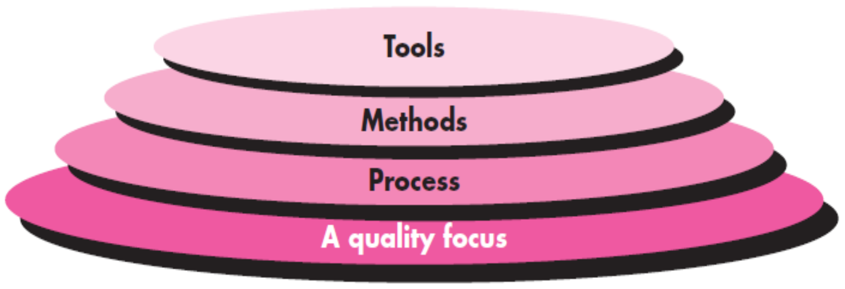
SEN Important question

Chapter 1

1) Describe the layered technology approach of Software Engineering.



A quality focus:

- Software engineering is a layered technology. Referring to the figure, any engineering approach (including software engineering) must rest on an organizational commitment to quality. The bedrock that supports software engineering is a quality focus.

Process:

- The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology.
- The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Methods:

- Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Tools:

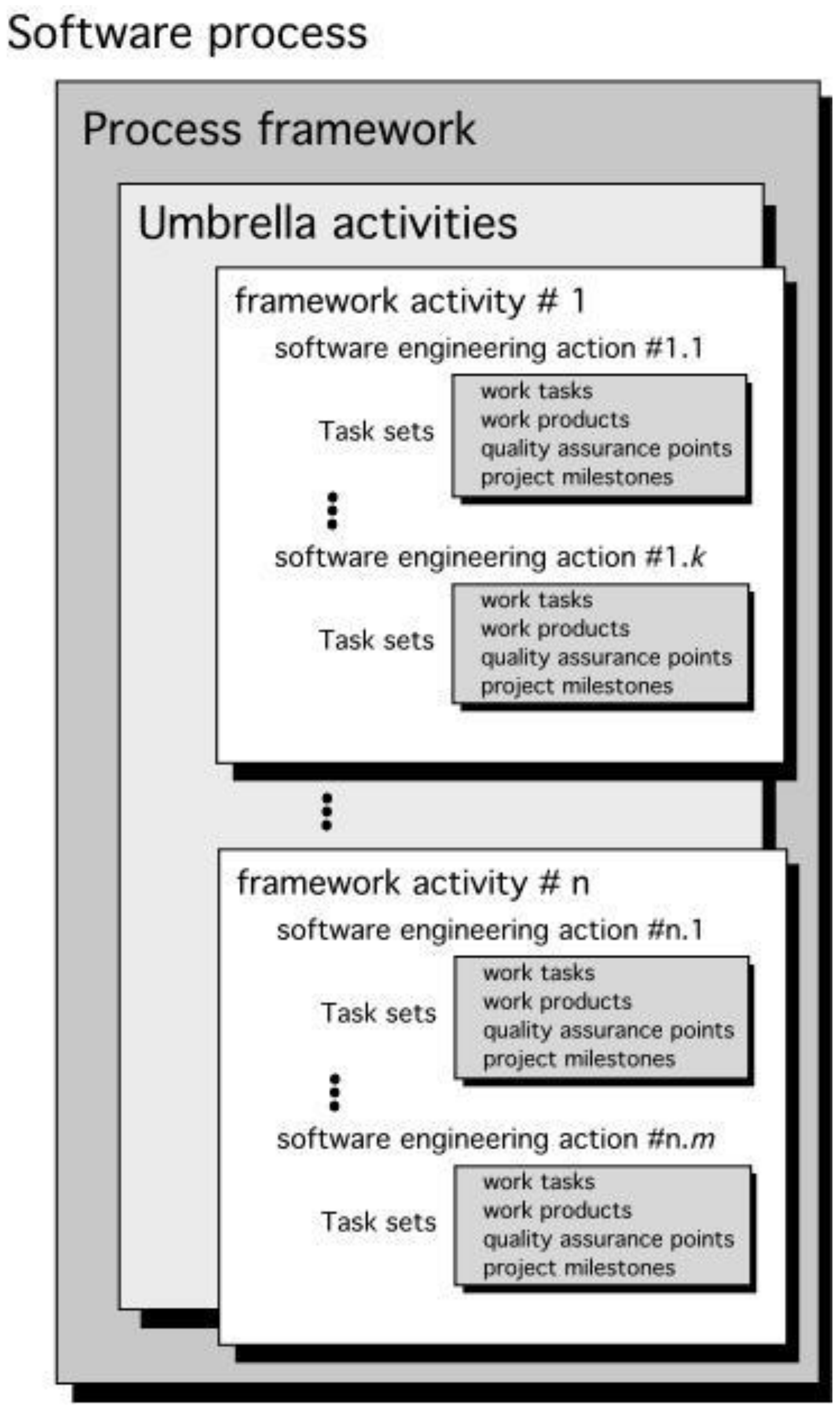
- Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

2) **Explain the basic process framework activities**

- A process framework establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- In fig each framework activity is populated by a set of software engineering actions. A collection of related tasks that produces a major software engineering work product.

Each action in process framework is populated with individual work tasks that accomplish some part of the work implied by the action.

1. **Communication:**
   Communication framework activity involves heavy communication and collaboration with the customer, encompasses requirements gathering, data gathering and other related activities.
2. **Planning:**
   Planning activity establishes a plan for software engineering work that follows. Planning describes the technical tasks to be conducted, the resources that will be required, schedule, and the risks that are likely in the work products to be produced.
3. **Modeling:**
   Modeling activity encompasses the creation of models that allow the developer and the customer to better understand software requirements specifications and the design that will achieve those requirements.
   There are two types of modeling i.e. analysis modeling and design modeling.
4. **Construction:**
   Construction activity combines code generation and the testing.
5. **Deployment:**
   The software is delivered to the customers who evaluates the delivered product and provides feedback based on the evaluation.



Software process
Process framework
Umbrella activities
framework activity # 1
software engineering action #1.1
Task sets
work tasks
work products
quality assurance points
project milestones
software engineering action #1.k
Task sets
work tasks
work products
quality assurance points
project milestones
framework activity # n
software engineering action #n.1
Task sets
work tasks
work products
quality assurance points
project milestones
software engineering action #n.m
Task sets
work tasks
work products
quality assurance points
project milestones

3) Explain waterfall model

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

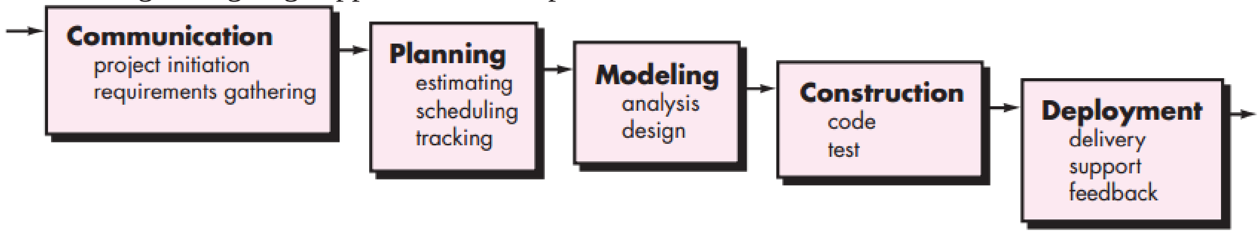


Fig: Waterfall Model

Situation in which waterfall model is applicable:

- There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made.
- Framework activities involved:

➢ **Communication:**
It involves heavy communication with the customer (or stakeholder) and encompasses requirement gathering and related activities.

➢ **Planning:**
In this activity, effort required, cost/budget, risk analysis ,time duration re estimated (project plan is made)

➢ **Modeling:**
This activity creates analysis and design models that both developers and customer to better understand the requirements. Data structure, software architecture and other details are made.

➢ **Construction:**
This activity performs code generation and testing to ensure whether requirements are fulfilled.
Code generation is done first and then testing is done after that.

➢ **Deployment:**
Once the product is fully developed, it is delivered to the customer. Customer evaluates the product and provides feedback.

Advantages:

- It is the simplest software process model.
- Easy to understand
- Phases are completed one at a time
- Works well for smaller projects

Disadvantages:

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

- The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

4) Explain Spiral Model

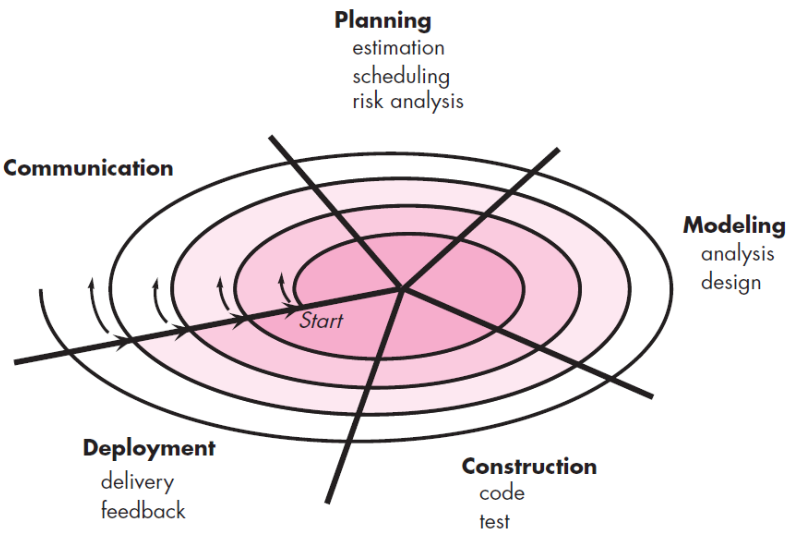


Fig: Spiral Model

- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.
- A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represent one segment of the spiral path illustrated in Figure. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.
- Risk is considered as each revolution is made.
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan.
- Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

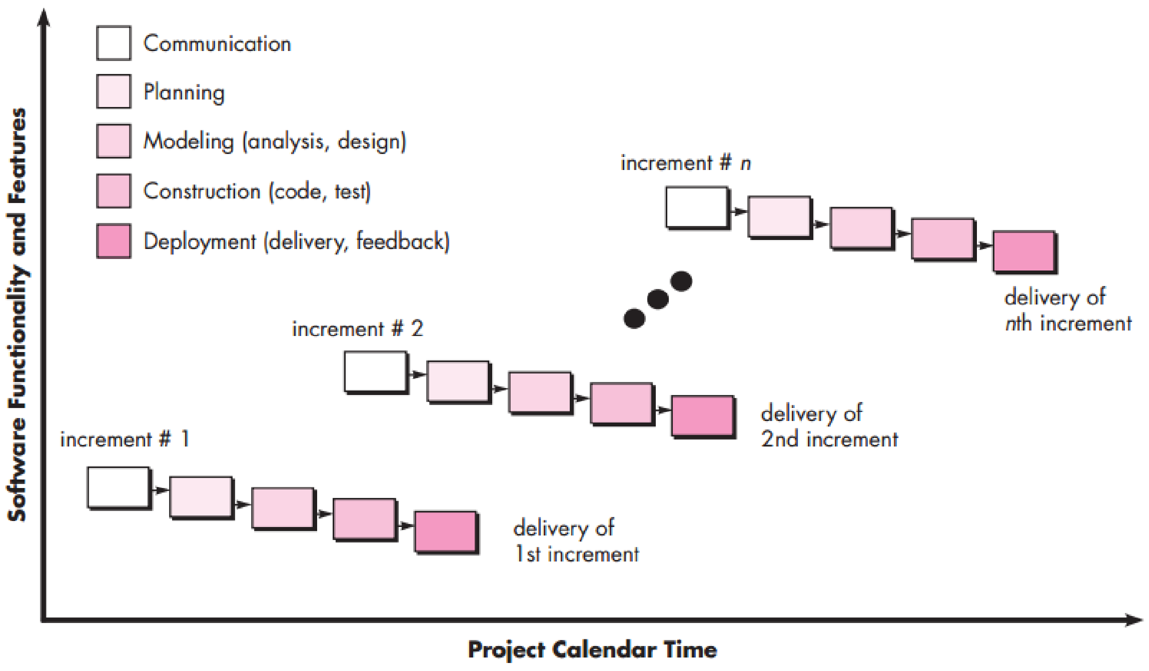
Advantages:

- The spiral model is a realistic approach to the development of large-scale systems and software.
- High amount of risk analysis hence, avoidance of risk is enhanced.

Disadvantages:

- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
- It demands considerable risk assessment expertise and relies on this expertise for success.
- If a major risk is not uncovered and managed, problems will undoubtedly occur.

5) Explain incremental Model



Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

Fig: Incremental Model

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

Situation in which incremental model is applicable:

- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process.
- In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered.
- The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment.

- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Example:

Word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

Advantages:

- Useful when staffing is unavailable
- Less costly to change the scope of the project
- Customer can respond to each build.

Disadvantages:

- Cost is higher than waterfall model.
- Needs good planning and design

6) State any four attributes of good software.

1. Functionality:
It refers to the degree of performance of the software against its intended purpose.

2. Reliability:
It refers to the ability of the software to provide desired functionality under the given conditions.

3. Usability:
It refers to the extent to which the software can be used with ease and simple.

4. Maintainability:
Software must evolve to meet changing needs.

5. Dependability:
Software must be trustworthy.

6. Efficiency:
Software should not make wasteful of system resources.

7. Acceptability
Software must be accepted by the users for which it was designed.

8. Portability:
It refers to the ease with which software developers can transfer software from one platform to another, without changes.

9. Integrity:
It refers to the degree to which unauthorized access to the software can be prevented.

10. **Robustness:**
    It refers to the degree to which the software can keep on functioning in spite of being provided with invalid data

7) PSP and TSP Framework Activities

PSP Framework Activities includes: -
The PSP model defines five framework activities:
Planning: This activity isolates requirements and develops both size and resource estimates. In
addition, a defect estimates (the number of defects projected for the work) is made. All metrics
are recorded on worksheets or templates. Finally, development tasks are identified and a project
schedule is created.
High-level design: External specifications for each component to be constructed are developed
and a component design is created. Prototypes are built when uncertainty exists. All issues are
recorded and tracked.
High-level design review: Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
Development: The component-level design is refined and reviewed. Code is generated, reviewed,
compiled, and tested. Metrics are maintained for all important tasks and work results.
Postmortem: Using the measures and metrics collected (this is a substantial amount of data that
should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

TSP Framework Activities includes:-
TSP defines the following framework activities: **project launch, high-level design, implementation, integration and test,** and **postmortem.**
Project Launch: It reviews core objective and describes the TSP structure and content. It assigns
terms and roles to developers and describes the customer needs statement. It also establishes team
and individual goals.
High Level Design: It creates high-level design, specifies the design, and inspects the design develop an integration test plan.
Implementation: Implementation uses the TSP to implement modules/unit, creates a detailed design of modules/units, reviews the design, translates the design to code, review the code, compile and test the modules/units and analyze the quality of the modules/units.
Integration and Test: Testing builds and integrates these builds into a system. It conducts a system test and produce user documentation.
Postmorterm: It conducts a postmortem analysis, writes a cycle report and produce peer and team evaluations.

8) Explain RAD Model

- The RAD model approach is applicable, if the business application requirements are modularized as function to be completed by individual teams and finally to integrate into a complete system.
- As such compared to waterfall model the team will be of larger size to function with proper coordination.
- Rapid application Development (RAD) is a modern software process model that emphasizes a short development cycle. The RAD Model is a —high-speed- adaptation of the waterfall model, in which rapid development is achieved by using a component based construction approach.
- If requirements are well understood and project scope is considered, the RAD process enables a development team to create a Fully Functional System within a very short period of time (e.g. 60 to 90 days). One of the distinct features of RAD model is the possibility of cross life cycle activities which will be assigned to teams, teams #1 to team #n leading to each module getting developed almost simultaneously.

Advantages:

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

Limitations:

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

9) Types of software

System software

• It is collection of programs written to service other programs. Some system software (e.g., compilers,
editors, and file management utilities) processes complex, but determinate, information structures.

• Other systems applications (e.g., operating system components, drivers, networking software,
telecommunications processors) process largely indeterminate data.

Application software

• Stand-alone programs that solve a specific business need.

• Applications in this area process business or technical data in a way that facilitates business operations
or management/technical decision making

Engineering/scientific software
• Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle
orbital dynamics, and from molecular biology to automated manufacturing
• E.g.: CAD software.
Embedded software
• Resides within a product or system and is used to implement and control features and functions for the
end user and for the system itself.
• Embedded software can perform limited functions (e.g., key pad control for a microwave oven) or
provide significant function and control capability (e.g., digital functions in an automobile such as fuel
control, dashboard displays, and braking systems).
• E.g. Control buttons of washing machine.
Product-line software
• Designed to provide a specific capability for use by many different customers.
• Product-line software can focus on a limited marketplace (e.g., inventory control products) or address
mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).
Web applications
• Called "WebApps," this network-centric software category spans a wide array of applications.
• In their simplest form, WebApps can be little more than a set of linked hypertext files that present
information using text and limited graphics.
Artificial intelligence software
• Makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.
• Applications within this area include robotics, expert systems, pattern recognition (image and voice),
artificial neural networks, theorem proving, and game playing

10) Failure curve for software

Definition of Software

Software is:
1. **Instructions** (computer programs) that when executed provide desired features, function, and performance;
2. **Data structures** that enable the programs to adequately manipulate information, and
3. **Descriptive information (documents)** in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics of software

Software is developed or engineered; it is not manufactured in the classical sense.
The two activities (software development and hardware manufacturing) are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems.

Software doesn't "wear out." But it does deteriorate!
Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure
This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its 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 shown in the "actual curve" .Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

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

Definition of Software

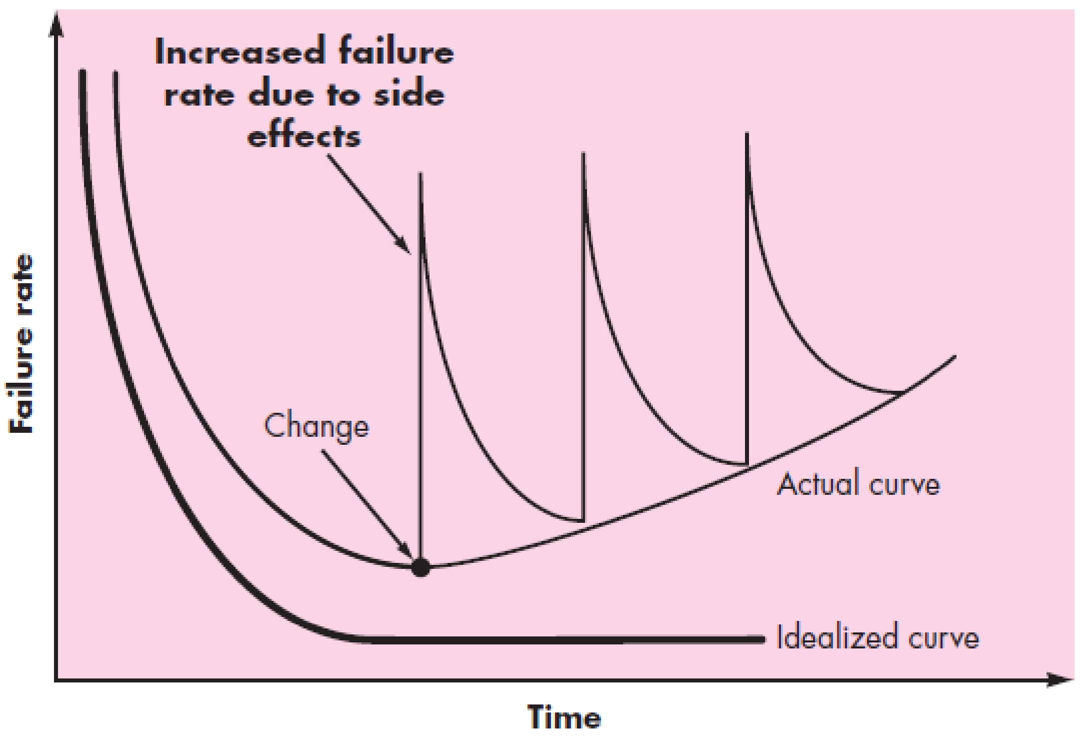
Software is:
1. **Instructions** (computer programs) that when executed provide desired features, function, and performance;
2. **Data structures** that enable the programs to adequately manipulate information, and
3. **Descriptive information (documents)** in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics of software

Software is developed or engineered; it is not manufactured in the classical sense.
The two activities (software development and hardware manufacturing) are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems.

Software doesn't "wear out." But it does deteriorate!

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory,
therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure
This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its
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 shown in the "actual curve" .Before the curve can return to the original
steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the
minimum failure rate level begins to rise—the software is deteriorating due to change.
**Although the industry is moving toward component-based construction, most software continues to
be custom built.**



Chapter 2

1) Explain the core principles of software engineering in details

The First Principle: The Reason It All Exists

- A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind.
- Before specifying a system requirement, system functionality, before determining the hardware platforms, first determine, whether it adds value to the system.

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

- All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system.
- It doesn't mean that features should be discarded in the name of simplicity.
- Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify.

The Third Principle: Maintain the Vision

- A clear vision is essential to the success of a software project.
- If you make compromise in the architectural vision of a software system, it will weaken and will eventually break even the well-designed systems.
- Having a powerful architect who can hold the vision helps to ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume

- Always specify, design, and implement by keeping in mind that someone else will have to understand what you are doing.
- The audience for any product of software development is potentially large.
- Design (make design), keeping the implementers (programmers) in mind. Code (program) with concern for those who will maintain and extend the system.
- Someone may have to debug the code you write, and that makes them a user of your code.

The Fifth Principle: Be Open to the Future

- A system with a long lifetime has more value.
- True "industrial-strength" software systems must last for longer.
- To do this successfully, these systems must be ready to adapt changes.
- Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem.

The Sixth Principle: Plan Ahead for Reuse

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

The Seventh principle: Think!

- Placing clear, complete thought before action almost always produces better results.
- When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again.
- If you do think about something and still do it wrong, it becomes a valuable experience.

2) Communication Principle

Principle 1 Listen:

- Try to focus on the speaker‘s words, rather than formulating your response to those words.
- Ask for clarification if something is unclear, but avoid constant interruptions.
- Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2 Prepare before you communicate:

- Spend the time to understand the problem before you meet with others. If necessary, perform some research to understand business domain.
- If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3 someone should facilitate the activity:

- Every communication meeting should have a leader (a facilitator)

(1) To keep the conversation moving in a productive direction,
(2) To mediate any conflict that does occur, and
(3) To ensure that other principles are followed.

Principle 4 Face-to-face communication is best:

- It usually works better when some other representation of the relevant information is present.
- For example, a participant may create a drawing /document that serve as a focus for discussion.

Principle 5 Take notes and document decisions:

- Someone participating in the communication should serve as a recorder and write down all important points and decisions.

Principle 6 Strive for collaboration:

- Collaboration occurs when the collective knowledge of members of the team is used to describe product or system functions or features.
- Each small collaboration builds trust among team members and creates a common goal for the team.

Principle 7 Stay focused; modularize your discussion:

- The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
- The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.

Principle 8 if something is unclear, draw a picture:

- Verbal communication goes only so far.
- A sketch or drawing can often provide clarity when words fail to do the job.

Principle 9
(a) Once you agree to something, move on.
(b) If you can’t agree to something, move on.
(c) If a feature or function is unclear and cannot be clarified at the moment, move on.

The people who participate in communication should recognize that many topics require discussion and that moving on is sometimes the best way to achieve communication agility.

Principle 10 Negotiation is not a contest or a game: It works best when both parties win.

- There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates.
- If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

3) Construction Principle

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

Coding Principles

In modern software engineering work, coding may be

(1) The direct creation of programming language source code (e.g., Java),

(2) The automatic generation of source code using an intermediate design-like representation of the component to be built (e.g. Microsoft front end where code is automatically generated), or

(3) The automatic generation of executable code using a "fourth-generation programming language" (e.g., Visual C++).

- **Preparation principles: Before you write one line of code, be sure you**
  - Understand of the problem you're trying to solve.
  - Understand basic design principles and concepts.
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
  - Select a programming environment that provides tools that will make your work easier. (E.g. TC, JRE etc.).
  - Create a set of unit tests that will be applied once the component you code is completed.

- **Programming principles: As you begin writing code, be sure you**
  - Constrain your algorithms by following structured programming practice.
  - Consider the use of pair programming.
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards
  - Write code that is self-documenting. (e.g. Comments)
  - Create a visual layout (e.g., indentation and blank lines) that aids understanding.

- **Validation Principles: After you've completed your first coding pass, be sure you**
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code.

4) Deployment Principle

Principle 1: Customer expectations for the software must be managed.

- Software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2: A complete delivery package should be assembled and tested.

- A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users.
- All installation scripts and other operational features should be thoroughly exercised.

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

- When a problem or question arises at end user's side, he/she expects responsiveness and accurate information. If support is ad hoc or nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate recordkeeping mechanisms should be established so that the software team can conduct an assessment of the kinds of support requested.

Principle 4: Appropriate instructional materials must be provided to end users.

- The software team delivers more than the software itself.
- Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided

Principle 5: Buggy software should be fixed first, delivered later.

- Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." Customer gets disappointed by this.
- Hence it is necessary to fix the bug before the product is delivered to the customer.

5) Requirements engineering task

(i) Inception and
(ii) Elicitation

1. Inception

- Most projects begin when a business need is identified or a potential new market or service is discovered.
- Stakeholders from the business community (e.g., business managers, marketing people, and product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.
- At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the customer and the software team.

2. Elicitation

- Elicitation means to define what is required.
- Requirement engineer asks the customer, user and others:

1. what the objectives for the system or product are
2. what is to be accomplished
3. how the system or product fits into the needs of the business,
4. how the system or product is to be used on a day-to-day basis

Requirement elicitation is difficult because numbers of problems are encountered:

- **Problems of scope:**
  The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems 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 (unclear) or untestable.

- **Problems of volatility:**

The requirements change over time.

6) SRS and its importance

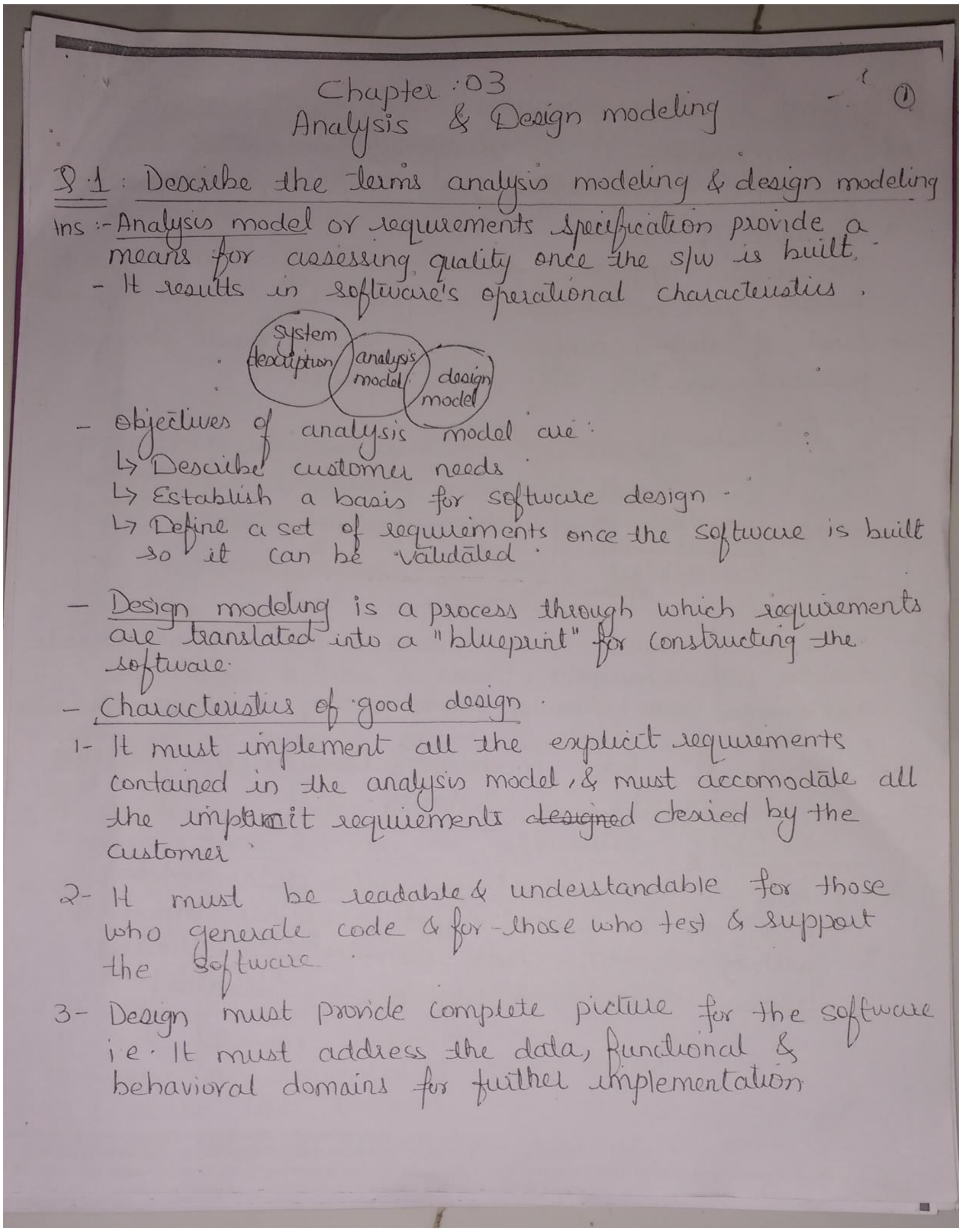
- A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built that must be specified before the project is to commence.
- It is a primary document for development of software.
- It is written by Business Analysts who interact with client and gather the requirements to build the software.

Need/Importance of SRS:

- **It establishes the basis for agreement between the customers and the suppliers on what the software product is to do.**
  The complete description of the functions performed by the software specified in the SRS will assist the users to determine if the software meets their needs.
- **Reduces the development effort.**
  The preparation of the SRS forces the concerned groups to consider all of the requirements before design begins and reduces later redesign, recoding, and retesting.
- **Provide a basis for estimating cost and schedules.**
  The description of the product to be developed given in SRS is a realistic basis for estimating project costs and prices.
- **Provide a baseline for verification and validation.**
  Organizations can develop their validation and verification plans much more productively from a good SRS.
- **Facilitate Transfer.**
  The SRS makes it easier to transfer the software product to new users or new machines.
- **Serve as basis for enhancement.**
  Because the SRS discusses the product but not just the project that is developed, the SRS serves as a basis for later enhancement of the finished product.

Chapter 3

1) Describe analysis and design modeling



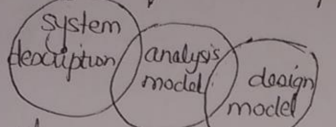
Chapter : 03

Analysis & Design modeling ①

Q.1 : Describe the terms analysis modeling & design modeling

Ans :- Analysis model or requirements specification provide a means for assessing quality once the s/w is built.

- It results in software's operational characteristics.



- objectives of analysis model are :
  - ↳ Describe customer needs.
  - ↳ Establish a basis for software design.
  - ↳ Define a set of requirements once the software is built so it can be validated.

- Design modeling is a process through which requirements are translated into a "blueprint" for constructing the software.

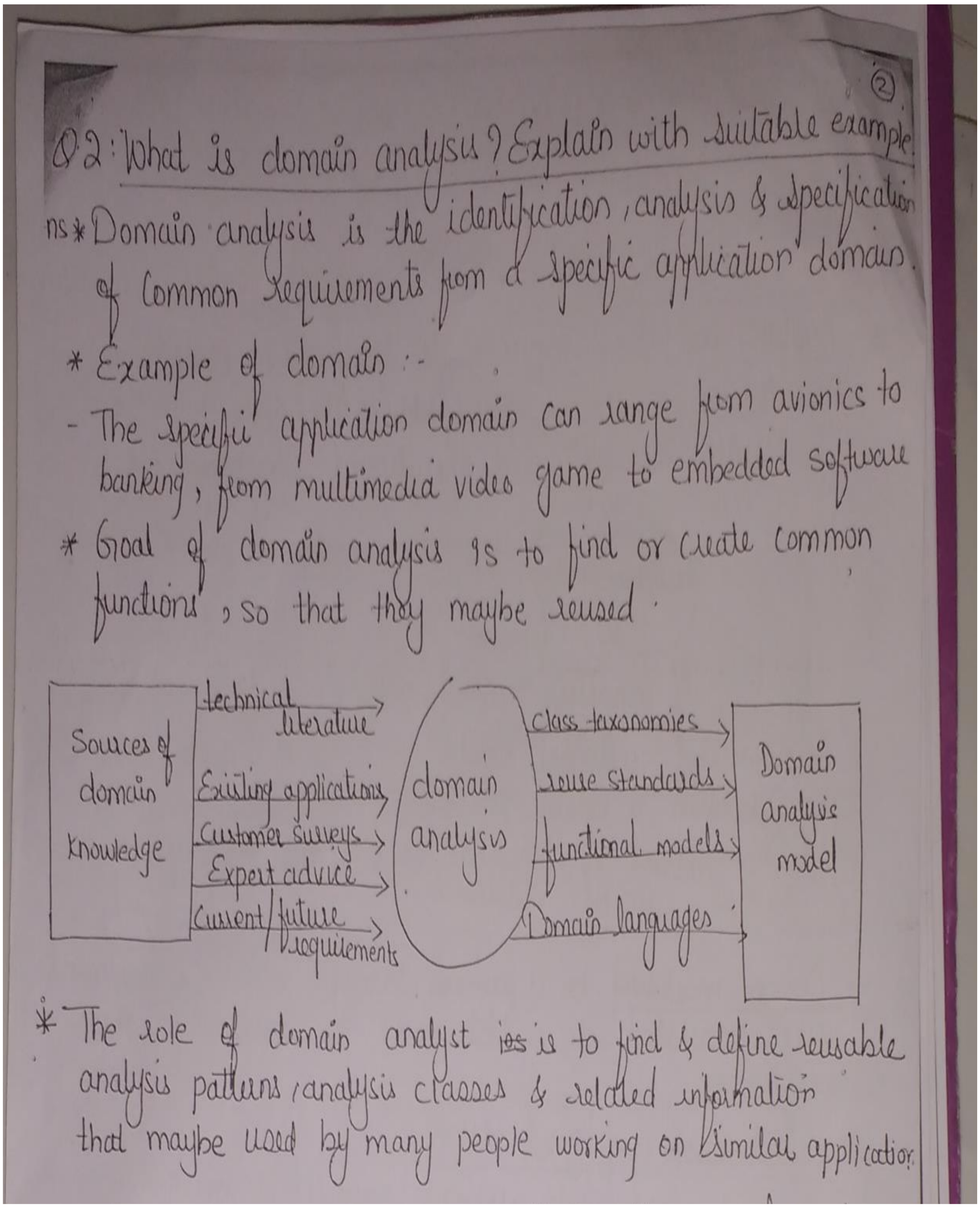
- Characteristics of good design.

1- It must implement all the explicit requirements contained in the analysis model, & must accomodate all the implicit requirements ~~designed~~ desired by the customer.

2- It must be readable & understandable for those who generate code & for those who test & support the software.

3- Design must provide complete picture for the software i.e. It must address the data, functional & behavioral domains for further implementation

2) What is domain analysis



②

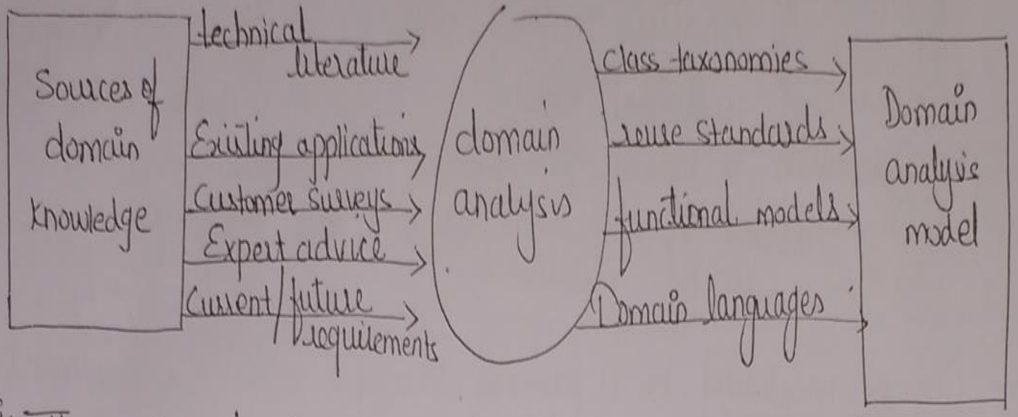
Q.2: What is domain analysis? Explain with suitable example.

ns * Domain analysis is the identification, analysis & specification of common requirements from a specific application domain.

* Example of domain :-

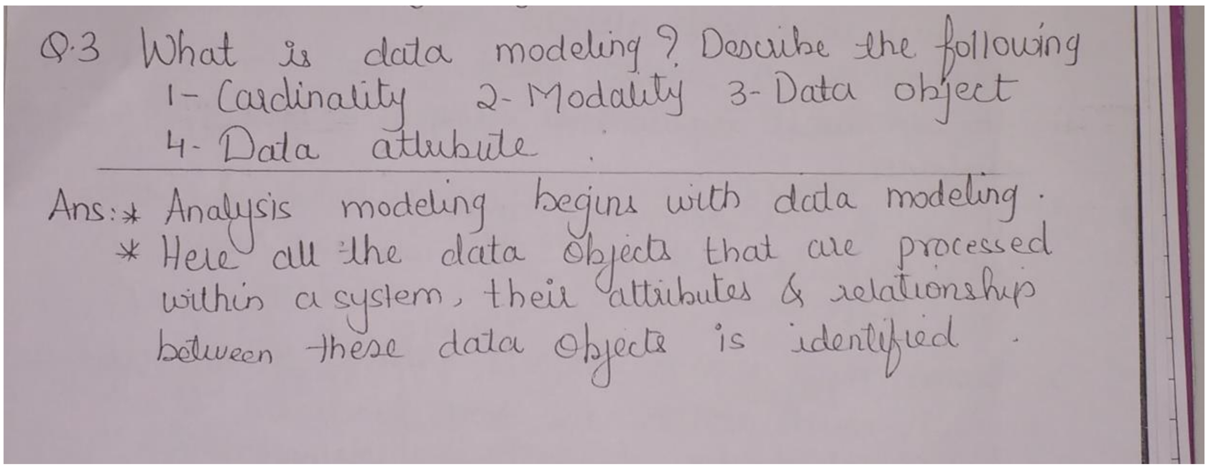
- The specific application domain can range from avionics to banking, from multimedia video game to embedded software

* Goal of domain analysis is to find or create common functions, so that they maybe reused.



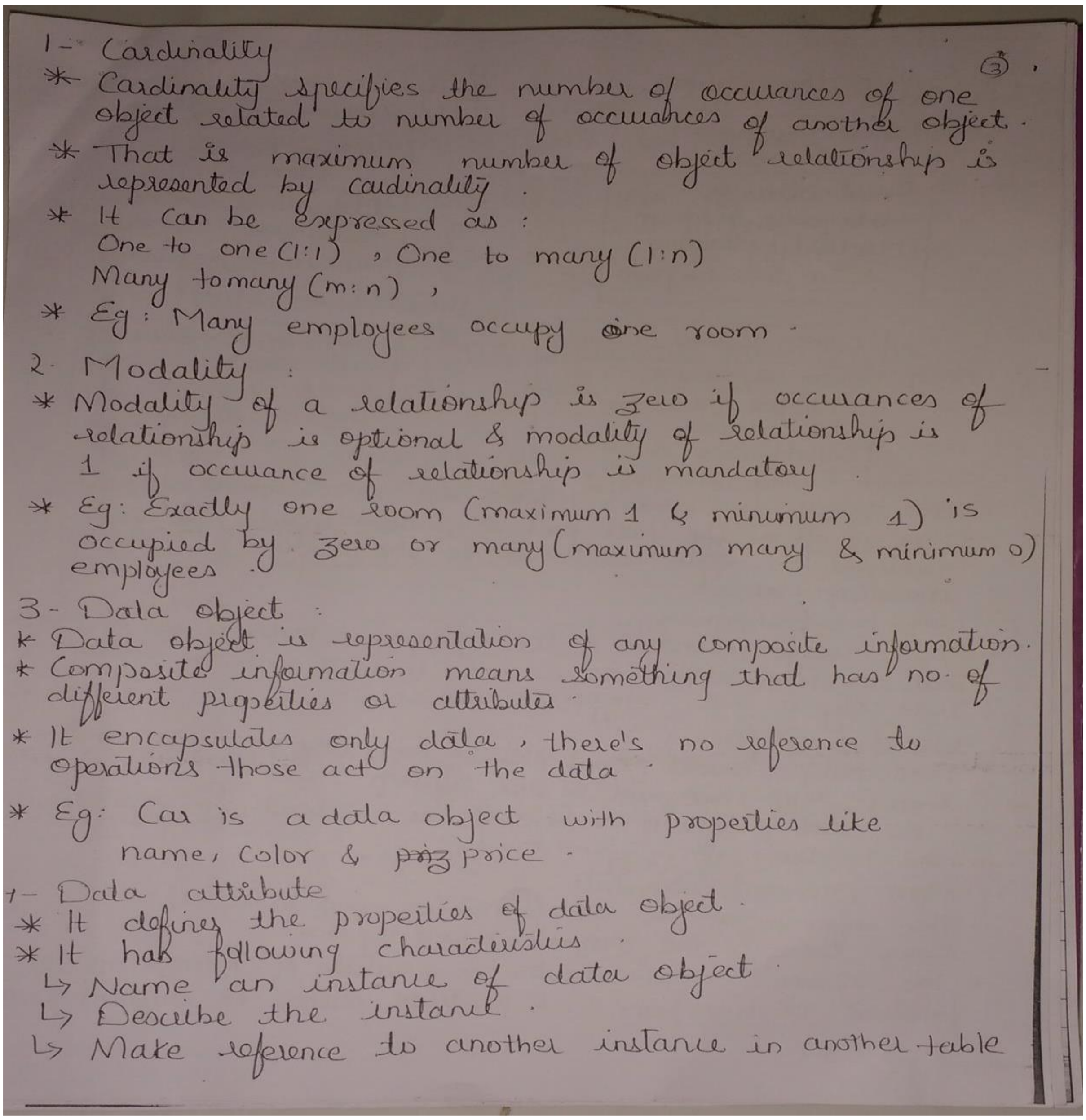
* The role of domain analyst ~~ios~~ is to find & define reusable analysis patterns, analysis classes & related information that maybe used by many people working on similar application.

3) What is data modeling? Explain cardinality and modality, object and attributes



Q.3 What is data modeling? Describe the following
1- Cardinality 2- Modality 3- Data object
4- Data attribute.

Ans: * Analysis modeling begins with data modeling.
* Here all the data objects that are processed within a system, their attributes & relationship between these data objects is identified.



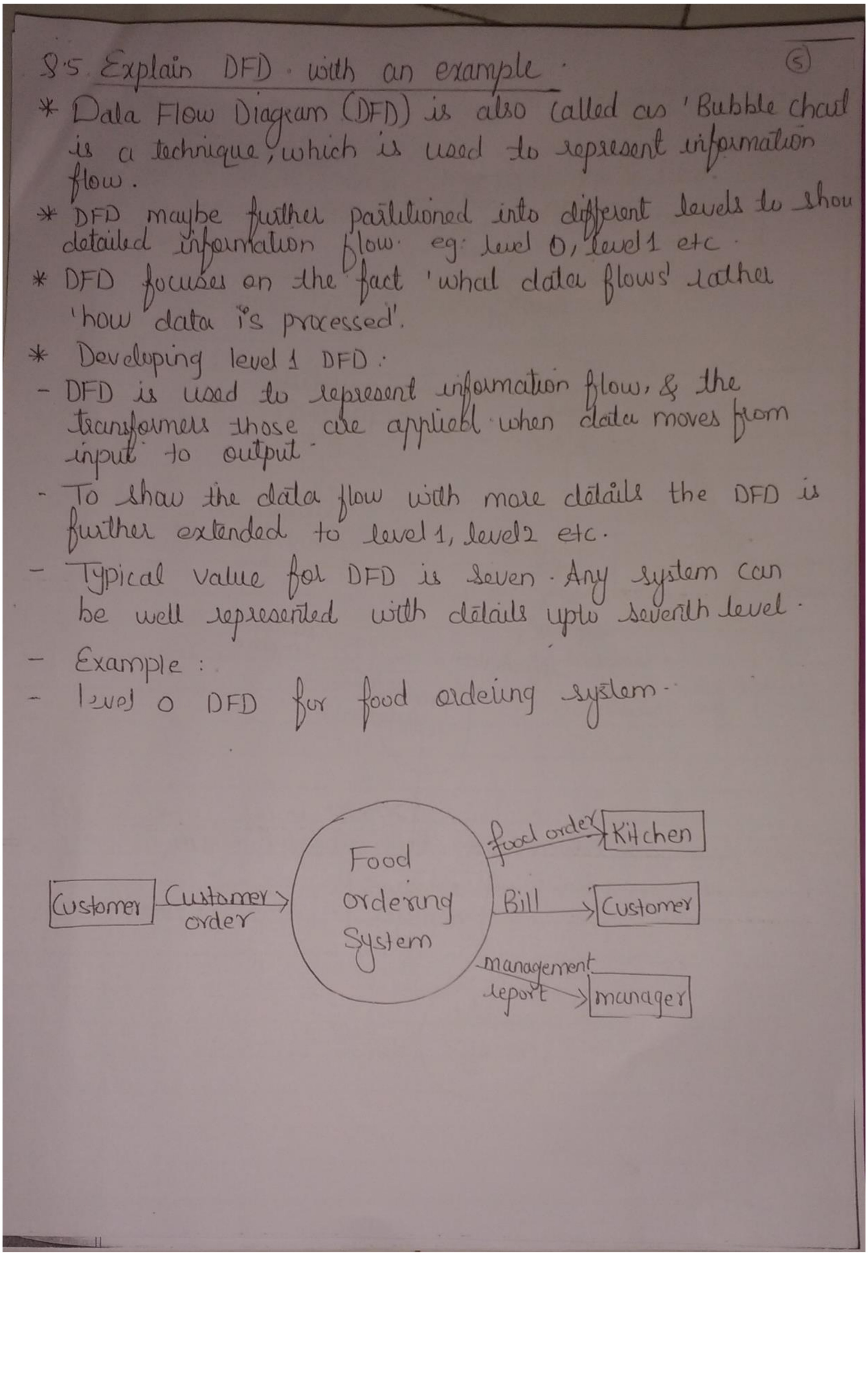
1- Cardinality (3)
* Cardinality specifies the number of occurances of one object related to number of occuarances of another object.
* That is maximum number of object relationship is represented by cardinality.
* It can be expressed as:
One to one (1:1), One to many (1:n)
Many to many (m:n),
* Eg: Many employees occupy one room.

2. Modality:
* Modality of a relationship is zero if occurances of relationship is optional & modality of relationship is 1 if occurance of relationship is mandatory.
* Eg: Exactly one room (maximum 1 & minimum 1) is occupied by zero or many (maximum many & minimum 0) employees.

3- Data object:
* Data object is representation of any composite information.
* Composite information means something that has no. of different properties or attributes.
* It encapsulates only data, there's no reference to operations those act on the data.
* Eg: Car is a data object with properties like name, Color & ~~prz~~ price.

4- Data attribute
* It defines the properties of data object.
* It has following characteristics.
↳ Name an instance of data object.
↳ Describe the instance.
↳ Make reference to another instance in another table

4) Explain level 0 and level 1 DFD with an example



Q.5 Explain DFD with an example. (5)

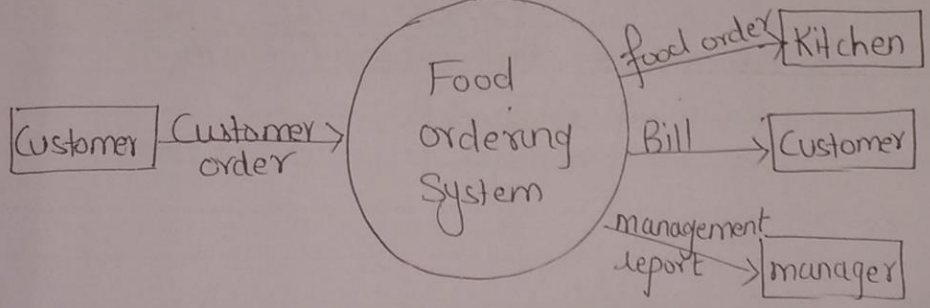
* Data Flow Diagram (DFD) is also called as 'Bubble chart' is a technique, which is used to represent information flow.

* DFD maybe further partitioned into different levels to show detailed information flow. eg: level 0, level 1 etc.

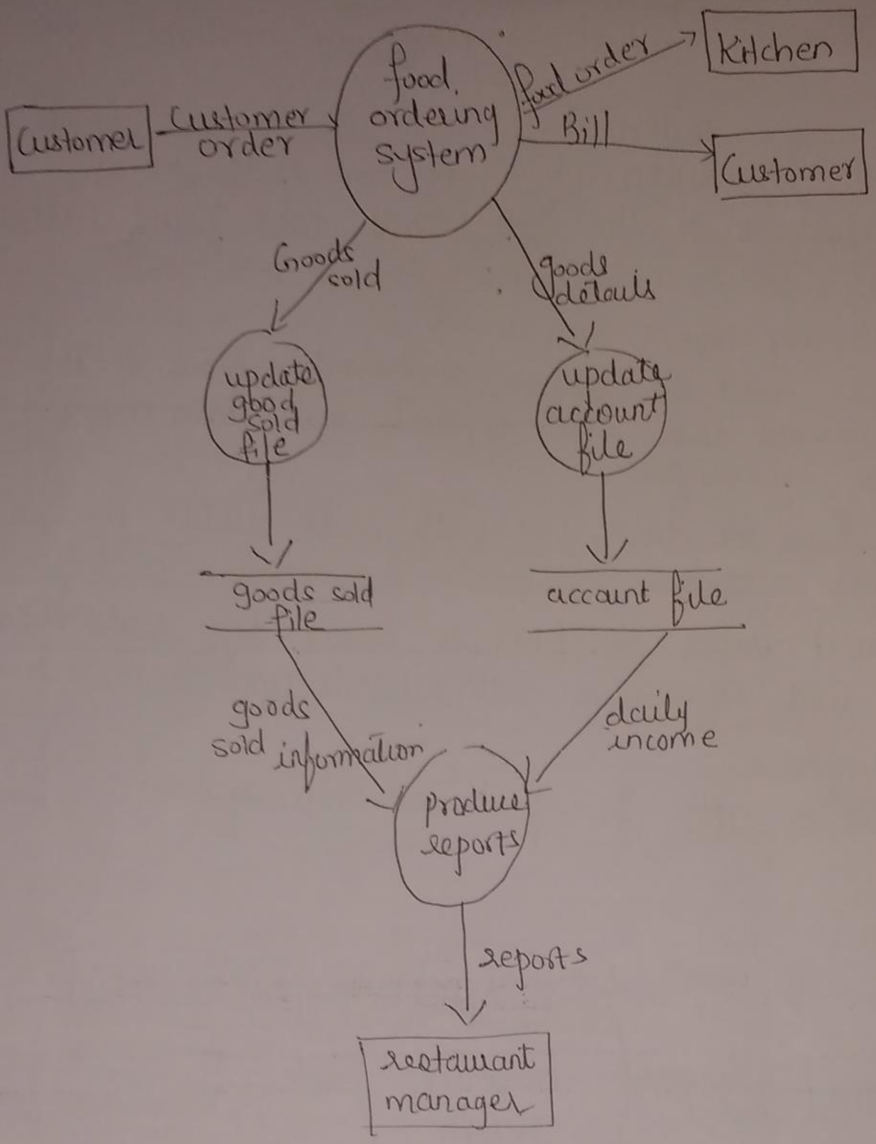
* DFD focuses on the fact 'what data flows' rather 'how data is processed'.

* Developing level 1 DFD :

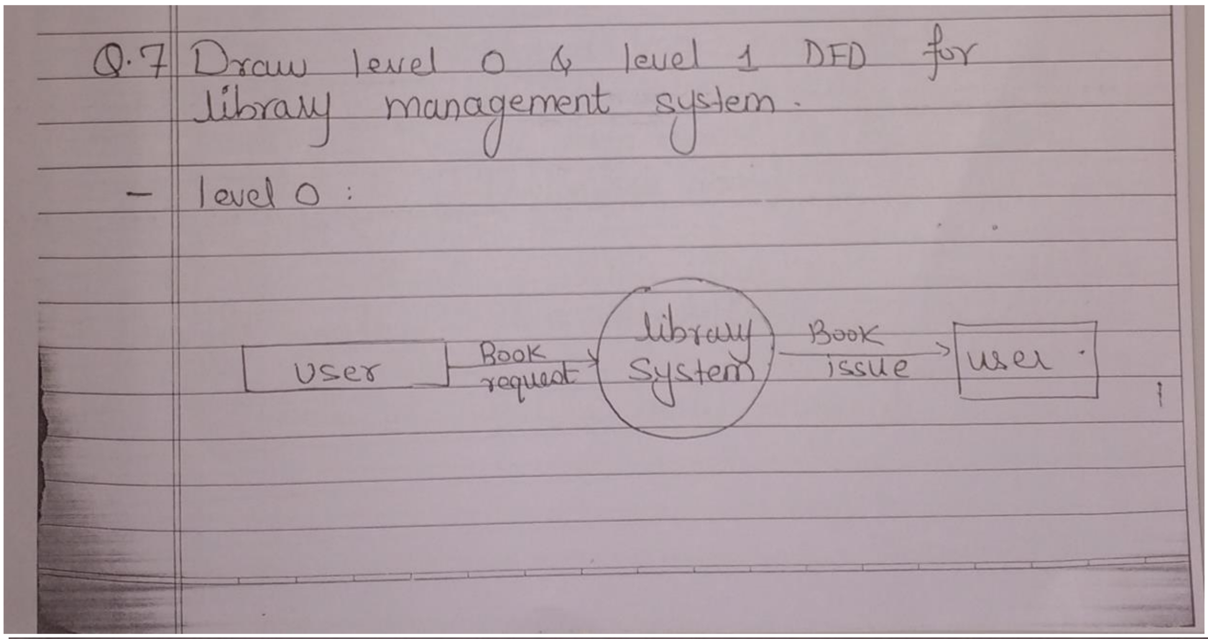
- DFD is used to represent information flow, & the transformers those are applied when data moves from input to output.
- To show the data flow with more details the DFD is further extended to level 1, level 2 etc.
- Typical value for DFD is seven. Any system can be well represented with details upto seventh level.
- Example :
- level 0 DFD for food ordering system.



- Level 1 DFD for food ordering system.

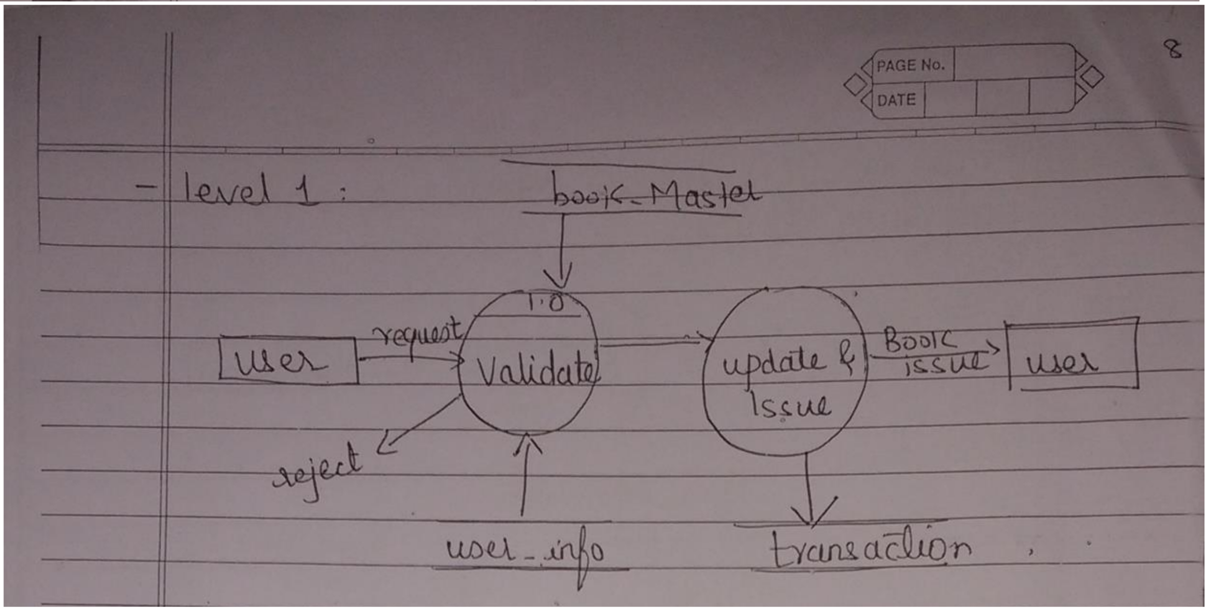


5) Draw level 0 and level 1 DFD for library management system.



Q.7 Draw level 0 & level 1 DFD for library management system.

– level 0 :



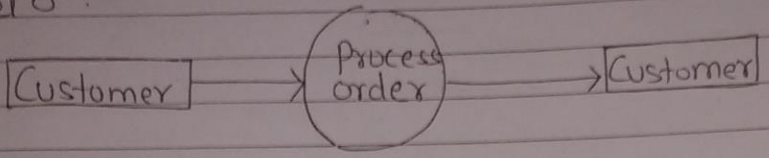
– level 1 :

6) **Draw level 0 and level 1 DFD for book publishing issues**

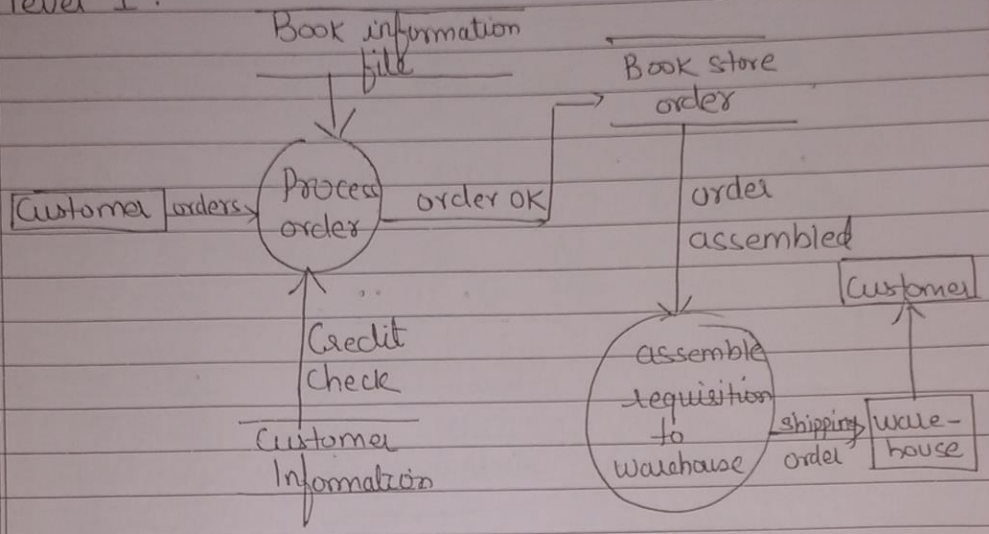
PAGE No. 7
DATE

Q.6 Draw a DFD level 0 & level 1 for a Book publishing House.

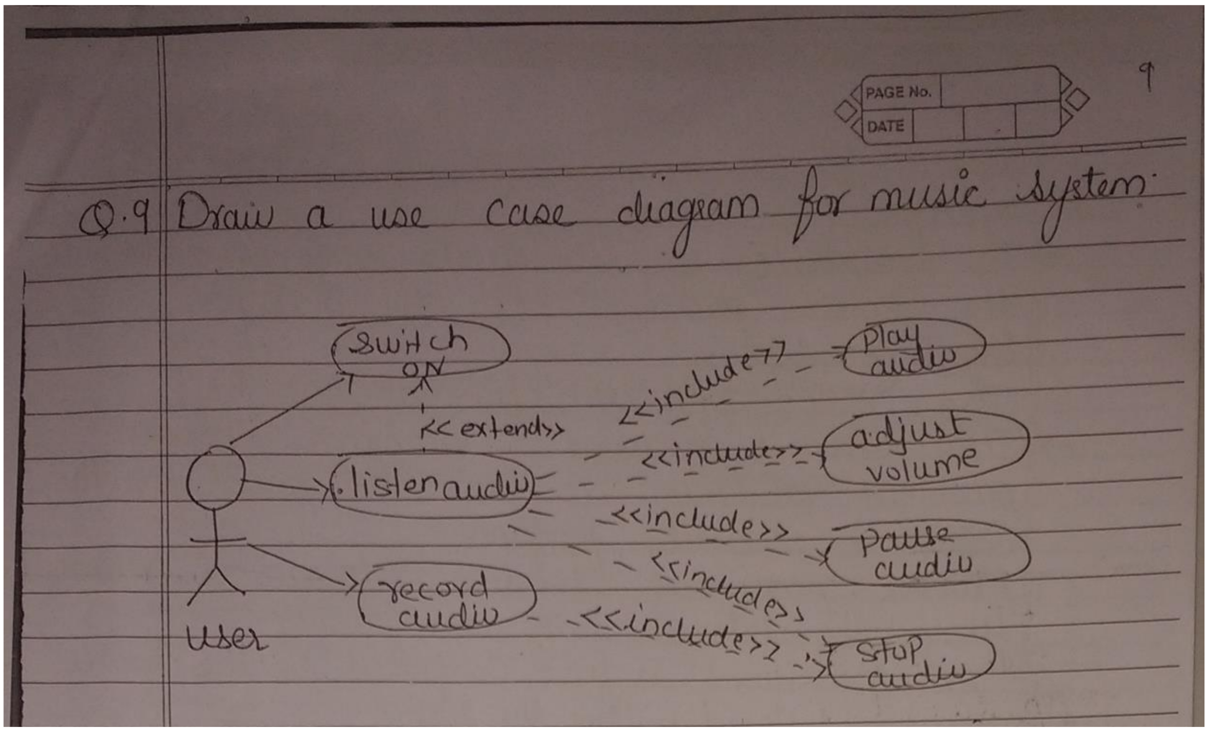
– level 0 :



– level 1 :

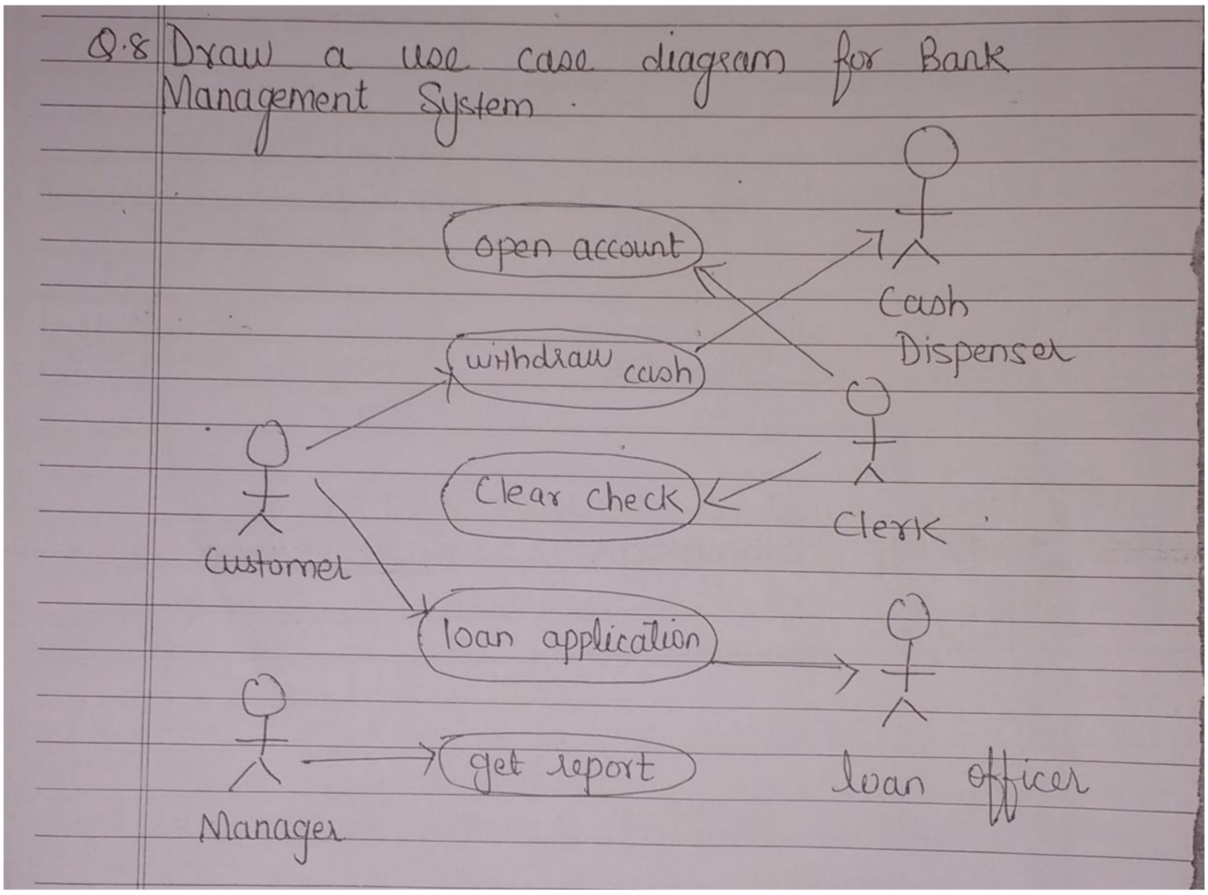


7) Draw use case diagram for music system


PAGE No. 9
DATE

Q.9 Draw a use case diagram for music system.

8) Draw use case diagram for bank management system


Q.8 Draw a use case diagram for Bank Management System.

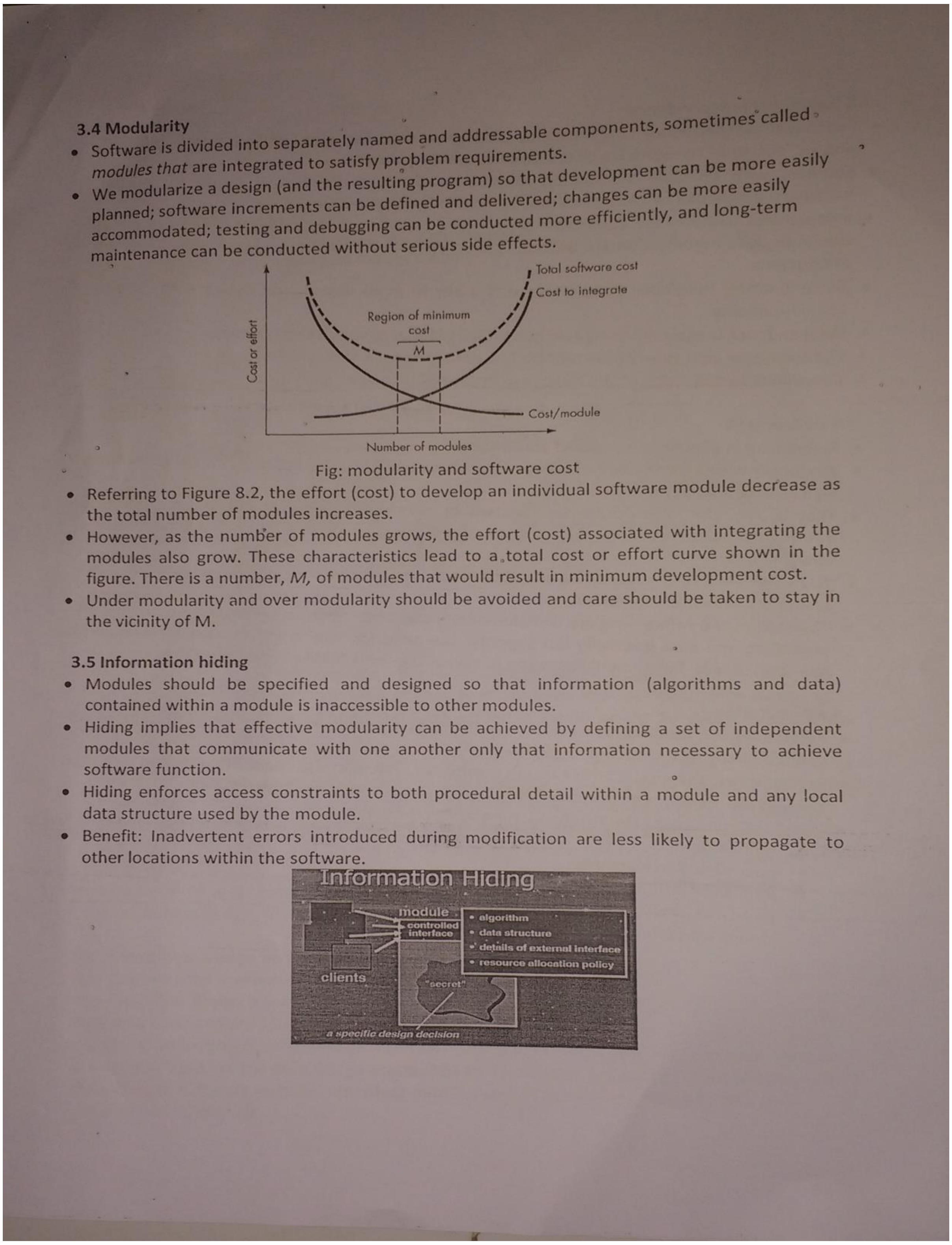
9) Describe

i) Modularity

ii) Functional independence

iii) Refactoring

iv) Information hiding



3.4 Modularity

- Software is divided into separately named and addressable components, sometimes called *modules that* are integrated to satisfy problem requirements.
- We modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

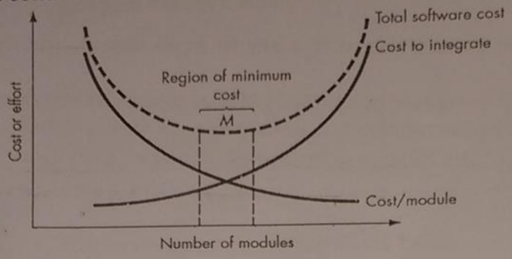
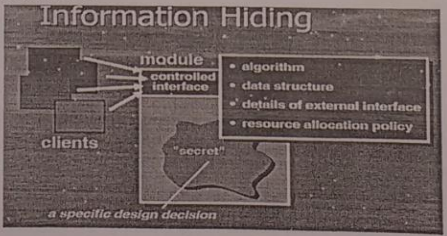


Fig: modularity and software cost

- Referring to Figure 8.2, the effort (cost) to develop an individual software module decrease as the total number of modules increases.
- However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, *M,* of modules that would result in minimum development cost.
- Under modularity and over modularity should be avoided and care should be taken to stay in the vicinity of M.

3.5 Information hiding

- Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- Hiding enforces access constraints to both procedural detail within a module and any local data structure used by the module.
- Benefit: Inadvertent errors introduced during modification are less likely to propagate to other locations within the software.



3.6 Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified.
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- To summarize, functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using two qualitative criteria: cohesion and coupling.
- *Cohesion* is an indication of the relative functional strength of a module.
- *Coupling* is an indication of the relative interdependence among modules.

3.7 Refinement

- Refinement is actually a process of *elaboration.*
- You begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

3.8 Refactoring

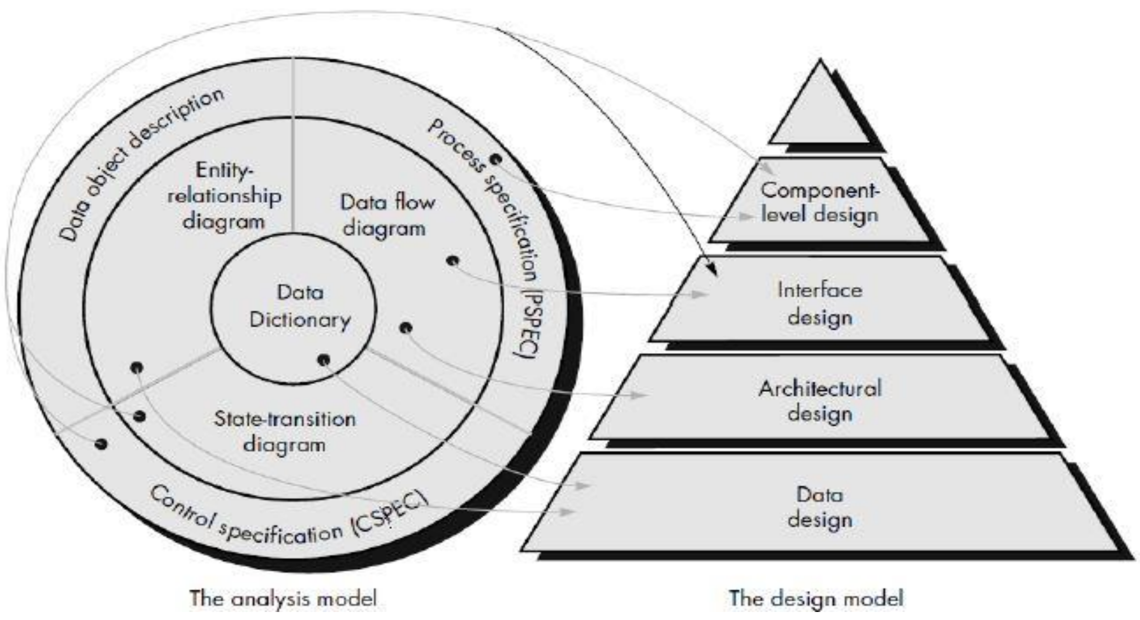
- Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for :
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - Or any other design failure that can be corrected.

10) Elements of analysis model

1. **Scenario based Elements**
   The system is described from the user 's point of view using this approach. This is often the first part of analysis model that is developed to serve as input for the creation of other modeling elements.
2. **Class-based Elements**
   Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes – a collection of things that have similar attributes and common behaviors.
3. **Behavioral Elements**
   The behavior of the system can have profound effect on the design that is chosen. The analysis model must provide modeling elements that depict the behavior. The state diagram is one of the methods for representing behavior of a system.
4. **Flow-Oriented Elements**
   The information is transformed as it flows through the computer based system. The system accepts inputs in a variety of forms, applies functions to transform it; and produces output in different forms. The transforms may comprise a single logical comparison, a complex numerical algorithm or an expert system. The elements of the information flow are included here.

11) Explain translation of analysis model to design model

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software. Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design.



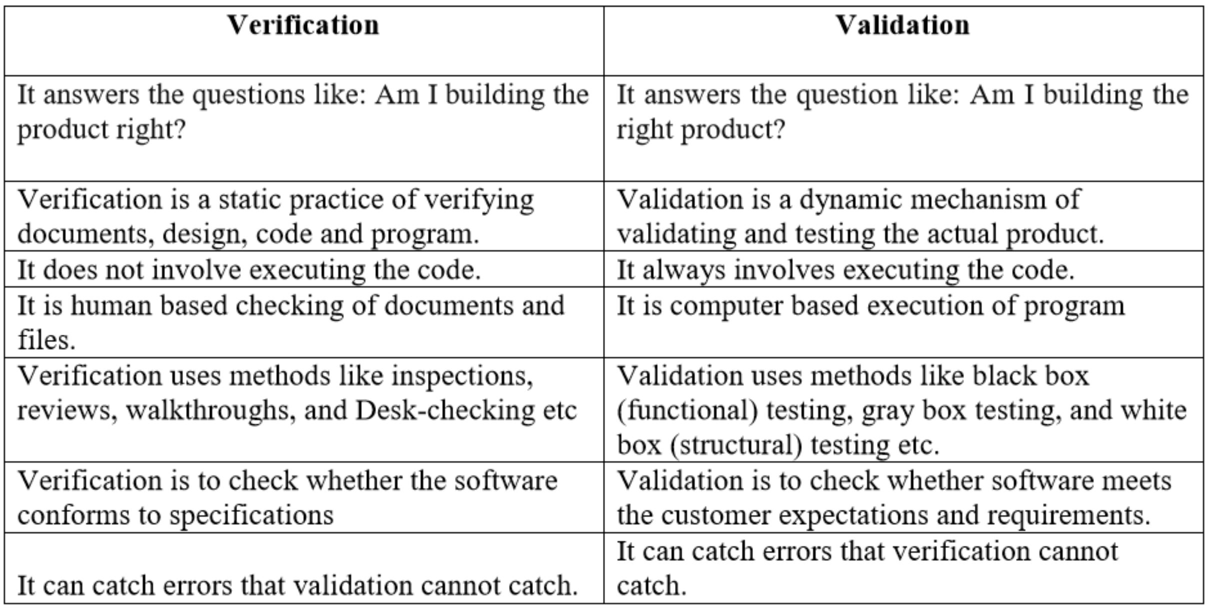
The flow of information during software design is illustrated in above figure. Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods, the design task produces a data design, an architectural design, an interface design, and a component design.
The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed. The architectural design defines the relationship between major structural elements of the software, the ―design patterns‖ that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied The architectural design representation the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model. The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design. The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

Chapter 4

1) Definition of software testing

- Testing is a process of executing a program with the intent of finding an error.
- Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which you can place specific test case design techniques and testing methods.

2) Difference between verification and validation



Verification	Validation
It answers the questions like: Am I building the product right?	It answers the question like: Am I building the right product?
Verification is a static practice of verifying documents, design, code and program.	Validation is a dynamic mechanism of validating and testing the actual product.
It does not involve executing the code.	It always involves executing the code.
It is human based checking of documents and files.	It is computer based execution of program
Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc	Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
Verification is to check whether the software conforms to specifications	Validation is to check whether software meets the customer expectations and requirements.
It can catch errors that validation cannot catch.	It can catch errors that verification cannot catch.

3) Explain unit testing with neat diagram

(a) **Unit Testing** is a level of the software testing process where individual units/components of a software/system are tested.
(b) The purpose is to validate that each unit of the software performs as designed.

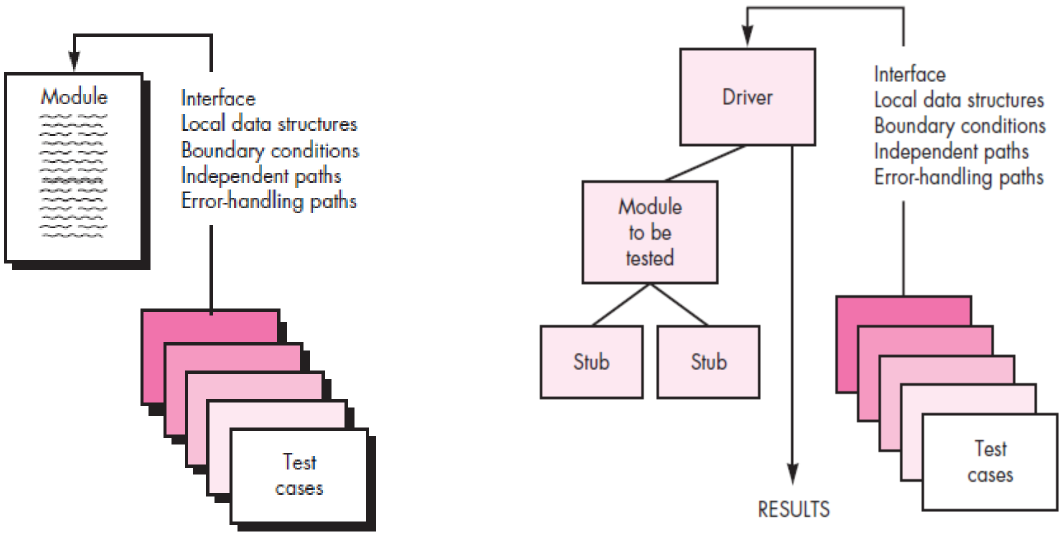


Figure: Unit Testing

(c) A unit is the smallest testable part of software.

(d) It usually has one or a few inputs and usually a single output.

(e) In procedural programming a unit may be an individual program, function, procedure, etc.

(f) In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Advantages

(a) Unit testing increases confidence in changing/maintaining code.

(b) If good unit tests are written and if they are run every time any code is changed, the likelihood of any defects due to the change being promptly caught is very high.

(c) If unit testing is not in place, the most one can do is hope for the best and wait till the test results at higher levels of testing are out.

(d) If codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

(e) Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.

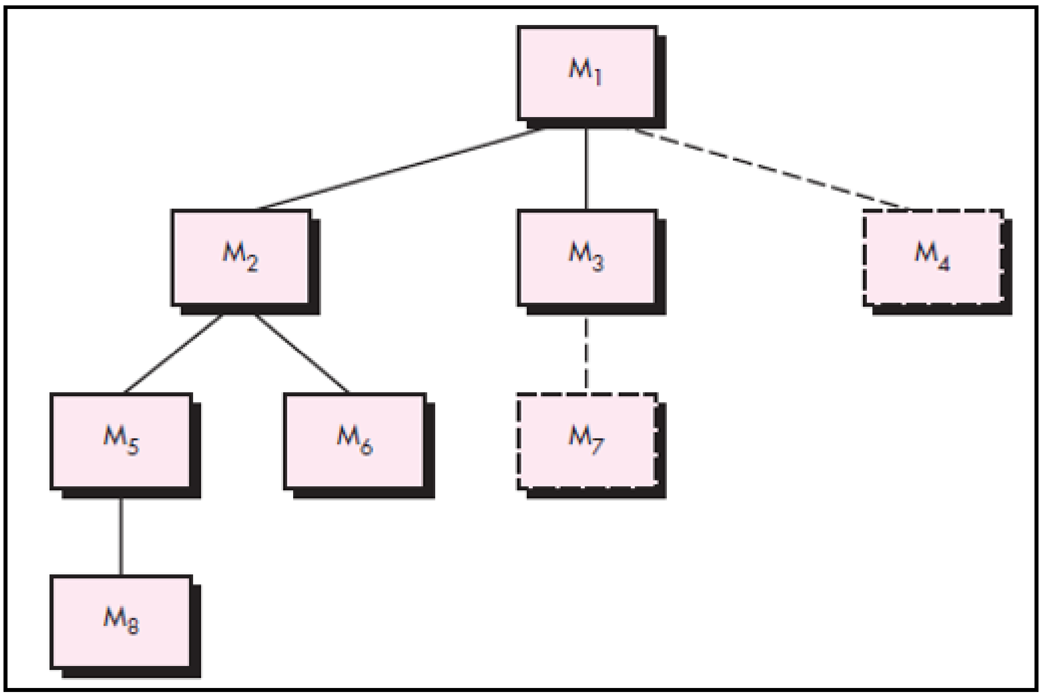
(f) The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.

(g) Compare the cost (time, effort, destruction, humiliation) of a defect detected during acceptance testing or say when the software is live.

(h) Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/weeks/months need to be debugged.

4) Explain Top-down integration testing

Top-down integration testing is an incremental approach to construction of the software architecture.
Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.



The integration process is performed in a series of five steps:
1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

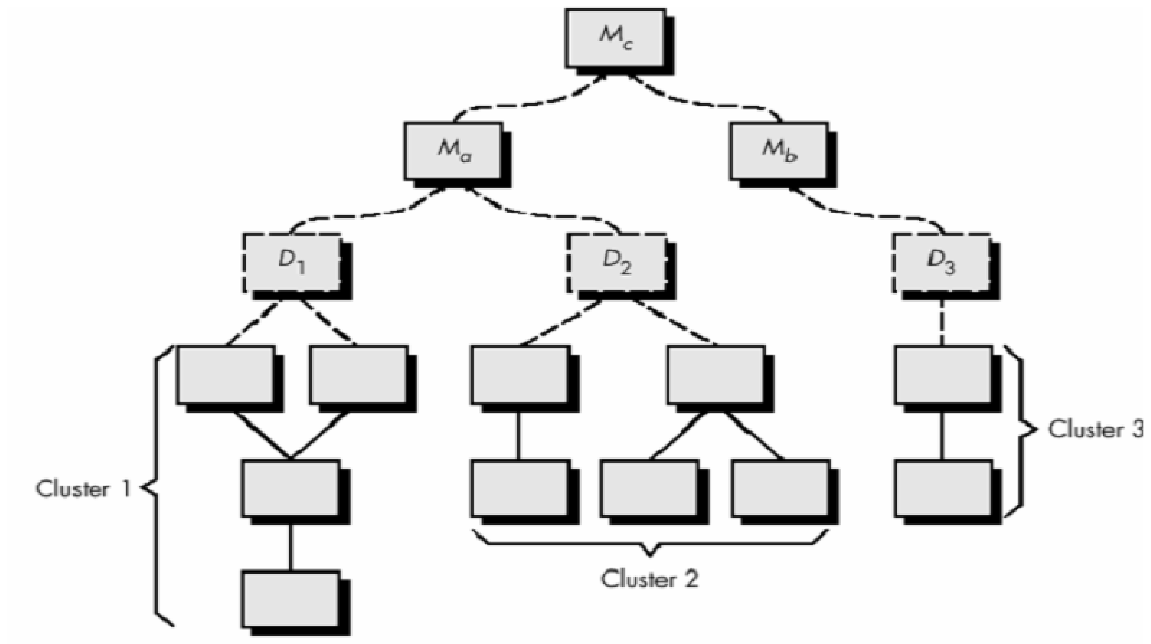
3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

5) Explain bottom-up integration testing

- *Bottom-up integration testing*, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure).
- Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.



A bottom-up integration strategy may be implemented with the following steps:
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.

2. A *driver* (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

6) Explain system testing with its types (Recovery, security, stress, performance)

Recovery testing:

• Is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

• If recovery is automatic, re-initialization check pointing mechanisms/ data recovery and restart are evaluated for correctness.

• If recovery requires human intervention, the mean time to repair is evaluated to determine whether it
is within acceptable limits

Security testing:

• It verifies that protection mechanisms built into a system will in fact protect it from proper penetration.

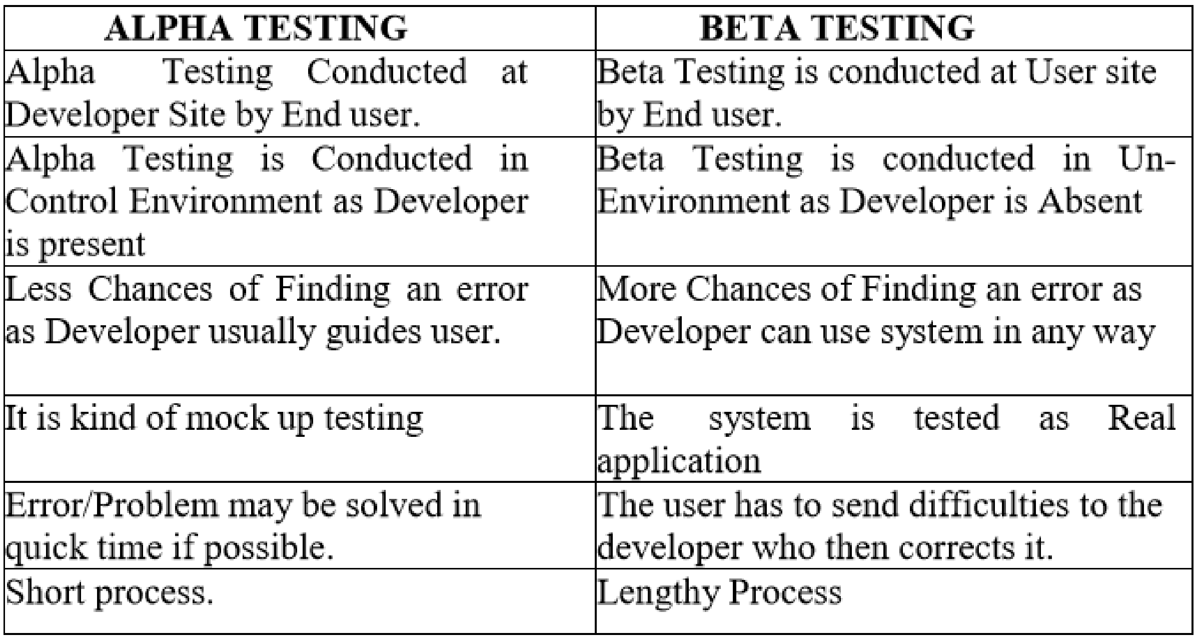
Stress testing:

• It executes a system in a manner that demands resources in abnormal quantity frequency of volume.

Performance testing:

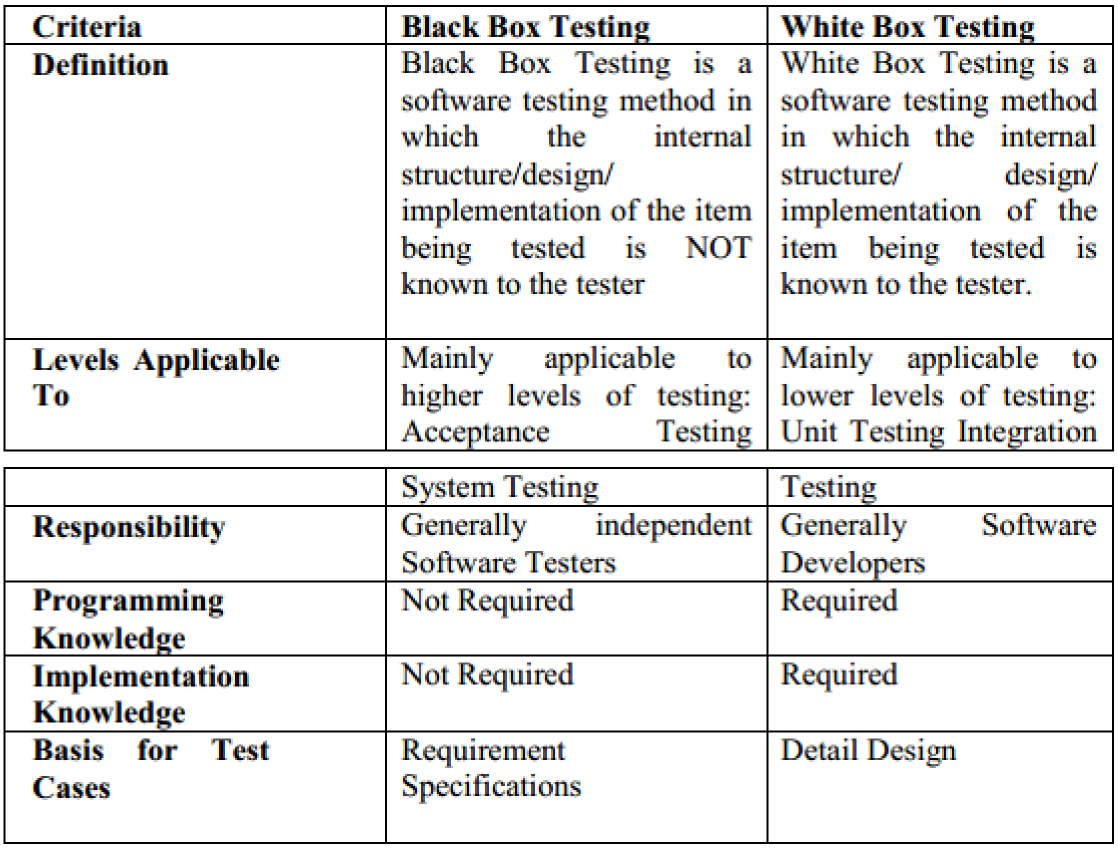
• It is designed to test the run time performance of software within the context of an integrated system.

7) Difference between alpha and beta testing



ALPHA TESTING	BETA TESTING
Alpha Testing Conducted at Developer Site by End user.	Beta Testing is conducted at User site by End user.
Alpha Testing is Conducted in Control Environment as Developer is present	Beta Testing is conducted in Un-Environment as Developer is Absent
Less Chances of Finding an error as Developer usually guides user.	More Chances of Finding an error as Developer can use system in any way
It is kind of mock up testing	The system is tested as Real application
Error/Problem may be solved in quick time if possible.	The user has to send difficulties to the developer who then corrects it.
Short process.	Lengthy Process

8) Difference between white-box testing and black-box testing



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

9) Explain debugging strategies

Three debugging strategies are
1. Brute Force
2. Backtracking
3. Cause elimination

1. Brute Force:

• The *brute force* category of debugging is probably the most common and least efficient method
for isolating the cause of a software error.

• You apply brute force debugging methods when all else fails. Using a “let the computer find the
error” philosophy, memory dumps are taken, run-time traces are invoked, and the program is
loaded with output statements.

• You hope that somewhere in the morass of information that is produced you'll find a clue that
can lead to the cause of an error.
• Although the mass of information produced may ultimately lead to success, it more frequently
leads to wasted effort and time. Thought must be expended first!

2. Backtracking

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

3. Cause elimination

• The third approach to debugging is *cause elimination*—is manifested by induction or
deduction and introduces the concept of binary partitioning.
• Data related to the error occurrence are organized to isolate potential causes. A "cause
hypothesis" is devised and the aforementioned data are used to prove or disprove the
hypothesis.
Alternatively, a list of all possible causes is developed and tests are conducted to eliminate
each. If initial tests indicate that a particular cause hypothesis shows promise, data are
refined in an attempt to isolate the bug.

Chapter 5

1) Why do the software projects fail? Give reasons.

• Software people don't understand their customer's needs
• The product scope is poorly defined
• Changes are managed poorly
• The chosen technology changes
• Business needs change (or are poorly defined)
• Deadlines are unrealistic
• Users are resistant
• Sponsorship is lost (or was never properly obtained)
• The project team lacks people with appropriate skills
• Managers (and practitioners) avoid best practices and lessons learned.

2) What is project scheduling? Explain its principle

Software project scheduling is an activity that distributes estimated effort across the planned
project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a
macroscopic schedule is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets
under way, each entry on the macroscopic schedule is refined into a *detailed schedule.* Here,
specific software tasks (required to accomplish an activity) are identified and scheduled.

Basic principles of software project scheduling:
I. **Compartmentalization:** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

II. **Interdependency:** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

III. **Time allocation:** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

IV. **Effort validation:** Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.

V. **Defined responsibilities:** Every task that is scheduled should be assigned to a specific team member.

VI. **Defined outcomes:** Every task that is scheduled should have a defined outcome.

VII. **Defined milestones:** Every task or group of tasks should be associated with a project milestone.

3) State four reasons why project deadlines cannot be met.

i. An unrealistic deadline established by someone outside the software development group and forced on managers and practitioners within the group.

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

iii. An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.

iv. Predictable and/or unpredictable risks that were not considered when the project commenced.

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

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

vii. Miscommunication among project staff that results in delays.

viii. A failure by project management to recognize that the
project is falling behind schedule and a lack of action to correct the problem.

4) Types of risk ?

Software risk always involves two characteristics:
1. *Uncertainty*—the risk may or may not happen; that is, there are no 100 percent probable risks.
2. *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.
• When risks are analyzed, it is important to quantify the level of uncertainty and the degree of
loss associated with each risk.
• To accomplish this, different categories of risks are considered.

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

• ***Technical risks*** threaten the quality and timeliness of the software to be produced.

• Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

• ***Business risks*** threaten the viability of the software to be built and often jeopardize the project or the product.

Candidates for the top five business risks are

(1) building an excellent product or system that no one really wants (market risk),

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

(3) building a product that the sales force doesn't understand how to sell (sales risk),

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

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

• ***Known risks*** are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

• ***Predictable risks*** are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

• ***Unpredictable risks*** are the joker in the deck.

5) Risk refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and
more is learned about the project and the risk, it may be possible to refine the risk into a set of
more detailed risks, each somewhat easier to mitigate, monitor, and manage. One way to do this
is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated
in the following form: Given that <condition> then there is concern that (possibly)
<consequence>. Using the CTC format for the reuse risk one could write:
Given that all reusable software components must conform to specific design standards and that
some do not conform, then there is concern that (possibly) only 70 percent of the planned

reusable modules may actually be integrated into the as-built system, resulting in the need to
custom engineer the remaining 30 percent of components. This general condition can be refined
in the following manner:

Sub-condition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Sub-condition 2: The design standard for component interfaces has not been solidified and may
not conform to certain existing reusable components.
Sub-condition 3: Certain reusable components have been implemented in a language that is not
supported on the target environment. The consequences associated with these refined subconditions
remain the same (i.e., 30 percent of software components must be custom
engineered), but the refinement helps to isolate the underlying risks and might lead to easier
analysis and response.

6) Describe RMMM strategy in detail.

• Risk mitigation, monitoring, and management (RMMM) plan.
• A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate Risk Mitigation, Monitoring and Management Plan.
• The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.
• Once RMMM has been documented and the project has begun, risk mitigation and monitoring
steps commence.
• Risk mitigation is a problem avoidance activity.
• Risk monitoring is a project tracking activity with three primary objectives:
(1) To assess whether predicted risks do, in fact, occur;
(2) To ensure that risk aversion steps defined for the risk are being properly applied; and
(3) To collect information that can be used for future risk analysis.
• Another job of risk monitoring is to attempt to allocate origin (what risk(s) caused which problems throughout the project).
• An effective strategy must consider three issues:
o Risk avoidance
o Risk monitoring
o Risk management and contingency planning
• If a software team adopts a proactive approach to risk, avoidance is always the best strategy.
This is achieved by developing a plan for **risk mitigation**. To mitigate this risk, project management must develop a strategy for reducing turnover.
• Among the possible steps to be taken are

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

• As the project proceeds, risk monitoring activities commence. The project manager monitors
factors that may provide an indication of whether the risk is becoming more or less likely.

• In the case of high staff turnover, the following factors can be monitored:

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.
- Potential problems with compensation and benefits.
- The availability of jobs within the company and outside it.

7) Enlist the features of SCM.

Versioning: As a project progresses, many versions of individual work products will be created.
The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.
Dependency tracking and change management: The repository manages a wide variety of relationships among the data elements stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on.
Requirements tracing: This special function depends on link management and provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, it provides the ability to identify which requirement generated any given work product (backward tracing).
Configuration management: A configuration management facility keeps track of a series of configurations representing specific project milestones or production releases.
Audit trails: An audit trail establishes additional information about when, why, and by whom
changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository.

8) Activities of SCM

Software Configuration Management Activities are: Identification of change To control and manage configuration items, each must be named and managed using an object-oriented approach

- Basic objects are created by software engineers during analysis, design, coding, or testing
- Aggregate objects are collections of basic objects and other aggregate objects
- An entity-relationship (E-R) diagram can be used to show the interrelationships among the objects

Version Control

- Combines procedures and tools to manage the different versions of configuration objects created during the software process
- An entity is composed of objects at the same revision level
- A variant is a different set of objects at the same revision level and coexists with other variants
- A new version is defined when major changes have been made to one or more objects

Change Control

- Change request is submitted and evaluated to assess technical merit and impact on the other configuration objects and budget
- Change report contains the results of the evaluation
- Change control authority (CCA) makes the final decision on the status and priority of the change based on the change report.

Software Configuration Audit

- A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the questions such as:
- Has the change specified in the ECO been made? Have any additional modifications been incorporated?
- Has a formal technical review been conducted to assess technical correctness?

Status Reporting Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

1. What happened?
2. Who did it?
3. When did it happen?
4. What else will be affected?

9) Describe people factor in software management spectrum.

The People
The people factor is so important that the Software Engineering Institute has developed a people management capability maturity model (PM-CMM), to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability. The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices. The PM-CMM is a companion to the software capability maturity model that guides organizations in the creation of a mature software process.
Following are the various categories of people associated with the project.

- The Stakeholders: - This include Senior managers, Project (technical) managers, Practitioners,
Customers, End user
- Team Leaders: - Leaders of various teams associated with the project
- The Software Team: - Entire software team
- Agile Teams: - Temporary teams associated with software
- Coordination and Communication Issues

Chapter 6

1) **What are SQA activities**

SQA activities

These activities are performed (or facilitated) by an independent SQA group that:

1. Prepares an SQA plan for a project.

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

2. Participates in the development of the project's software process description.

The software team selects a process for the work to be performed. The SQA group reviews the
process description for compliance with organizational policy, internal software standards, externally
imposed standards

3. Reviews software engineering activities to verify compliance with the defined software process.

The SQA group identifies, documents, and tracks deviations from the process and verifies that
corrections have been made.

**4. Audits designated software work products to verify compliance with those defined as part of
the software process.**

The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies
that corrections have been made; and periodically reports the results of its work to the project manager.

**5. Ensures that deviations in software work and work products are documented and handled
according to a documented procedure.**

Deviations may be encountered in the project plan, process description, applicable standards, or
software engineering work products.

6. Records any noncompliance and reports to senior management.

Noncompliance items are tracked until they are resolved.

2) Explain DMAIC and DMADV of six sigma strategy.

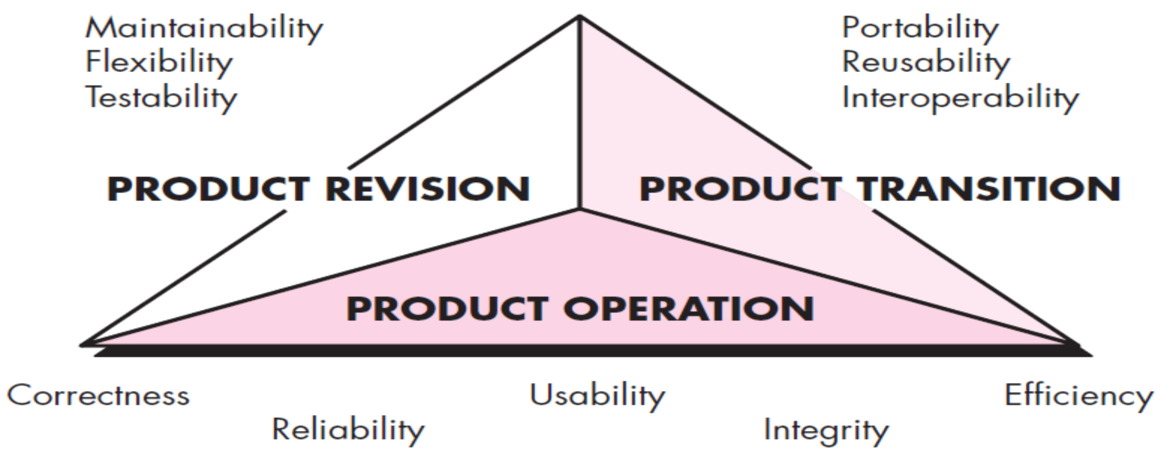
DMAIC
The DMAIC project methodology has five phases:
Define the system, the voice of the customer and their requirements, and the project goals, specifically.
Measure key aspects of the current process and collect relevant data.
Analyze the data to investigate and verify cause-and-effect relationships. Determine what the relationships are, and attempt to ensure that all factors have been considered. Seek out root cause of the defect under investigation.
Improve or optimize the current process based upon data analysis using techniques such as design of experiments or mistake proofing, and standard work to create a new, future state process. Set up pilot runs to establish process capability. Control the future state process to ensure that any deviations from target are corrected before they result in defects.
Implement control systems such as statistical process control, production boards, visual workplaces, and continuously monitor the process.
Some organizations add a Recognize step at the beginning, which is to recognize the right problem to work on, thus yielding an RDMAIC methodology.

DMADV or DFSS
The DMADV project methodology, known as DFSS ("Design For Six Sigma"), features five phases:
Define design goals that are consistent with customer demands and the enterprise strategy.
Measure and identify CTQs (characteristics that are Critical To Quality), product capabilities, production process capability, and risks.
Analyze to develop and design alternatives
Design an improved alternative, best suited per analysis in the previous step
Verify the design, set up pilot runs, implement the production process and hand it over to the process owner(s).

3) What are the benefits of ISO 9000.

- Well defined and documented procedures improve the consistency of output
- Quality is constantly measured
- Procedures ensure corrective action is taken whenever defects occur
- Defect rates decrease
- Defects are caught earlier and are corrected at a lower cost
- Defining procedures identifies current practices that are obsolete or inefficient
- Documented procedures are easier for new employees to follow
- Organizations retain or increase market share, increasing sales or revenues
- Improved product reliability
- Better process control and flow
- Better documentation of processes
- Greater employee quality awareness
- Reductions in product scrap, rewords and rejections.

4) Explain the McCall's Quality factors.(6M)



• McCall's software quality factors, shown in figure above, focus on three important aspects of a
software product:

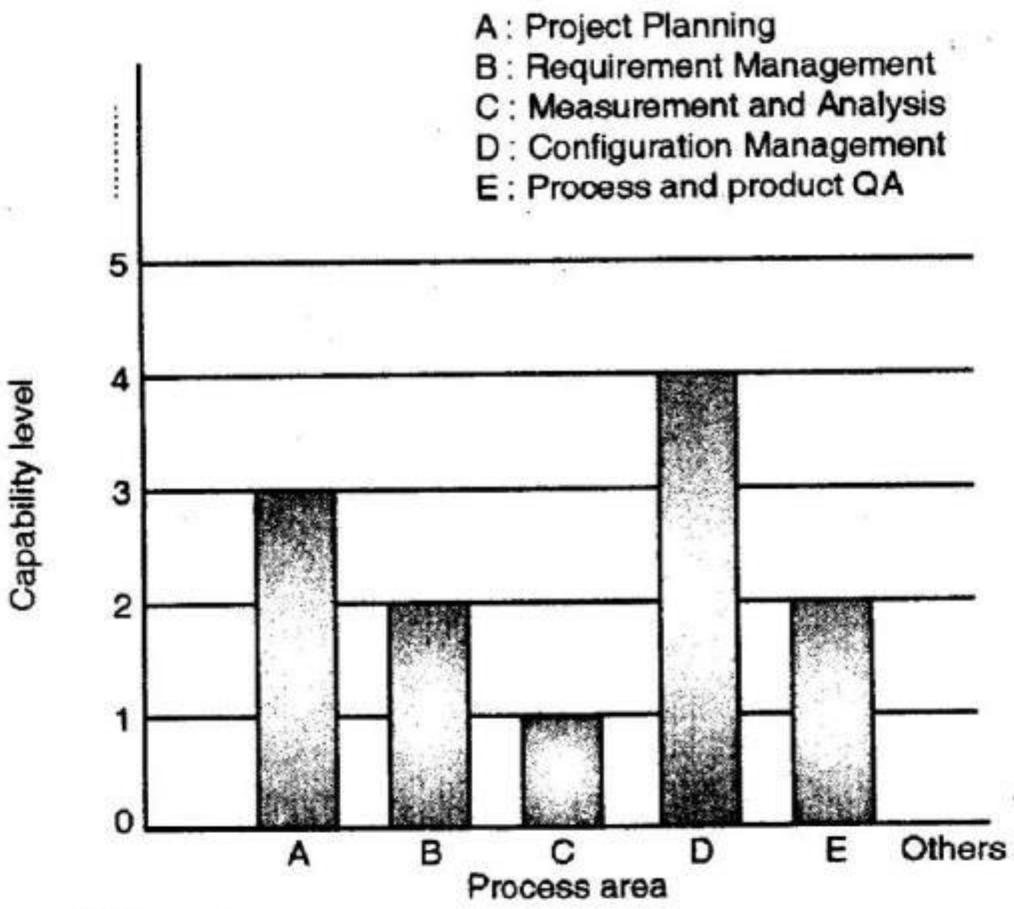
○ its operational characteristics,
○ its ability to undergo change,
○ its adaptability to new environments.

• The factors/attributes include:

○ **Correctness:** The extent to which a program satisfies its specification and fulfills the customer's
mission objectives.
○ **Reliability:** The extent to which a program can be expected to perform its intended function with
required precision
○ **Efficiency:** The amount of computing resources and code required by a program to perform its
function.
○ **Integrity:** Extent to which access to software or data by unauthorized persons can be controlled.
○ **Usability:** Effort required to learn, operate, prepare input for, and interpret
○ **Maintainability:** Effort required to locate and fix an error in a program.
○ **Flexibility:** Effort required to modify an operational program.
○ **Testability:** Effort required to test a program to ensure that it performs its intended function.
○ **Portability:** Effort required to transfer the program from one hardware and/or software system
environment to another.
○ **Reusability:** Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
○ **Interoperability:** Effort required to couple one system to another.

5) Explain CMMI with neat diagram

• Capability Maturity Model Integration (CMMI) is a process improvement approach that helps organizations improves their performance.
• CMMI (Capability Maturity Model Integration) is a proven industry framework to improve product quality and development efficiency for both hardware and software
• Objectives of CMMI:
➢ Specific Objectives
a. Establish Estimates
b. Develop a Project Plan
c. Obtain Commitment to the Plan
➢ Generic Objectives:
a. Achieve Specific Goals
b. Institutionalize a Managed Process
c. Institutionalize a Defined Process
d. Institutionalize a Quantitatively Managed Process
e. Institutionalize an Optimizing Process
CMMI maturity levels:
• **Level 1: Initia**l. The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
• **Level 2: Repeatable**. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
• **Level 3: Defined**. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level
• **Level 4: Managed**. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level
• **Level 5: Optimizing**. Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.



: CMMI process area capability Profile