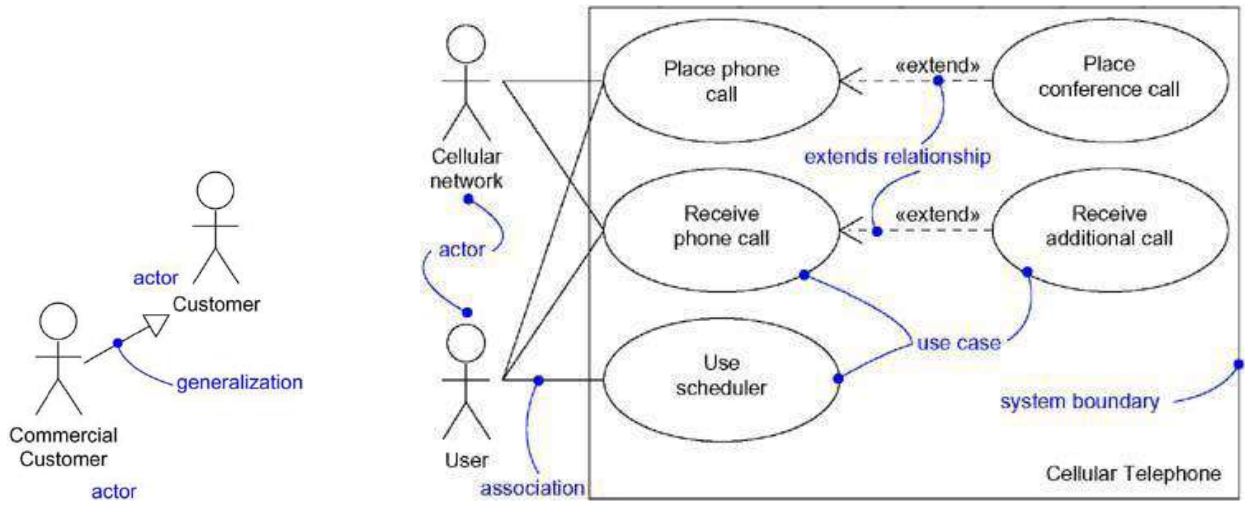
-- captures functional aspect of the system.

-- Actors appear outside the rectangle.

--Use cases within rectangle providing functionality.

--Relationship association is a solid line between actor & use cases.





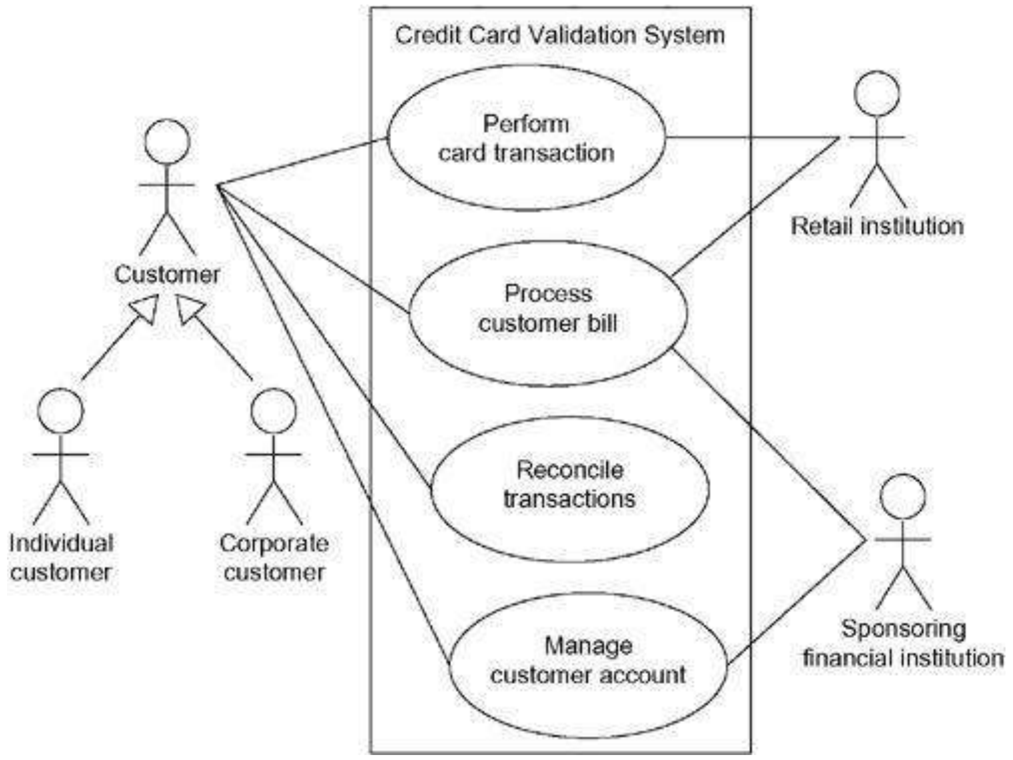


Fig: Use Case Diagram of Credit Card Validation

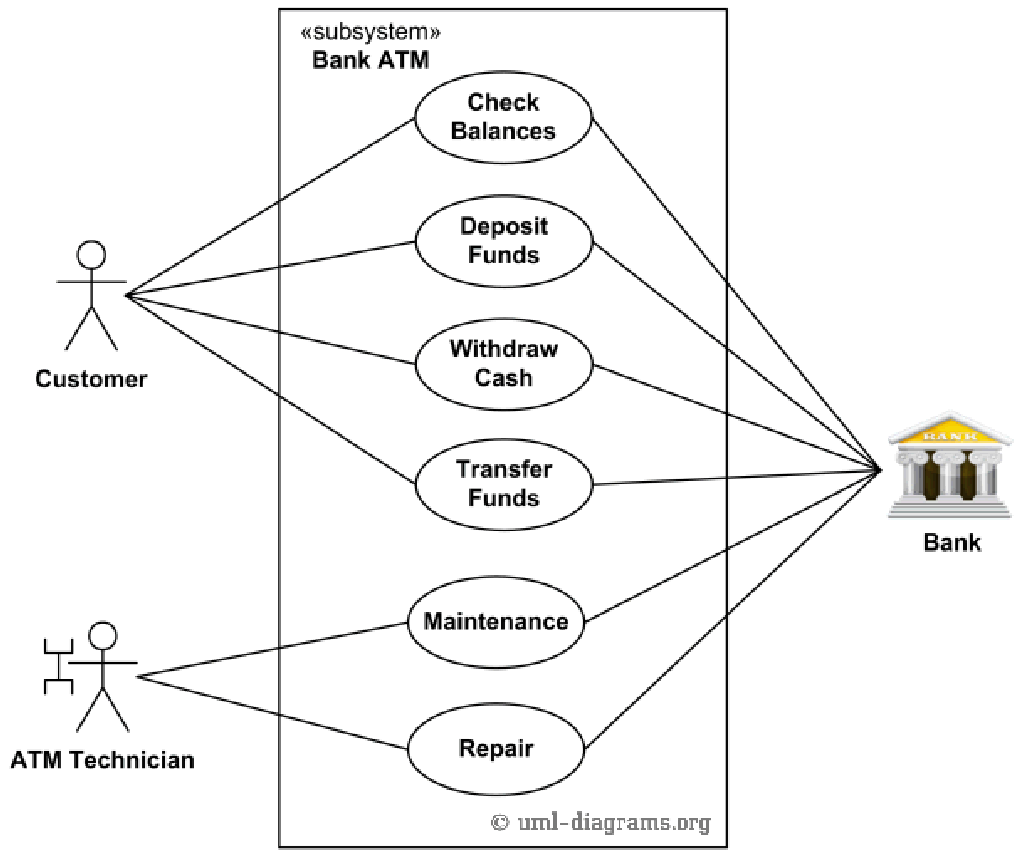
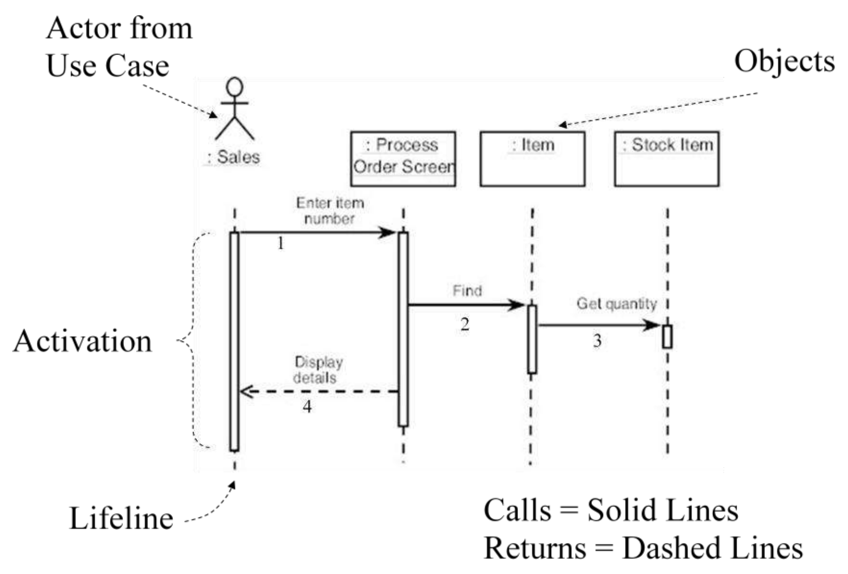


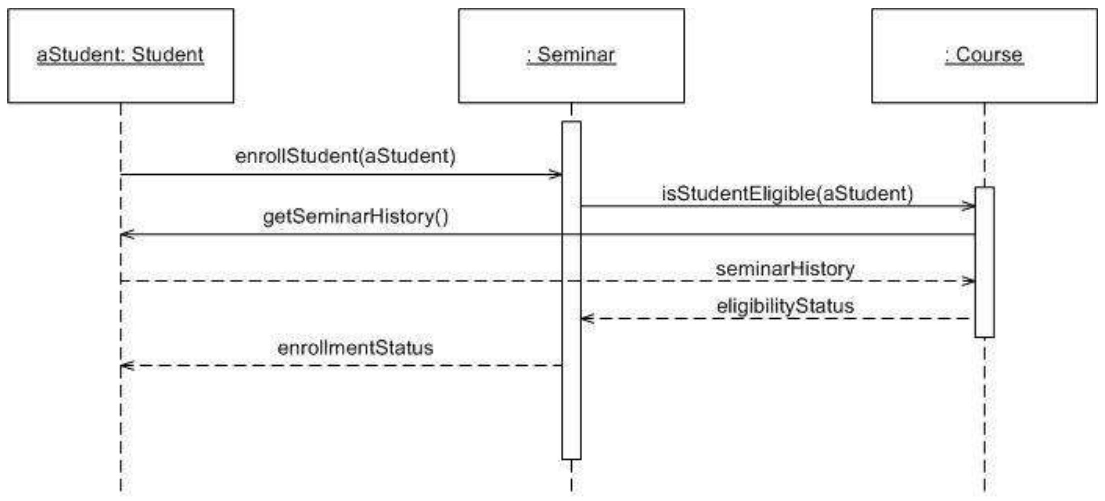
Fig: Use Case Diagram of ATM Machine

4. *Interaction diagrams*

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. It shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages;

Interaction Diagram = Sequence diagram + Collaboration diagram





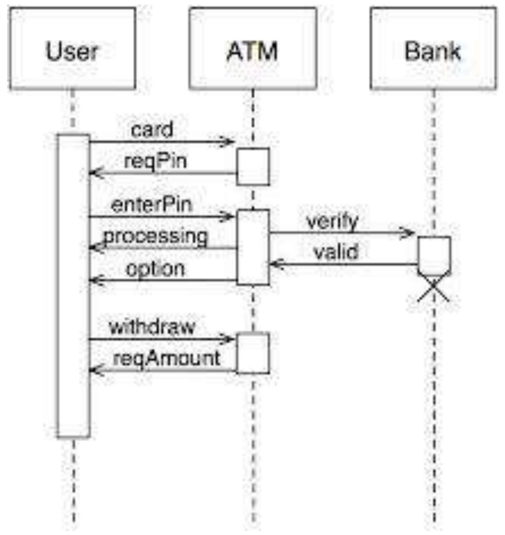


Fig Sequence diagram of ATM Machine

5. Collaboration Diagram

A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

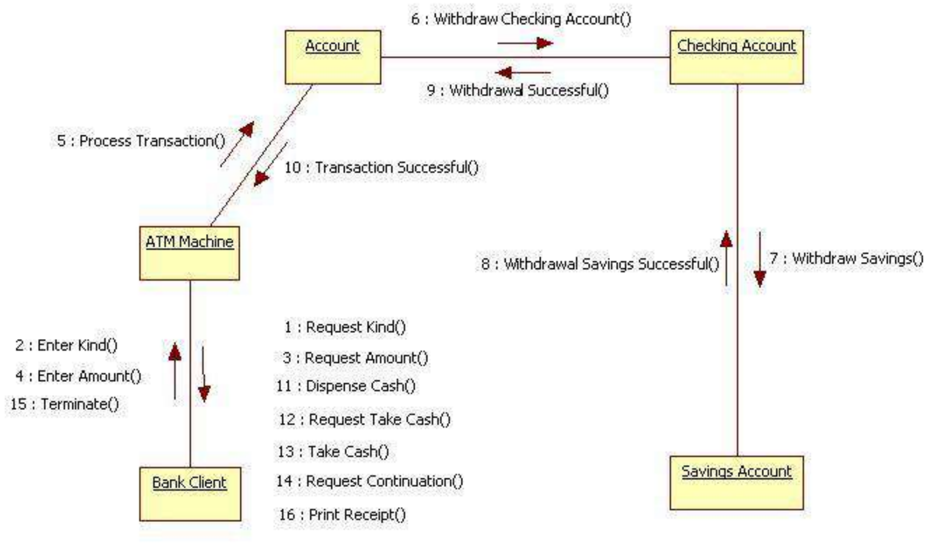


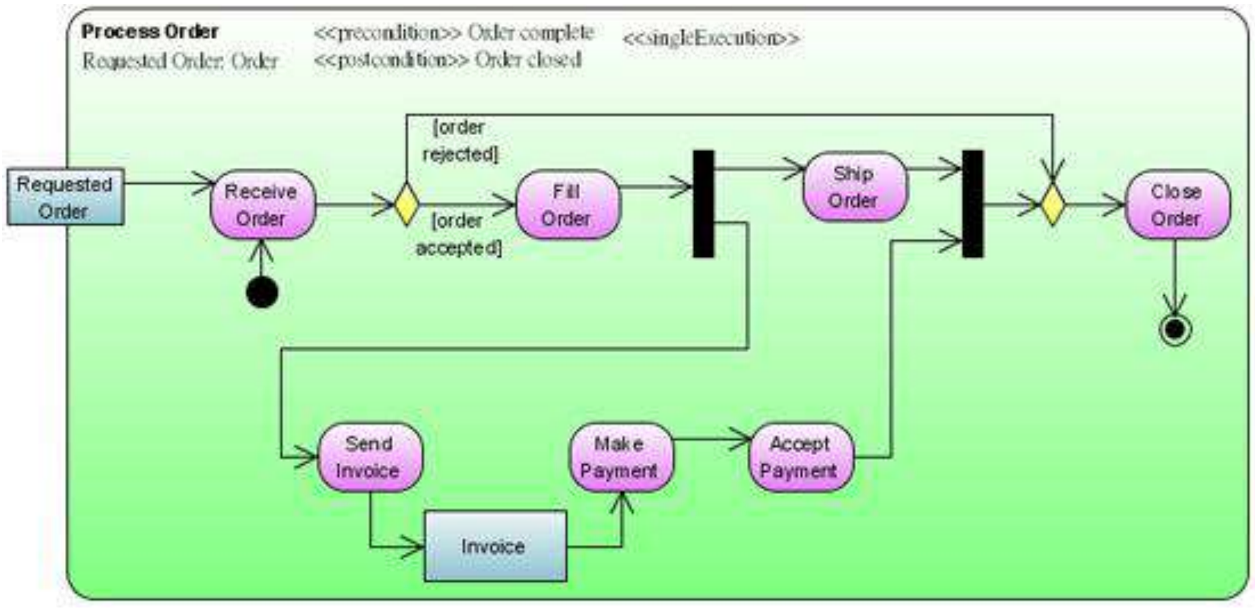
Fig: Collaboration diagram of an ATM Machine

6. *Statechart diagrams*

A *statechart diagram* shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

7. *Activity diagrams*

An *activity diagram* is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.



8. *Component diagrams*

A *component diagram* shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

9. *Deployment diagrams*

A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. They are related to component diagrams in that a node typically encloses one or more components.

4.5 Software Prototyping Techniques

Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Outline of the prototyping process

The process of prototyping involves the following steps

1. Identify basic requirements

Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

2. Develop Initial Prototype

   The initial prototype is developed that includes only user interfaces. (See Horizontal Prototype, below)

3. Review

   The customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. Revise and Enhance the Prototype

   Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 and #4 may be needed.

Advantages of prototyping

1. **Reduced time and costs**: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of *what the user really wants* can result in faster and less expensive software.
2. **Improved and increased user involvement**: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the user's desire for look, feel and performance.

Dimensions of prototypes

3. Horizontal Prototype

A common term for a user interface prototype is the **horizontal prototype**. It provides a broad view of an entire system or subsystem, focusing on user interaction more than low-level system functionality, such as database access. Horizontal prototypes are useful for:

- Confirmation of user interface requirements and system scope

- Demonstration version of the system to obtain buy-in from the business
- Develop preliminary estimates of development time, cost and effort.

4. Vertical Prototype

A **vertical prototype** is a more complete elaboration of a single subsystem or function. It is useful for obtaining detailed requirements for a given function, with the following benefits:

- Refinement database design
- Obtain information on data volumes and system interface needs, for network sizing and performance engineering
- Clarifies complex requirements by drilling down to actual system functionality

4.6 Requirement definition and specification

The introduction to the Software Requirement Specification (SRS) document should provide an overview of the complete SRS document. While writing this document please remember that this document should contain all of the information needed by a software engineer to adequately design and implement the software product described by the requirements listed in this document.

CHAPTER 5: Software Design

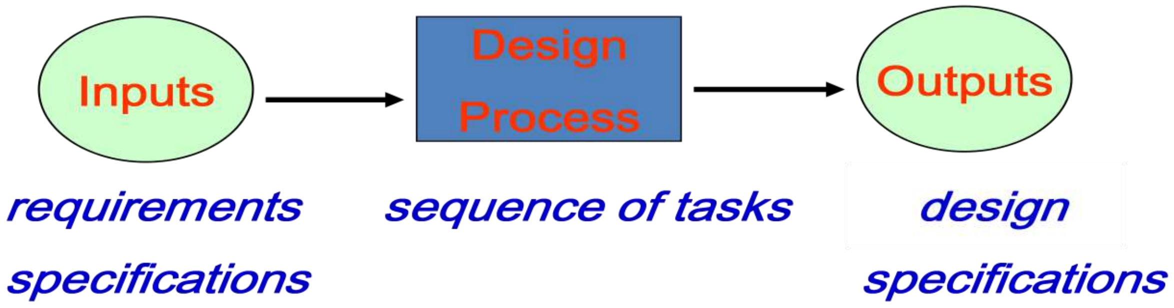
4.2. Introduction to Software Design

"The most common miracles of software engineering are the transitions from analysis to design and design to code."

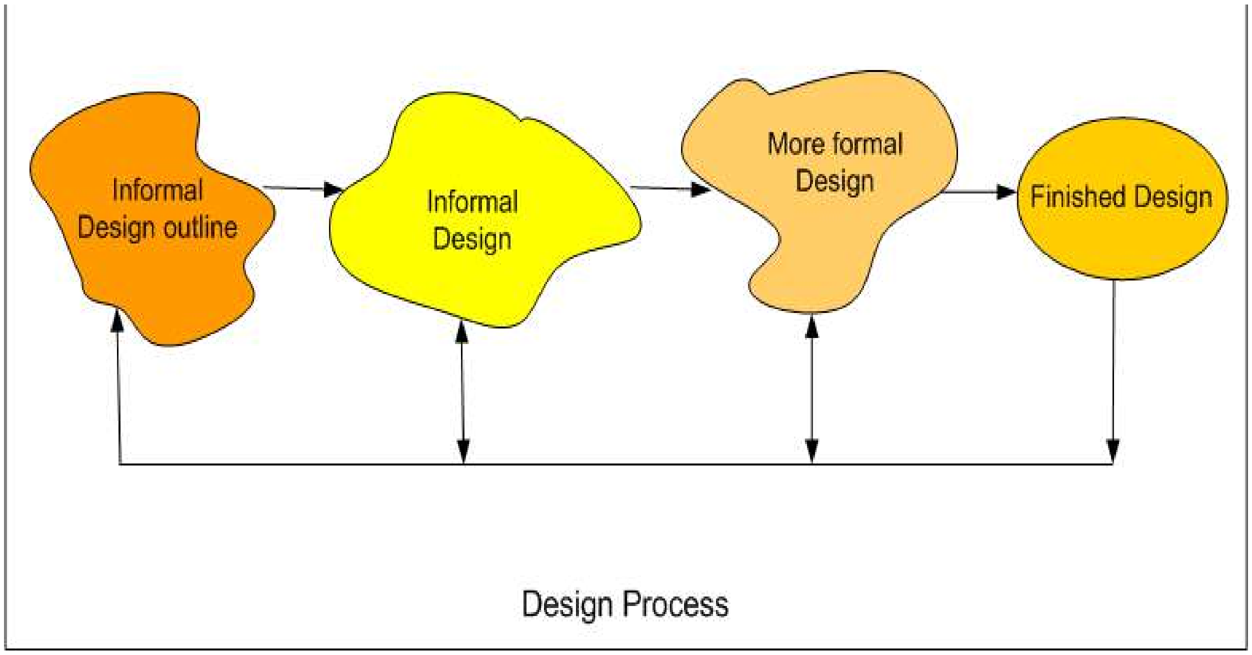
Richard Dué

A Process is a set of related and (sequenced) tasks that transforms a set of input to a set of output. Software design is more creative than analysis. So it is a problem solving activity.

The output of software design is Software Design Document (SDD).



Software designers do not arrive at a finished design immediately. They develop design iteratively through number of different versions. The starting point is informal design which is refined by adding information to make it consistent and complete as shown in the figure below:



As a design is decomposed, errors and omissions in earlier stages are discovered. These feed back to allow earlier design models to be improved.

A specification for the next stage is the output of each design activity.

The final results of the process are precise specifications of the algorithms and data structures to be implemented.

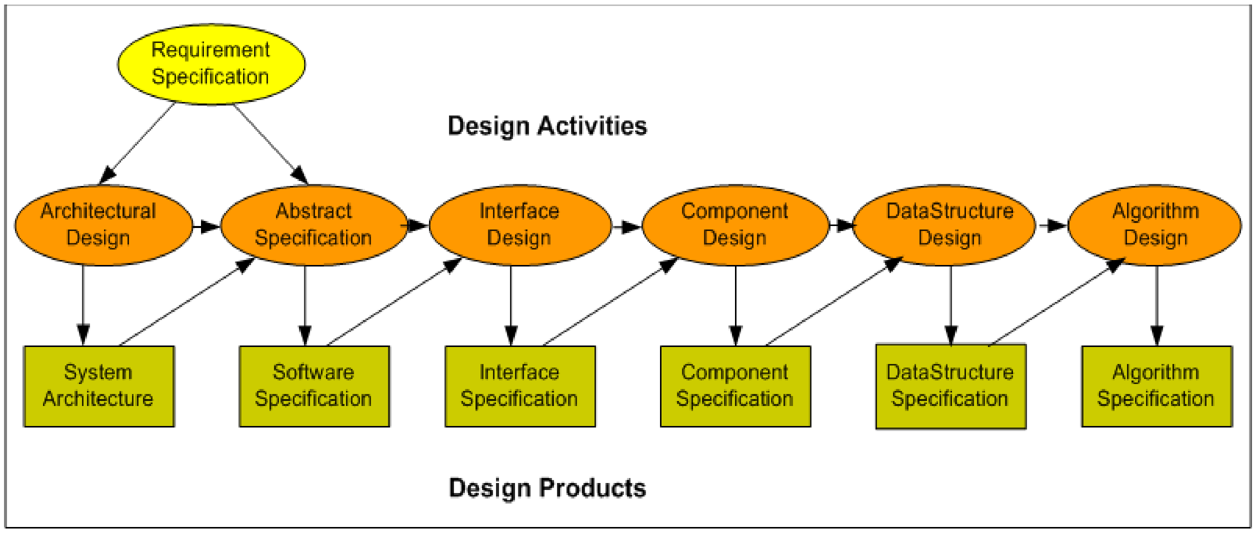
Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design and
- Detailed design.

High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture.

During **detailed design**, the data structure and the algorithms of the different modules are designed. The outcome of the detailed design stage is usually known as the module-specification document.

A general model of software design process



General software Design Technique

- 2 main steps:

– *Analysis of problem* : Breaking Down, Understanding and clearly Defining the Problem

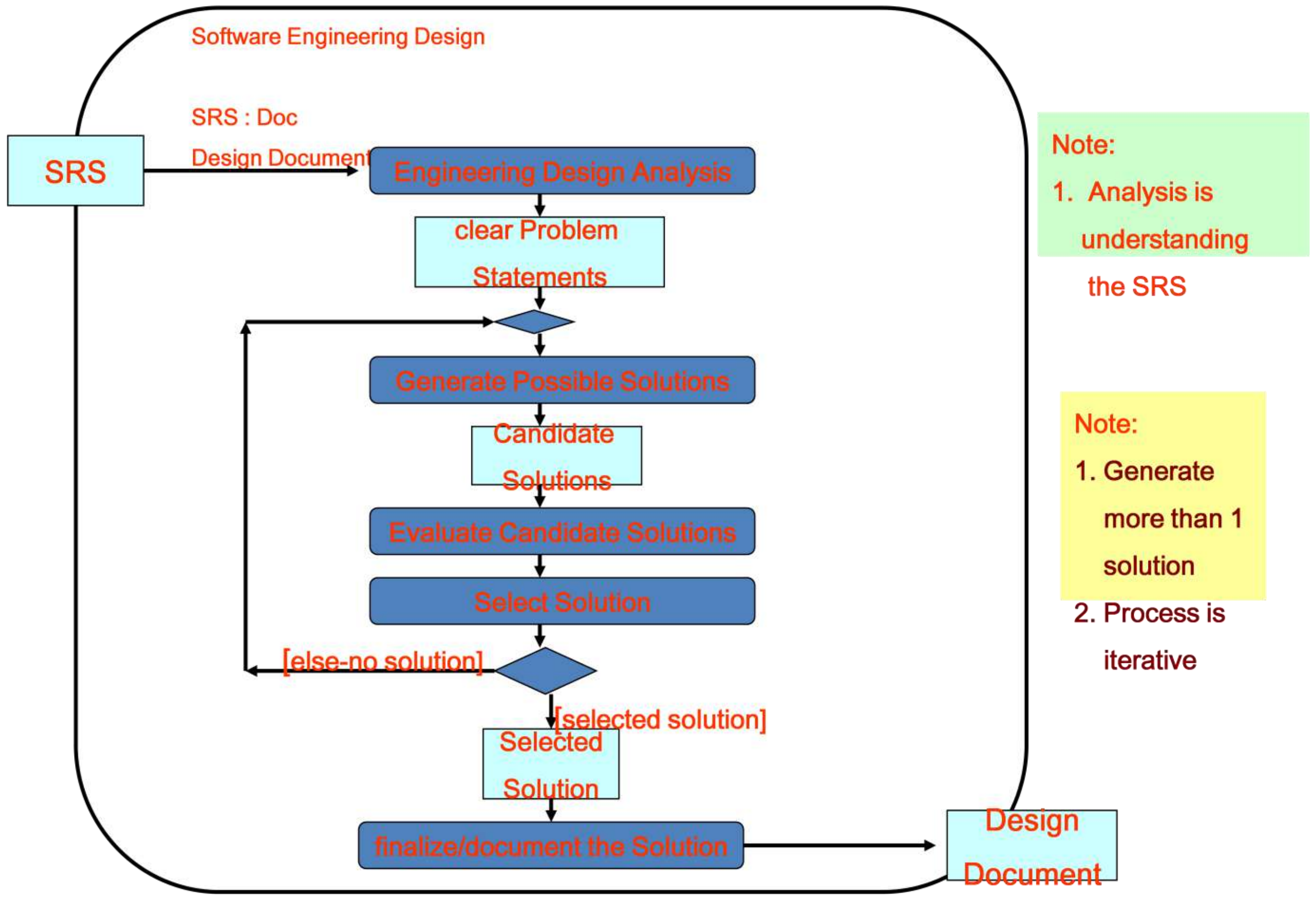
– *Design the solution* : Finding the Solution(s) to the (Sub-)Problem(s) and Constructing the overall solution

1. Understand the problem and clearly defining it. (very important)

   ----

1. Develop possible solution(s)
2. Evaluate the potential solutions
3. If there is a “best” selection, choose it; otherwise iterate through step 2 again.
4. Finalize and document the chosen solution

Software Design Technique (in Activity Diagram)


Software Engineering Design
SRS : Doc
Design Document
SRS
Engineering Design Analysis
clear Problem Statements
Generate Possible Solutions
Candidate Solutions
Evaluate Candidate Solutions
Select Solution
[else-no solution]
[selected solution]
Selected Solution
finalize/document the Solution
Design Document
Note:
1. Analysis is understanding the SRS
Note:
1. Generate more than 1 solution
2. Process is iterative

4.3. Characteristics of a good Software Design

The design needs to be

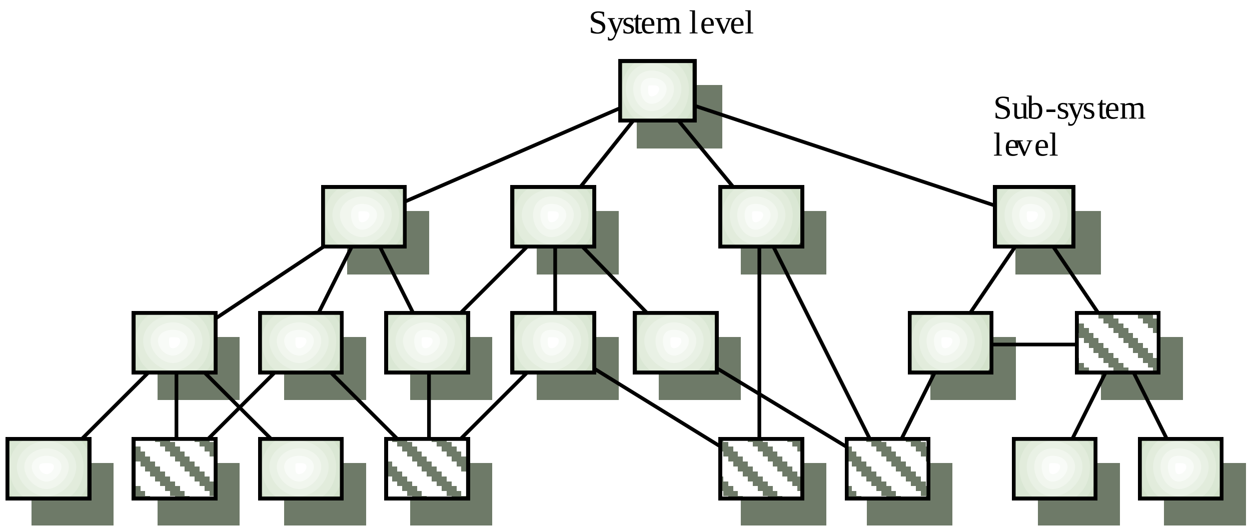
1. Correct & complete
2. Understandable
3. At the right level
4. Maintainable

4.4. Design Principle

Top-down Design

In principle, top-down design involves starting at the uppermost components in the hierarchy and working down the hierarchy level by level.

In practice, large systems design is never truly top-down. Some branches are designed before others. Designers reuse experience (and sometimes components) during the design process.



A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Bottom-Up Design

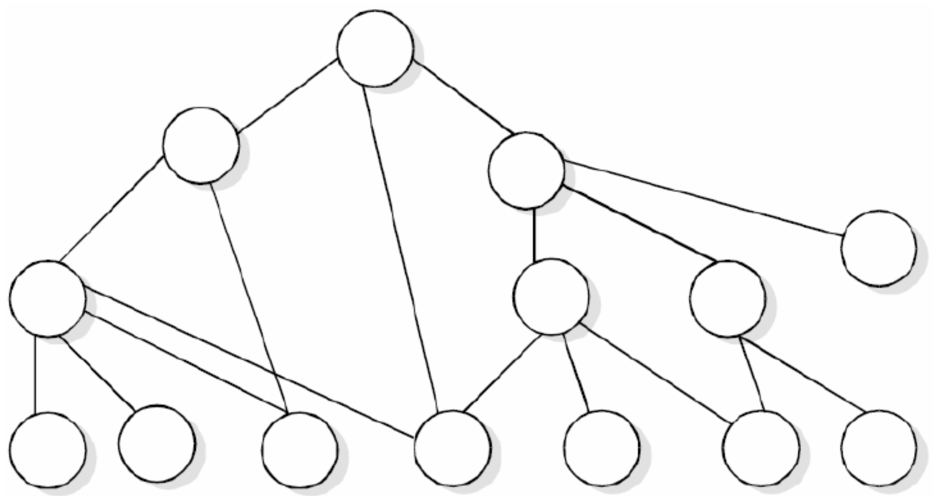


Fig: Bottom-up tree structure

Hybrid Design

For top-down approach to be effective, some bottom-up approach is essential for the following reasons:

_ to permit common sub modules.

_ near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there is more number of modules at low levels than high levels.
_ in the use of pre-written library modules, in particular, reuse of modules.

4.5. Design concepts

1. Abstraction: When we consider a modular solution to any problem, many *levels of abstraction* can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken.
2. Refinement: *Stepwise refinement* is a top-down design strategy
3. **Modularity**

- A system is modular if it is composed of well defined, conceptually simple and independent units interacting through well defined interfaces.
- The following are several advantages of modular systems.

– Modular systems are easier to understand and explain because their parts make sense and can stand on their own.
– Modular systems are easier to document because each part can be documented as n independent unit.
– Programming individual modules is easier because programmer can focus on just one small, simple problem rather than a large complex problem.
– Testing and debugging individual modules is easier because they can be dealt with in isolation from the rest of the program.
– Bugs are easier to isolate and understand, and they can be fixed without fear of introducing problems outside the module.
– Well composed modules are more reusable because they are more likely to comprise part of a solution of many problems. Also a good module should be easy to extract from any one of the program and insert into another.

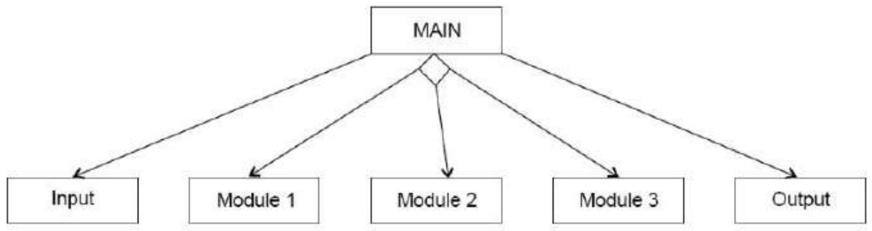


Fig. 19 : Transaction-centered structure

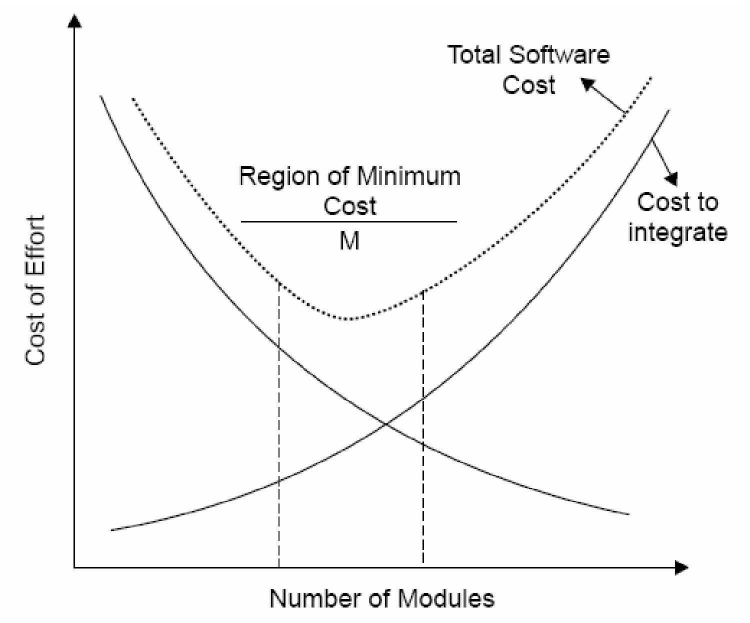
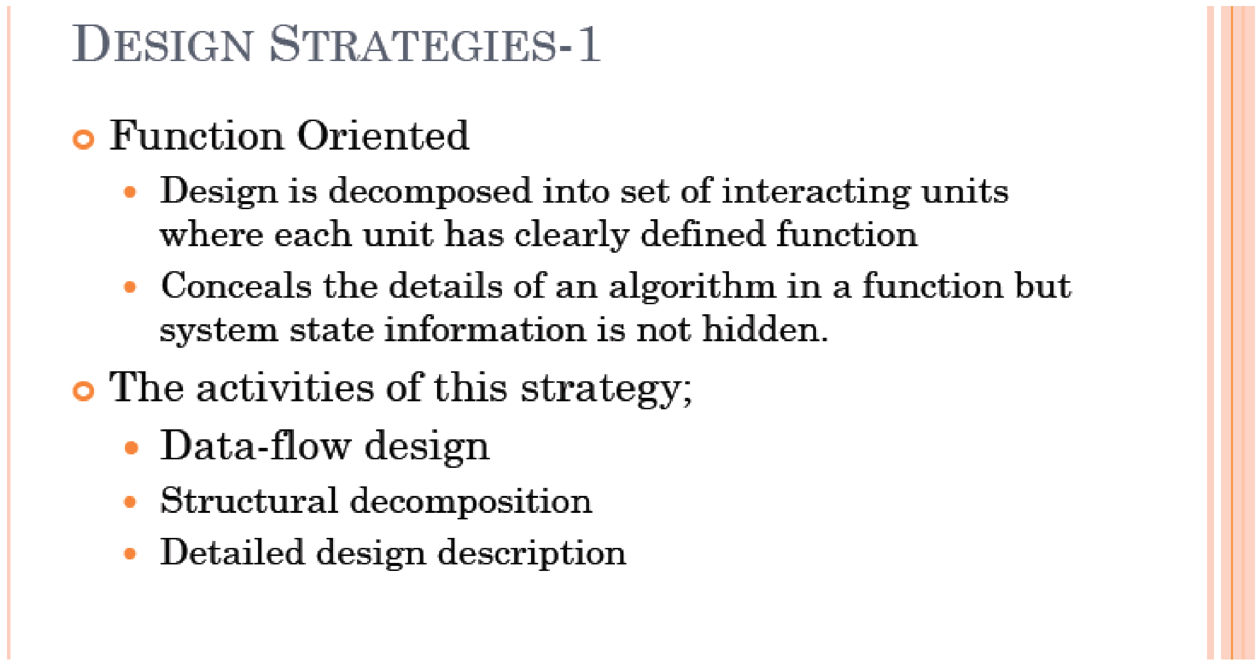


Fig: Modularity and software cost

4. Software Architecture: *Software architecture* alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system"
5. Control Hierarchy: *Control hierarchy,* also called *program structure,* represents the organization of program components (modules) and implies a hierarchy of control.
6. Structural Partitioning: If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 13.4a, *horizontal partitioning* defines separate branches of the modular hierarchy for each major program function.
7. Data Structure: *Data structure* is a representation of the logical relationship among individual elements of data
8. Software Procedure

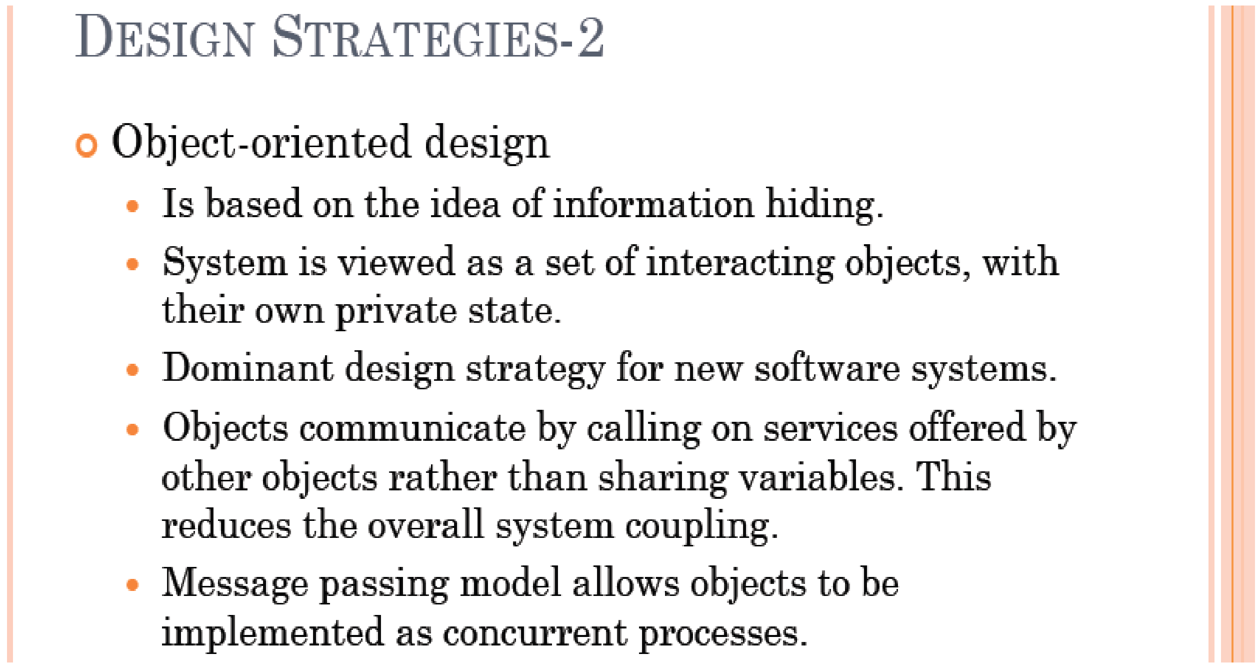
9. Information Hiding

4.6. Design Strategy



DESIGN STRATEGIES-1

- Function Oriented
  - Design is decomposed into set of interacting units where each unit has clearly defined function
  - Conceals the details of an algorithm in a function but system state information is not hidden.
- The activities of this strategy;
  - Data-flow design
  - Structural decomposition
  - Detailed design description



DESIGN STRATEGIES-2

- Object-oriented design
  - Is based on the idea of information hiding.
  - System is viewed as a set of interacting objects, with their own private state.
  - Dominant design strategy for new software systems.
  - Objects communicate by calling on services offered by other objects rather than sharing variables. This reduces the overall system coupling.
  - Message passing model allows objects to be implemented as concurrent processes.

Function oriented design

- Function oriented design is the result of focusing attention to the function of the program.

- This is based on stepwise refinement. Stepwise refinement is a top down strategy where a program is refined as a hierarchy of increasing levels of detail.
- We start with a high level description of what the program does. Then, in each step, we take one part of our high level description and refine it, i.e., specify in somewhat greater detail what that particular does.
- The process should proceed from a highly conceptual model (abstractions) to lower level details.
- The refinement of each module is done until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinement in top down design structure.
- Some of the nodes of the tree will be modules of the program
- Others will be simply statements.

Object oriented design

- It is the result of focusing attention not on the function performed by the program, but instead on the **data** that is to be manipulated by the program.
- OOD is one of the latest approaches to software development and it shows much promise in solving the problems associated with building modern software system.
- **Object**: Software packages designed and developed to correspond with real world entities that contain all the data and services required to function as their associated entities messages.
- Methods: Methods are services that objects perform to satisfy the functional requirements of the problem domain. Object request services of other objects through message passing.
- Benefits of OOD
  - Objects are inherently reusable.
  - The characteristics of information hiding stabilize systems by localizing changes to objects.
  - The concept of objects performing services is a more natural way of thinking.
- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities).
- The state is decentralized among the objects and each object manages its own state information.
- For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data.
- In fact, the functions defined for one object cannot refer or change data of other objects.

- Objects have their own internal data which define their state.
- Similar objects constitute a class.
- In other words, each object is a member of some class. Classes may inherit features from super class.
- Conceptually, objects communicate by message passing.

Function oriented vs. Object oriented design

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc.
- For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc.
- Grady Booch sums up this difference as "identify verbs if you are after procedural design and nouns if you are after object-oriented design"
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system.
- For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it.
- In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.
- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

4.7. Design process and Design quality

Quality: Cohesion:

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is a measure of functional strength of a module.

A measure of how well the parts of a component fit together, *i.e. how functionally related the parts are*

For example, strong cohesion exists when all parts of a component contribute different aspects of related functions

Strong cohesion promotes understanding and reasoning, and thus provides dividends with respect to maintenance and reuse via **separation of concerns**

Cohesion

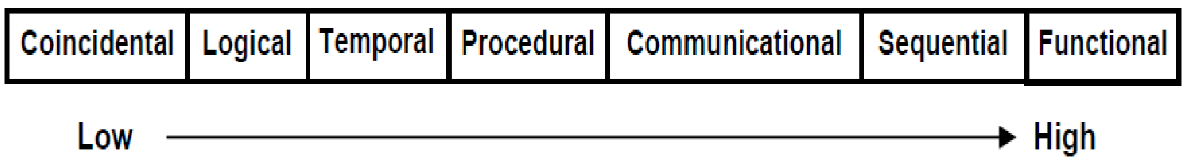


Fig. 4.1: Classification of cohesion

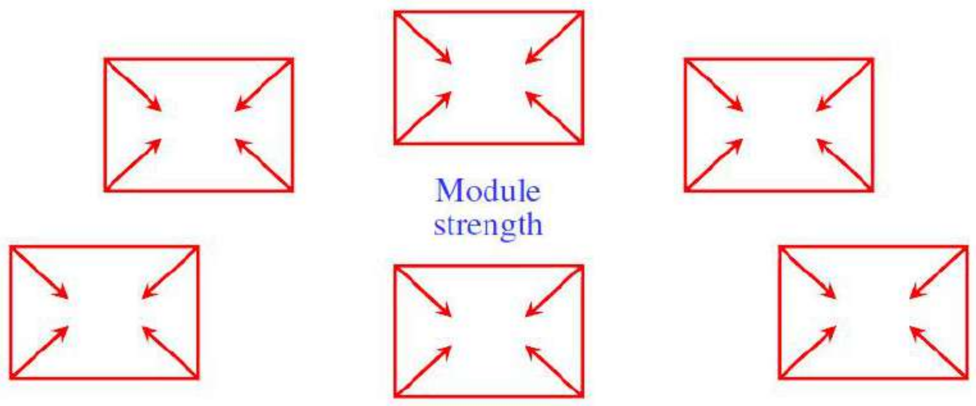


Fig. 10 : Cohesion=Strength of relations within modules

1. **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the get-input, print-error, and

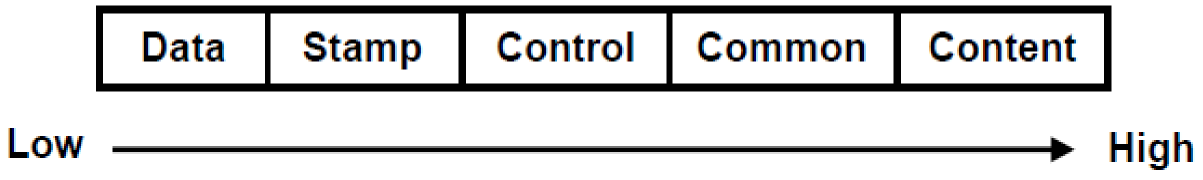
summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

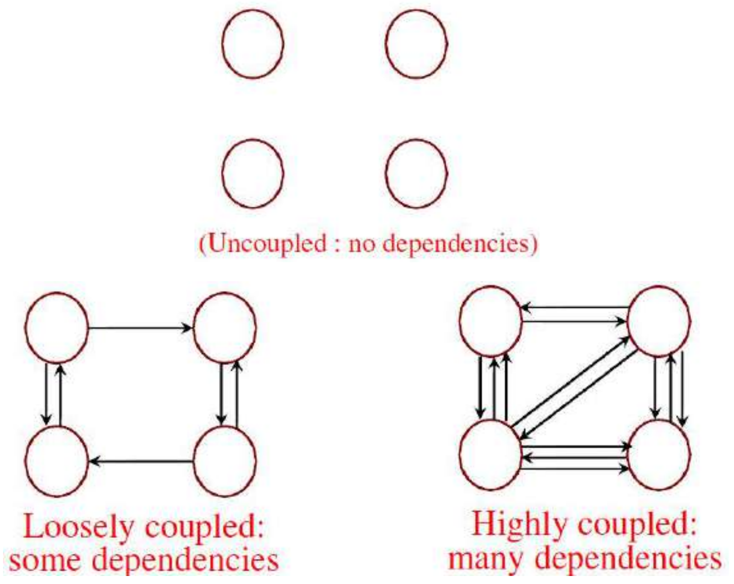
2. **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.
3. **Temporal cohesion:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion
4. **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.
5. **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.
6. **Sequential cohesion:** A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.
7. **Functional cohesion:** Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

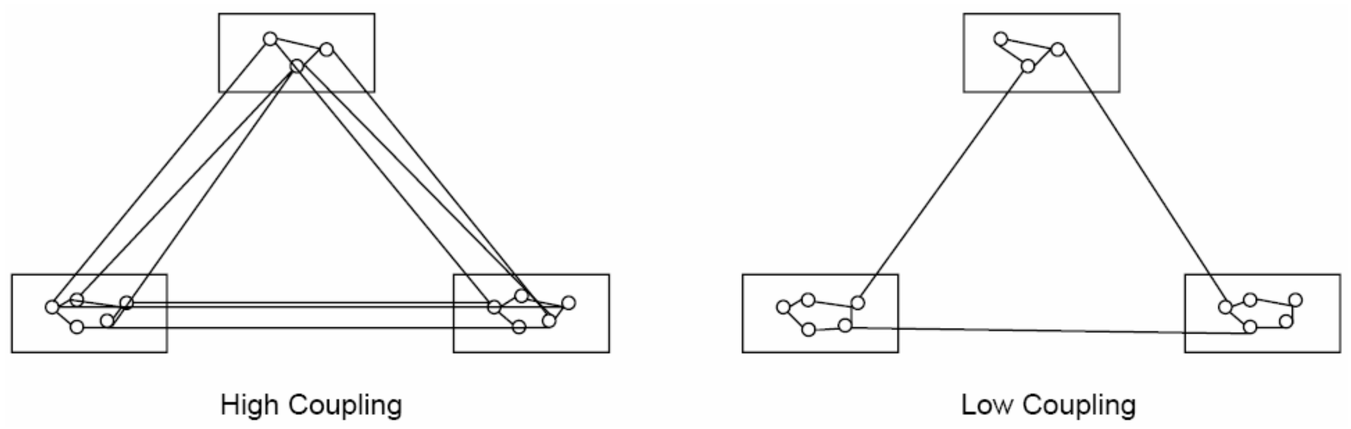
Coupling

- Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

- Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules.







1. **Data coupling:** Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

2. **Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
3. **Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.
4. **Common coupling:** Two modules are common coupled, if they share data through some global data items.

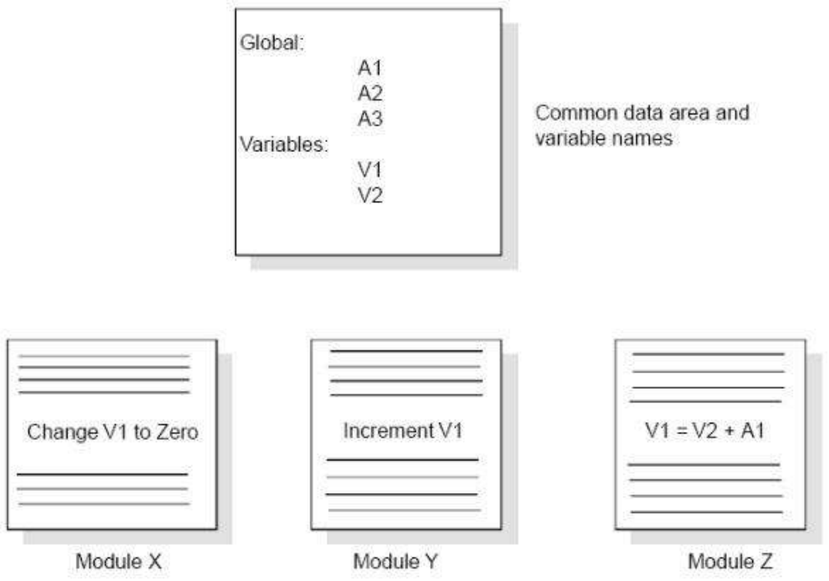


Fig. 8 : Example of common coupling

5. **Content coupling:** Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

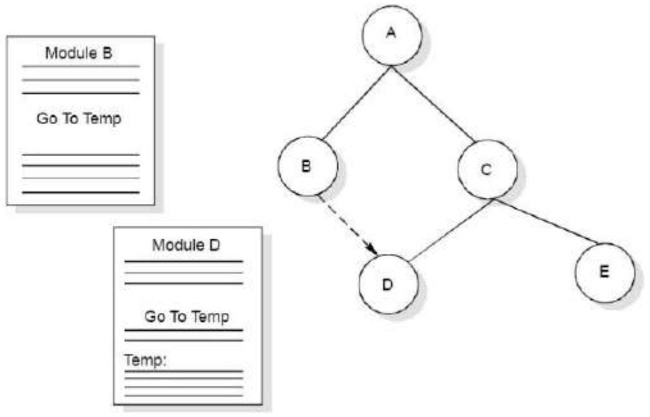


Fig. 9 : Example of content coupling

Functional Independent

- A module having high cohesion and low coupling is said to be functionally independent from other modules.
- By the term functional independence, we mean that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- Functional independence is a key to good design primarily due to the following reasons.

– Functional independence reduces error propagation. Therefore, error existing in one module doesn’t directly affect other modules and also any error existing in other modules doesn't directly affect that particular module.

– Reuse of the module is possible, because each module performs some well defined and precise function and the interface of the module with other modules is simple and minimal. Therefore, any such module can be easily taken out and reused in a different program.

– Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent from each other.

4.8. Software Architecture and its types

The *architectural design* defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements.

✧ The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.

✧ The output of this design process is a description of the software architecture.

✧ The architectural design is concerned with establishing a basic structural framework for a system.

It involves identifying the major components of the system and the communications between those components

CHAPTER 6

6.1 Software Testing

- Introduction

– After detailed design, the <u>coding</u> and <u>integration</u> will begin.

– The source code for each is written and tested.

– The modules are brought together (Integrated) to form the system and tested.

"*Testing is the process of executing a program with the intent of finding errors*."

What is testing?

- Testing is **not** showing that there are no errors in the program.
- Testing **cannot** show that the program performs its intended goal correctly.

So, Testing is the process of executing the program in order to find errors. A successful test is one that finds an error.

Why should we Test?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Who should do the Testing?

- Testing requires the developers to find errors from their software.
- It is difficult for software developer to point out errors from own creations.
- Many organizations have made a distinction between development and testing phase by making different people responsible for each phase.

Test manager

- manage and control a software test project
- supervise test engineers
- define and specify a test plan

- Software Test Engineers and Testers

- define test cases, write test specifications, run tests

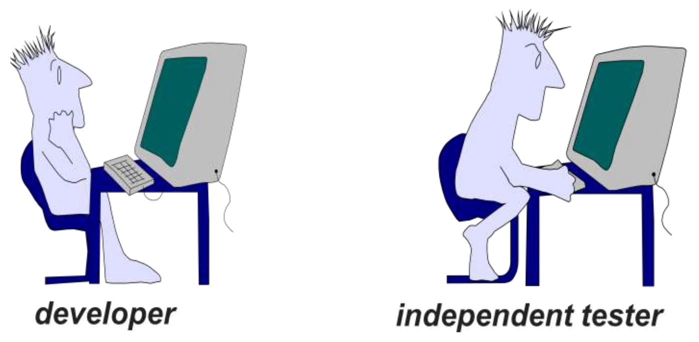
- Independent Test Group

- Development Engineers

- Only perform unit tests and integration tests

- Quality Assurance Group and Engineers

- Perform system testing
- Define software testing standards and quality control

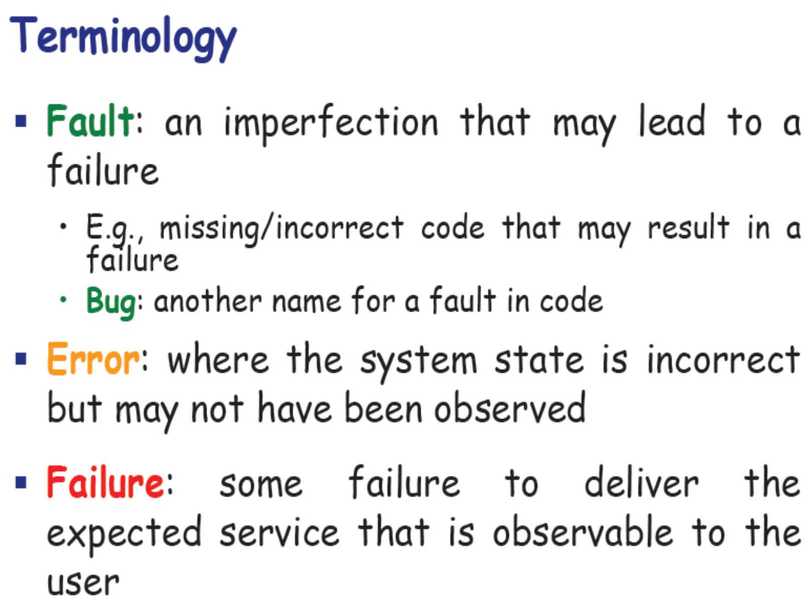


Understands the system but, will test "gently" and, is driven by "delivery"

Must learn about the system, but, will attempt to break it and, is driven by quality

What should we Test?

We should test the program’s responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are 28x28. If only one second it required executing one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.



Terminology

- **Fault**: an imperfection that may lead to a failure
  - E.g., missing/incorrect code that may result in a failure
  - **Bug**: another name for a fault in code
- **Error**: where the system state is incorrect but may not have been observed
- **Failure**: some failure to deliver the expected service that is observable to the user

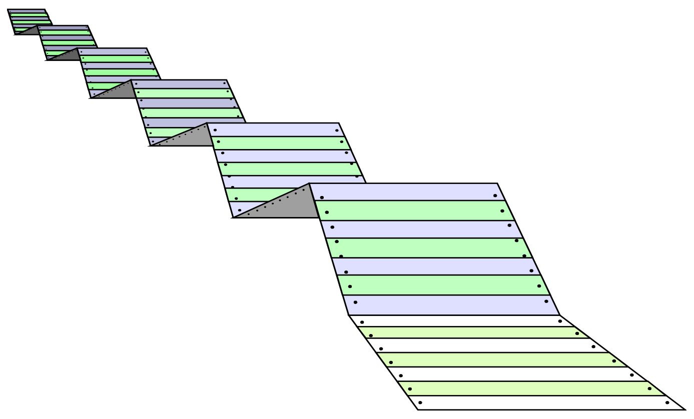
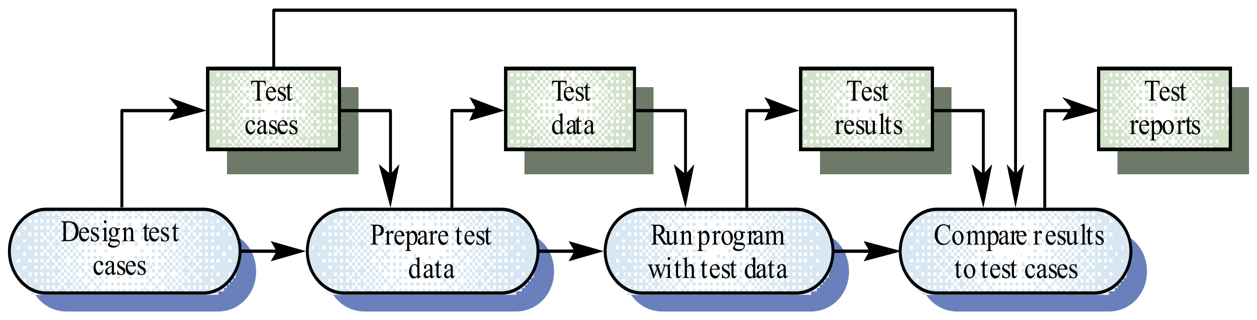


Fig: What a testing shows

6.2 Software testing Process



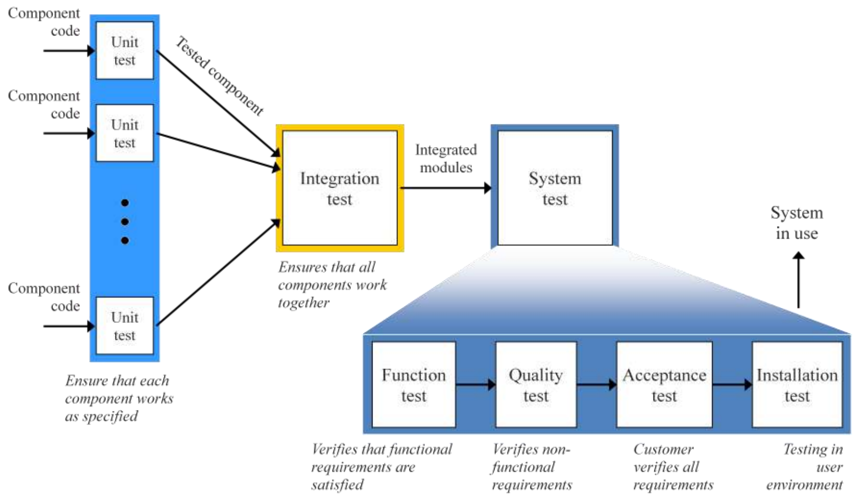
6.3 Principal of Testing

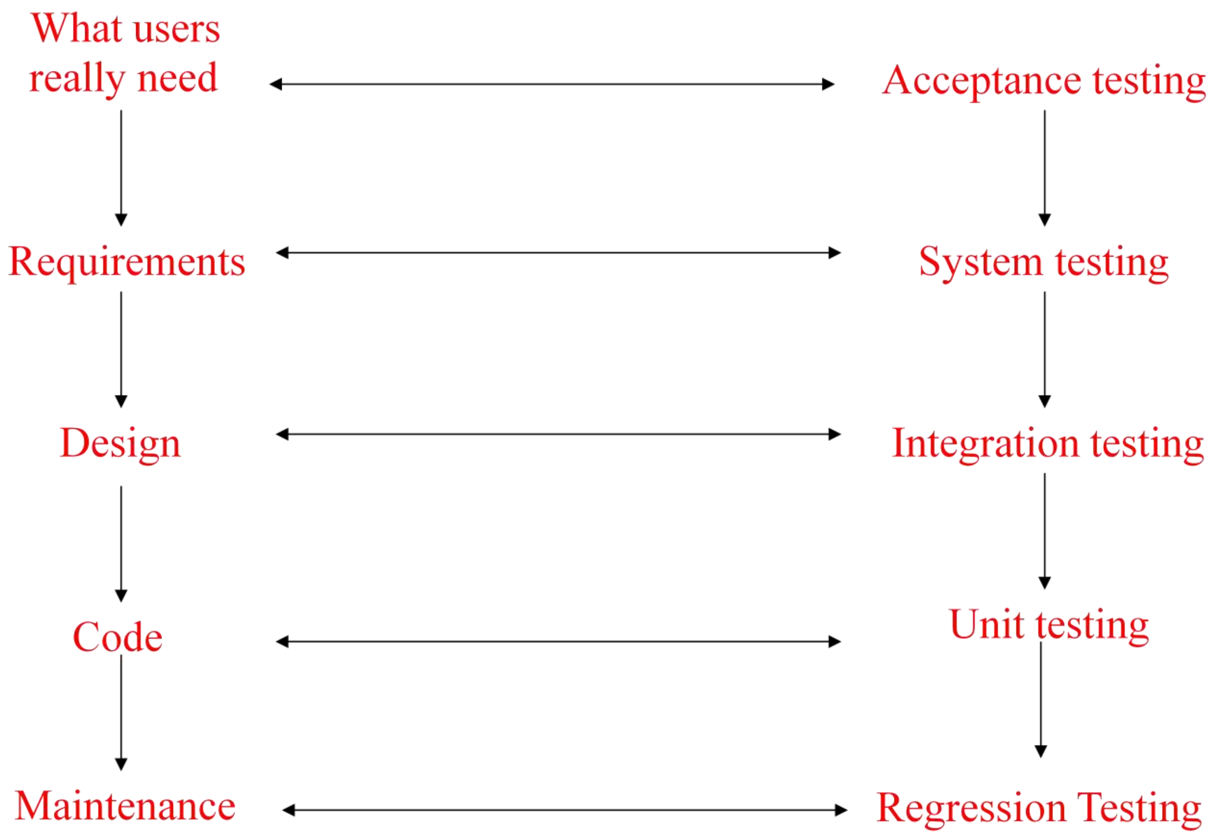
The goal of testing is to find errors, not to show that the program is errorless.

Parts of programs where a lot of errors have already been found are a good place to look for more errors.

The goal is not to humiliate the programmer!

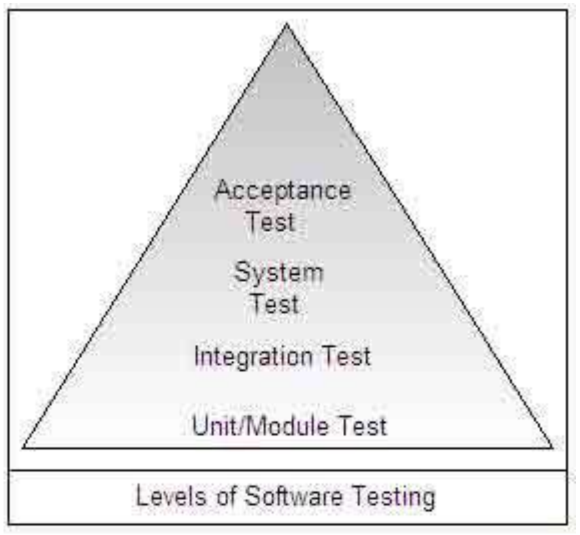
6.4 Test Case design


Component code
Unit test
Component code
Unit test
Component code
Unit test
Ensure that each component works as specified
Tested component
Integration test
Ensures that all components work together
Integrated modules
System test
System in use
Function test
Verifies that functional requirements are satisfied
Quality test
Verifies non-functional requirements
Acceptance test
Customer verifies all requirements
Installation test
Testing in user environment

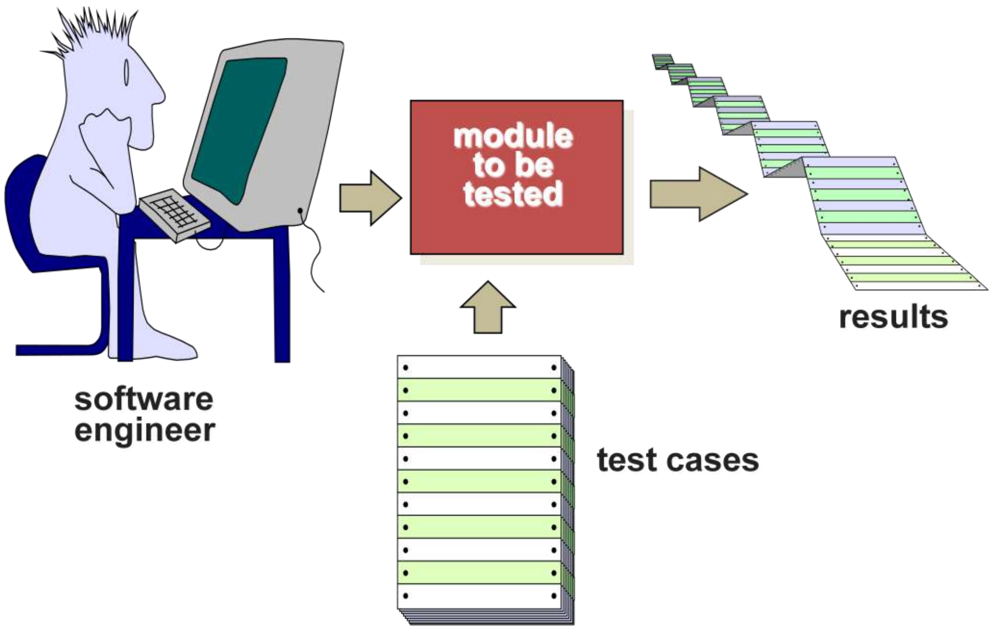


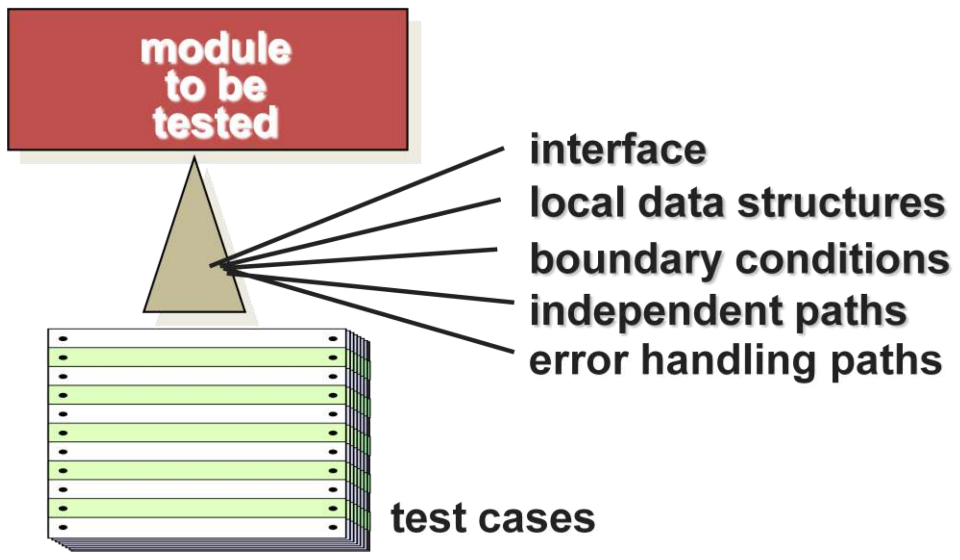
Label of software testing

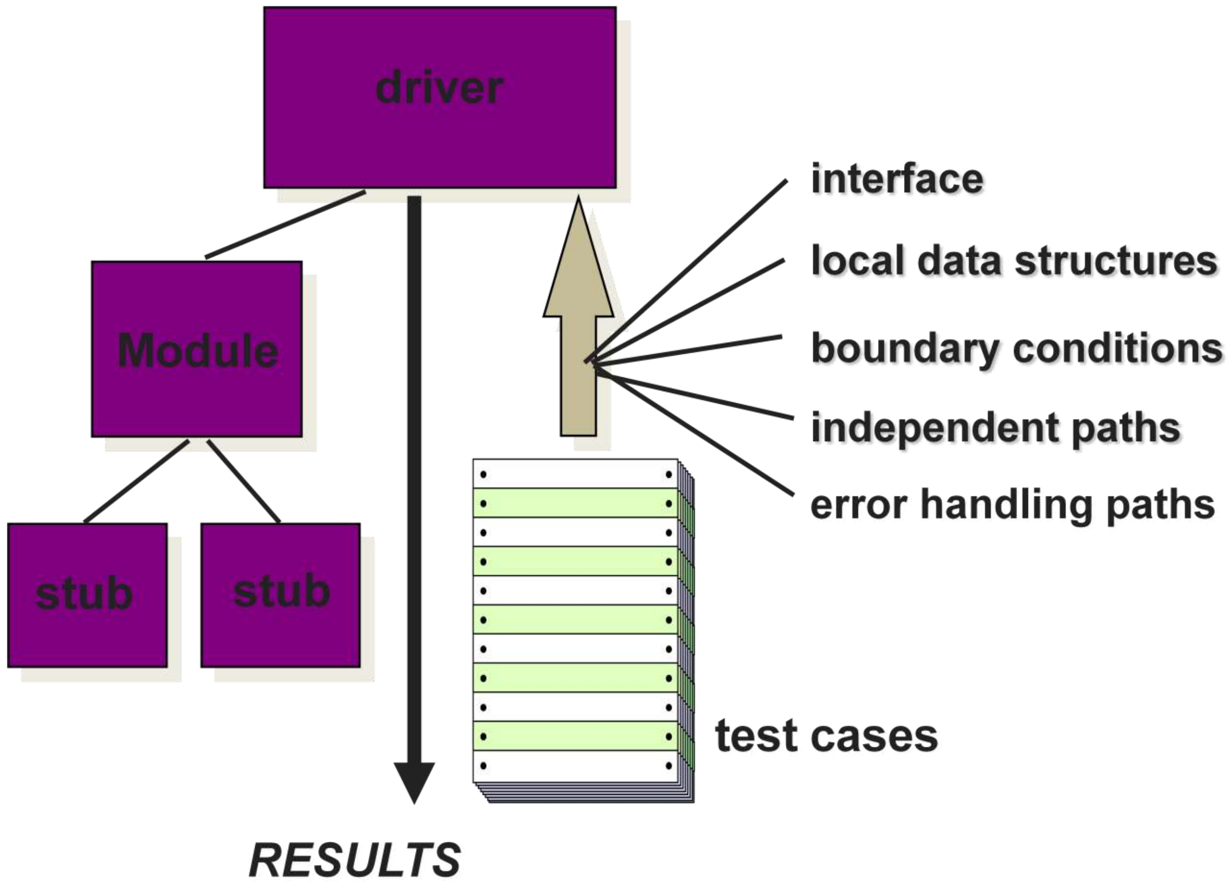
1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing



Unit Testing







- The most 'micro' scale of testing.

- Tests done on particular functions or code modules.
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).

6.5 Black-Box Testing(Boundary Value Analysis, Equivalence class Partitioning)

Black Box Testing

The technique of testing without having any knowledge of the interior workings of the application is Black Box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, when performing a black box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.

Advantages

- Well suited and efficient for large code segments.
- Code Access not required.
- Clearly separates user's perspective from the developer's perspective through visibly defined roles.
- Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language or operating systems.

Disadvantages

- Limited Coverage since only a selected number of test scenarios are actually performed.
- Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
- Blind Coverage, since the tester cannot target specific code segments or error prone areas.
- The test cases are difficult to design.

6.6 White-Box testing(Statement Coverage, Path coverage, Cyclomatic complexity)

White box testing is the detailed investigation of internal logic and structure of the code. White box testing is also called glass testing or open box testing. In order to perform white box testing on an application, the tester needs to possess knowledge of the internal working of the code.

The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

Advantages

- As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.
- It helps in optimizing the code.
- Extra lines of code can be removed which can bring in hidden defects.
- Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.

Disadvantages

- Due to the fact that a skilled tester is needed to perform white box testing, the costs are increased.
- Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems as many paths will go untested.
- It is difficult to maintain white box testing as the use of specialized tools like code analyzers and debugging tools are required.

Criteria	**Black Box Testing**	**White Box Testing**
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
Levels Applicable To	Mainly applicable to higher levels of testing: Acceptance Testing System Testing	Mainly applicable to lower levels of testing: Unit Testing Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design

6.7 Software Verification and Validation.

V & V

- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

– *Verification:* "Are we building the product right?"

– *Validation:* "Are we building the right product?"

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase,
- Whereas validation is the process of determining whether a fully developed system conforms to its requirements specification.

Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free

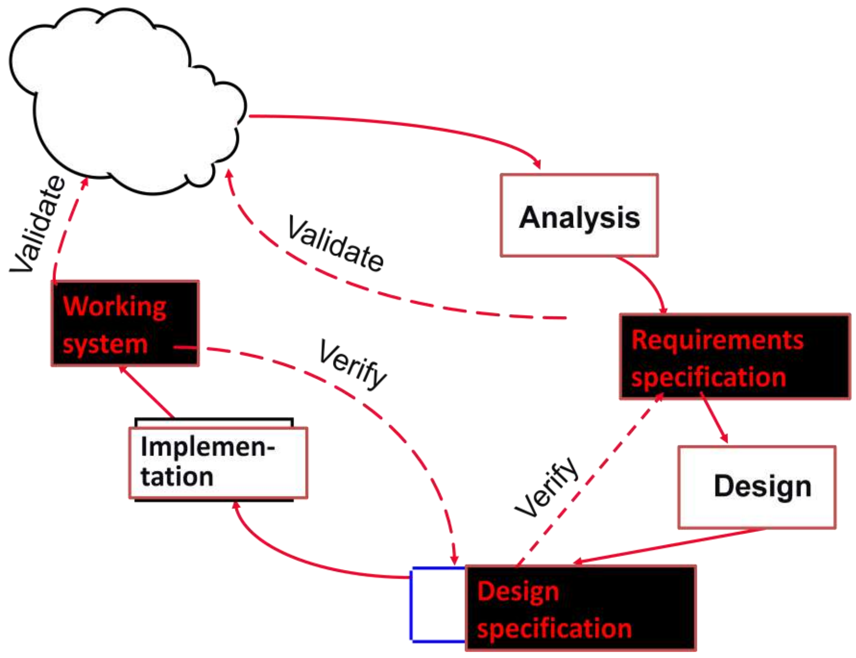


Fig Stages, Deliverables and V&V Activities

V & V Goals

- Verification and validation should establish confidence that the software is fit for its purpose.

– This does NOT mean completely free of defects.
– Rather, it must be good enough for its intended use. The type of use will determine the degree of confidence that is needed.

Verification & Validation

These two terms are very confusing for people, who use them interchangeably. Let's discuss about them briefly.

S.N.	Verification	Validation
1	Are you building it right?	Are you building the right thing?
2	Ensure that the software system meets all the functionality.	Ensure that functionalities meet the intended behavior.
3	Verification takes place first and includes the checking for documentation, code etc.	Validation occurs after verification and mainly involves the checking of the overall product.
4	Done by developers.	Done by Testers.
5	Have static activities as it includes the reviews, walkthroughs, and inspections to verify that software is correct or not.	Have dynamic activities as it includes executing the software against the requirements.
6	It is an objective process and no subjective decision should be needed to verify the Software.	It is a subjective process and involves subjective decisions on how well the Software works.

Alpha Test	Beta Test
What they do	
Improve the quality of the product and ensure beta readiness.	Improve the quality of the product, integrate customer input on the complete product, and ensure release readiness.

When they happen	
Toward the end of a development process when the product is in a near fully-usable state.	Just prior to launch, sometimes ending within weeks or even days of final release.
How long they last	
Usually very long and see many iterations. It’s not uncommon for alpha to last 3-5x the length of beta.	Usually only a few weeks (sometimes up to a couple of months) with few major iterations.
Who cares about it	
Almost exclusively quality/engineering (bugs, bugs, bugs).	Usually involves product marketing, support, docs, quality and engineering (basically the entire product team).
Who participates (tests)	
Normally performed by test engineers, employees, and sometimes “friends and family”. Focuses on testing that would emulate ~80% of the customers.	Tested in the “real world” with “real customers” and the feedback can cover every element of the product.
What testers should expect	
Plenty of bugs, crashes, missing docs and features.	Some bugs, fewer crashes, most docs, feature complete.
How they’re addressed	
Most known critical issues are fixed, some features may change or be added as a result of early feedback.	Much of the feedback collected is considered for and/or implemented in future versions of the product. Only important/critical changes are made.
What they achieve	
About methodology, efficiency and regiment. A good alpha test sets well-defined benchmarks and measures a product against those benchmarks.	About chaos, reality, and imagination. Beta tests explore the limits of a product by allowing customers to explore every element of the product in their native environments.
When it’s over	
You have a decent idea of how a product performs	You have a good idea of what your customer thinks

and whether it meets the design criteria (and if it's "beta-ready")	about the product and what s/he is likely to experience when they purchase it.
What happens next	
Beta Test!	Release Party!

CHAPTER 7

7.1 Software Measurement

- Metrics for software quality
- Software Quality Assurance
- Software reliability

The ISO 9000 quality standards.

Software quality assurance (SQA)

- Software Quality Assurance is an umbrella activity that is applied throughout the software process...
- The aim of s/w quality assurance process is to develop high quality s/w product
- SQA is the set of activities designed to evaluate the process by which s/w is developed or maintained.
- SQA is a planned and systematic pattern of all actions necessary to provide adequate confidence that the product conforms to establish technical requirements.
- The purpose of SQA is to provide assurance that the procedures, tools and techniques used during product development and modification are adequate to provide the desired level of confidence in the work products.
- **Software quality assurance (SQA) is a process that ensures that developed software meets and complies with defined or standardized quality specifications.**
- **SQA is an ongoing process within the software development life cycle (SDLC) that routinely checks the developed software to ensure it meets desired quality measures.**

What is quality?

Quality, simplistically, means that a product should meet its specification.

- Basic goal of software engineering is to produce quality software
- s/w quality is a goal and important field of s/w engineering.
- Addressed by several standardization bodies such as ISO, IEEE and ANSI
- s/w quality is conformance to explicit stated functional and performance requirements, explicitly documented development standards and implicit characteristic that are expected of all professionally developed s/w

- *Quality* refers to any measurable characteristics such as correctness, maintainability, portability, testability, usability, reliability, efficiency, integrity, reusability and interoperability.

The above definition emphasized on these points.

- s/w requirements are the foundation from which quality is measured. Lack of performance to requirement is lack of quality.
- Specified standard defines a set of development criteria that guide the manner in which s/w is engineered. If the criteria are not followed, lack of quality is surely in result.
- There is a set of implicit requirements that often goes unmentional. If s/w conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Software quality types:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Hierarchical Quality Model

This is problematical for software systems

- There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
- Some quality requirements are difficult to specify in an unambiguous way;
- Software specifications are usually incomplete and often inconsistent.

Software Qualities

- Correctness
- Reliability

- Robustness
- Performance
- User friendliness
- Verifiability
- Maintainability
- Repairability
- Safety
- Evolvability
- Reusability
- Portability
- Understandability
- Interoperability
- Productivity
- Size
- Timeliness
- Visibility

ISO 9000

An international set of standards for quality management.

Applicable to a range of organisations from manufacturing to service industries.

ISO 9001 applicable to organisations which design, develop and maintain products.

ISO 9001 is a generic model of the quality process that must be instantiated for each organisation using the standard.

ISO 9000 certification

Quality standards and procedures should be documented in an organisational quality manual. An external body may certify that an organisation's quality manual conforms to ISO 9000 standards. Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

Quality management activities

1. Quality assurance
    - Establish organisational procedures and standards for quality.
2. Quality planning
    - Select applicable procedures and standards for a particular project and modify these as required.
3. Quality control
    - Ensure that procedures and standards are followed by the software development team.
4. Quality management should be separate from project management to ensure independence.

http://ecomputernotes.com/software-engineering