1) What do you mean by software complexity?

- Software complexity is a way to describe a specific set of characteristics of your code. These characteristics all focus on how your code interacts with other pieces of code.
- The measurement of these characteristics is what determines the complexity of your code. It's a lot like a software quality grade for your code. The problem is that there are several ways to measure these characteristics.
- We're not going to look at all these different measurements. (It wouldn't be super useful to do so anyway.) Instead, we're going to focus on two specific ones: cyclomatic complexity and NPath. These two measurements are more than enough for you to evaluate the complexity of your code.

Complex vs complicated

- The idea that code feels complex or is harder to understand is worth discussing. That's because there's a term that we use to describe that type code: complicated. It's also common to think that complex and complicated mean the same thing.
- But that's not quite the case. We use these two terms to describe two different things in our code. The confusion comes from the fact that our code is often both complex and complicated.
- So far, we've only discussed the meaning of complex. When we say that code is complex, we're talking about its level of complexity. It's code that has a cyclomatic complexity value. (Or a high value in another measurement method.) It's also something that's measurable.

2) Is software complexity essential?

Essential complexity is the entanglement(complication)/combination of components/ideas in software necessary for solving the problem at hand. It cannot be avoided.

The complexity in s/w is essential, which are derived from following 4 elements.

1. Complexity of Problem Domain.
2. Difficulty of managing developmental process.
3. Flexibility possible through software.
4. Problems of characterizing the behavior of discrete system.

3) Five attributes of a complex system

i. **Hierarchic Structure:** Complexity takes the form of hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elementary components is reached.

ii. **Relative Primitives:** "The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system." i.e a primitive for one observer may be higher level of abstraction to another.

iii. **Separation of Concerns:** " intra-component linkage are generally stronger than inter component linkages." i.e. each part is in relative isolation. This fact has the effect of separating the high-frequency dynamics of the components—involving the internal structure of the components—from the low-frequency dynamics—involving interaction among components.
This difference between intra- and inter-component interactions provides a clear *separation of concerns* among the various parts of a system, making it possible to study each part in relative isolation.
   - Inter- is a common prefix that means between or among groups.
   - Intra- means within or inside.
   - For example, while the internet is a system that connects computers around the world, an intranet, is a network of computers that only connects people within a certain group, such as employees at a company.

iv. **Common Patterns:** " Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements". i.e complex systems have common patterns, which involve the reuse of small components. Example:- cells found in both plants or animals, or of larger structures, such as vascular systems, also found in both plants and animals.

v. **Stable Intermediate Forms:** " A complex system that works is invariably found to have evolved from a simple system that worked". i.e A complex system designed from scratch never works and can not be patched up to make it work. You have to start over, beginning with a working simple system

4) Define OOP, OOA and OOD

Object Oriented Programming

Object Oriented Programming is a very loaded term with many conflicting definitions, some people don't even consider object orientation to be a thing!

Alan Key the author of Smalltalk the first language credited as being "object oriented" was credited to saying that when he coined the term "Object-Oriented" he didn't have C++ in mind (which uses a similar class mode of object orientation as Java).

Smalltalk was more about sending messages to separate entities called object similar to an actor system and not what we have in C++ or Java. Some languages that are considered object oriented (or to have object orientation) are very dissimilar to Java for example Java Script or Lua with their prototype based object orientation.

Because of how "shifty" the definition of OOP can be I'll try to give one that's most useful to Java programmers. OOP is a programing paradigm where you program with instances of classes (called objects) which have their own identity and may have

their own data. In Java this is done through class based design – each object is an instance of some class a "design schema" for similar objects.
Some things from the Java programming language that don't match this OOP model:

- primitives
- static fields and methods
- reflection

Object Oriented Analysis

Object Oriented Analysis in software engineering is creating a model (a "simplified" version of the real system) that encompasses all of the functional requirements.
What differs OOA from other forms of analysis is that we use the object oriented approach to create the model. I.e we organize the requirements using objects (often modelled around real life objects that the model should represent).
Common models used in OOA are:

- use cases – where we try to model specific senarios of usage for the system.
- object models – where we try to model the running the system and all relationships ("is a " – subtyping, composition "has a", "implements a" – interfaces).

OOA used in the concrete analysis of one thing, generally follow the five basic steps:

i. The first step, to determine the **object and class**. The object here is the data and its processing method of abstract, it reflects the ability of information system to save and deal with some things in the real world. The class is the set of common properties and method of multi object is described, including how to establish a new object in a class description.
ii. The second step, to determine the **structure (structure).** Structure refers to the complexity of the problem domain and connection. Members of the class structure reflects the generalization specialization relationships, part whole structure reflects the relationship between the global and local.
iii. The third step, to determine the **theme (subject).** Theme refers to the overall situation and the overall analysis model.
iv. The fourth step, to determine **attribute (attribute).** Attribute data elements, can be used to describe the object or the classification structure, can be given in the chart, and specified in the object store.
v. The fifth step, determining **method (method).** Method is some processing methods must be carried out after receiving the news: a method to define in the graph, and specified in the object store. For each object and structure, used to add, modify, delete those and choose a method itself is implied (although they are to be defined in the object in the store, but not in the graph are given), while others are displayed.

Object Oriented Design

In software engineering Object Oriented Design is the "just" the implementation phase of the project where a programmer will try to implement a system that fits the model created during the OOA phase.
Here the programmer will usually create separate classes to handle some group functionality, communication formats, persistent data etc.Another important topic of OOD is the usage of architectural patterns for example MVC (Model-View-Controller), Publish-Subscribe, Data Transfer Objects n-tier architecture etc. and the use of Software Design Patterns for example Singleton, Component/Composite, Bridge, Proxy etc.

5) Explain general characteristics of OOD with an example of each characteristics.

i. ***Class & Object:*** A Class defines the abstract characteristics of a thing (object) in other word a class is a *blueprint* or *factory* that describes the nature of something. Informally **Object** is a tangible entity, that exhibits some well defined behavior. Also, an object is a pattern of class, which serve to unify the ideas of algorithmic and data abstraction.
ii. **Abstraction** is simplifying complex reality by modeling classes appropriate to the problem. An Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide defined conceptual boundaries, relative to the perspective of the viewer.
As per seidewitz and Stark, the abstraction may be
    1. **Entity Abstraction**: An object that represent useful model of problem domain or solution domain entity.
    2. **Action Abstraction:** general set of operations.
    3. **Coincidental Abstraction:**An object that packages a set of operations that have no relation to each other.
iii. **Encapsulation:** It hides the details of the implementation of an object. Encapsulation conceals the functional details of a class from objects that send messages to it. Abstraction helps people to think about what they are doing, encapsulation allows program changes to be reliably made with little effort. Encapsulations provides explicit barriers among different abstractions and thus leads to a clear separation of concerns.

iv. **Modularity**: Act of partitioning program into individual components. Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. Modules serve as the physical containers in which we declare the classes and objects of our logical design. Modularity is the property of a system that has

been decomposed into set of cohesive and loosely coupled modules. Thus, the principles of abstraction, encapsulation and modularity are synergistic.

v. **Hierarchy**: It is a ranking or ordering of abstractions. A set of abstraction often forms a hierarchy. Two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).
   - That is, Hierarchies can be studied under 2 broader headings:
   1. **Inheritance (IS-A** type relationship**)**: inheritance is a *generalization/specialization* hierarchy. That is, super classes represent generalized abstraction, and subclasses represent specializations in which fields and methods from the super classes are added, modified or even hidden. Types:- (i) Single Inheritance (ii) Multiple Inheritance
   2. **Aggregation**: It is a "Part Of" Hierarchy. Here a superclass is at higher level of abstraction than the subclass. For example, Mr. X is a part of class Msc-1st. The two are not dependent with each other that is, removing any will not lead to their end.

vi. **Typing**: This concept is evolved from ADT: *Abstract Data Type, which represents precise characterization of structural or behavioral properties which, all entities share. Class and type are used interchangeably.* Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.
   - **Strong and Weak typing**: In strong typing, operations can not be called upon an object unless the exact signature of that operation is defined in the objects class or superclass.
   - **Static and Dynamic typing**: Static typing also called as early binding-refers to the time when names are bound to types. That is, types of all variables and expressions are fixed at the time of compilation. Dynamic typing also called as late binding means that types of all variables and expressions are not known until run time.

vii. **Concurrency** is the property that distinguishes an active object from one that is not active. That is simultaneous running of the different process.

viii. **Persistence**: There are large no. objects in the s/w taking space, ranging from evaluation of an expression to objects in database that outlive the execution of a single program.

6) Explain the different relationships that exist between classes and object with a proper example of each.

One of the advantages of Object-Oriented programming language is code reuse. This reusability is possible due to the relationship b/w the classes. Object oriented programming generally support 4 types of relationships that are: inheritance, association, composition and aggregation. All these relationship is based on "is a" relationship, "has-a" relationship and "part-of" relationship.

- **Inheritance:** Inheritance is "IS-A" type of relationship. "IS-A" relationship is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. Inheritance is a parent-child relationship where we create a new class by using existing class code. It is just like saying that "A is type of B". For example is "Apple is a fruit", "Ferrari is a car".
- **Composition:** Composition is a "part-of" relationship. Simply composition means mean use of instance variables that are references to other objects. In composition relationship both entities are interdependent of each other for example "engine is part of car", "heart is part of body".
- **Polymorphism:** Polymorphism is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways. With polymorphism, an operation can be implemented differently by the classes in the hierarchy.
  The operation location is common to all subclasses and therefore need not be redefined, but we expect the operations draw and move to be redefined since only the subclasses know how to draw and move themselves.
- **Association:** Association is a "has-a" type relationship. Association establish the relationship b/w two classes using through their objects. Association relationship can be one to one, One to many, many to one and many to many. For example suppose we have two classes then these two classes are said to be "has-a" relationship if both of these entities share each other's object for some work and at the same time they can exists without each other's dependency or both have their own life time.
  - **Multiplicity or Cardinality**: An example of this kind of association is many accounts being registered by the bank. Hence, the relationship shows a star sign near the student class (one to many and many to many etc.
  - **Directed Association**: By default, an association that exists between classes is bi-directional. Ideally, you may illustrate the flow of the association by utilizing a directed association. The arrowhead indicates the container-contained relationship.
  - **Reflexive Association**: An example here is when a class has many different types of responsibilities. For example, an employee of a company can be an executive, assistant manager, or a CEO. There is no symbol that can be used here, however, the relation will point back at the same class.
- **Aggregation:** Aggregation is based is on "has-a" relationship. Aggregation is a special form of association. In association there is not any classes (entity) work as owner but in aggregation one entity work as owner. In aggregation both entities meet for some work and then get separated. Aggregation is a one way association.

Example: Let us take an example of "Student" and "address". Each student must have an address so relationship b/w Student class and Address class will be "Has-A" type relationship but vice versa is not true (it is not necessary that each address contain by any student). So Student work as owner entity.

7) What is domain class model?

- a domain model is a conceptual model of the domain that incorporates both behavior and data.
- domain model is a system of abstractions that describes selected aspects of a sphere of knowledge, influence or activity (a domain). The model can then be used to solve problems related to that domain. The domain model is a representation of meaningful real-world concepts pertinent to the domain that need to be modelled in software. The concepts include the data involved in the business and rules the business uses in relation to that data.
- A domain model generally uses the vocabulary of the domain, thus allowing a representation of the model to be communicated to non-technical stakeholders. It should not refer to any technical implementations such as databases or software components that are being designed.
- A domain model is generally implemented as an object model within a layer that uses a lower-level layer for persistence and "publishes" an API to a higher-level layer to gain access to the data and behavior of the model.
- In the Unified Modeling Language (UML), a class diagram is used to represent the domain model.

8) Explain the each step to construct the domain model.

1. Find classes.
2. Prepare a data dictionary.
3. Find associations.
4. Find attributes of objects and links.
5. Organize and simplify classes using inheritance.
6. Verify that access path exists for likely queries.
7. Iterate and refine the model.
8. Reconsider the level of abstraction.
9. Group classes into packages

9) Explain the role of domain analysis in identification of classes.

Having a formal, systematic reuse process assumes the existence of software to reuse; domain analysis helps identify this software. A domain analysis primarily consists of a well-structured, intense study of a collection of problems or a collection of applications [Ara93]. A domain analysis helps answer the questions:

- What software can we reuse?
- How do we make it reusable?

A domain analysis helps identify the common parts of a problem; in other words, those features which will translate later into shared software. The domain analysis also helps develop an understanding of the problem which, when documented, leads to models of the domain, preliminary design documents, and interface specifications. The domain analysis process consists of characterizing and understanding the *problem space.* The analyst has the goal of factoring out commonality in the problems so as to identify common, shared threads. The analyst then attempts to characterize and understand the *solution space* for the domain. The process culminates in a mapping of the problem space to the solution space by modeling a framework, architecture, design, and/or software to generate applications within the domain [Tra93]. *Domain engineering* differs from domain analysis in that domain engineering includes the actual construction of the software system.

10) What do you mean by software quality?

SOFTWARE QUALITY is the degree of conformance to explicit or implicit requirements and expectations.

Software quality is defined as conformance to explicitly stated functional and non-functional requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

- Functional attributes (or what work the product performs)
- Non-functional attributes (product properties other than work it performs)
  - Performance Efficiency (time behavior) – Does the printer function respond within three seconds?
  - Compatibility (interoperability) – Can the user print over a variety of networks and printers and on computers with different operating systems (Windows and Mac)?
  - Usability (learnability) – Can the user figure out how to print or will it take a rocket scientist?
  - Reliability (recoverability) – When the printer is unplugged in the middle of printing a task, is the user notified?
  - Security (non-repudiation) – Is there a record that the printer printed the file successfully?
  - Maintainability (testability) – Can test criteria be specified for the print function?
  - Portability (adaptability) – Can the software automatically adapt to new printer models, or an update in printer driver software? Can the print function provide shortcuts for highly sophisticated users?

11) Mention the internal quality of a design.

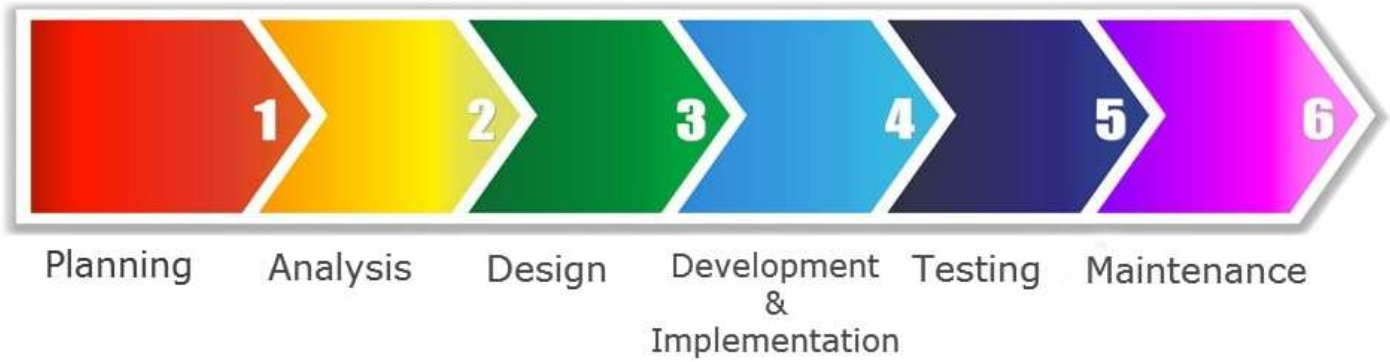
Internal Quality (Structural)

- Internal quality has to do with the way that the system has been constructed. It is a much more granular measurement and considers things like clean code, complexity, duplication, component reuse. This quality can be measured through predefined standards, linting tools, unit tests etc. Internal quality affects your ability to manage and reason about the program.
- Is your program able to cope with new requirements easily moving forward? Is your program efficient enough to deal with an inevitable increase in data volume? Is your domain logic decoupled from the framework so that it can be updated without breaking the system? Do you have tests to guard existing functionality?

These are some of the questions that need to be answered in order understand internal quality.

There are two common aspects of quality: one of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality.
– W. A. Shewhart

12) Explain the basic process involved in software development process.



- **Planning**: Without the perfect plan, calculating the strengths and weaknesses of the project, development of software is meaningless. Planning kicks off a project flawlessly and affects its progress positively.
- **Analysis**: This step is about analyzing the performance of the software at various stages and making notes on additional requirements. Analysis is very important to proceed further to the next step.
- **Design**: Once the analysis is complete, the step of designing takes over, which is basically building the architecture of the project. This step helps remove possible flaws by setting a standard and attempting to stick to it.
- **Development & Implementation**: The actual task of developing the software starts here with data recording going on in the background. Once the software is developed, the stage of implementation comes in where the product goes through a pilot study to see if it's functioning properly.
- **Testing**: The testing stage assesses the software for errors and documents bugs if there are any.
- **Maintenance**: Once the software passes through all the stages without any issues, it is to undergo a maintenance process wherein it will be maintained and upgraded from time to time to adapt to changes. Almost every software development Indian company follows all the six steps, leading to the reputation that the country enjoys in the software market today.

13) Define agent and agent oriented programming

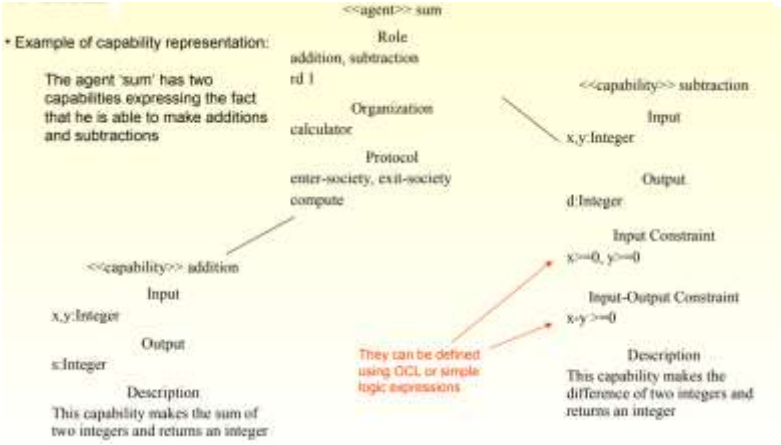
- An **agent** (sometimes called an "adaptive agent") is generally regarded to be an autonomous entity that can interact with its environment. In other words, it must be able to perceive its environment through sensors and act upon its environment with effectors.
- An **agent** is an hardware or software system placed in an environment that has the following properties: Autonomy, Social ability, Reactivity, Proactivity
- **Agent-oriented programming (AOP)** is a specialisation of object-oriented programming (OOP) in the way the computational system is conceived: "The computational system is seen as composed of communicating modules, each with its own way of handling messages." [Shoham, 1993]
- **T**he (mental) state of modules (agents) consists of components such as beliefs, capabilities and intentions.
- A computation consists of agents that:
    - Inform other agents about facts
    - Offer and request services
    - Accept or refuse proposals
    - Compete for accessing shared resources
    - Collaborate for achieving common goals

Middleware: One way to implement modular or extensible AOP support is to define standard AOP APIs to middleware functions that are themselves implemented as software agents.

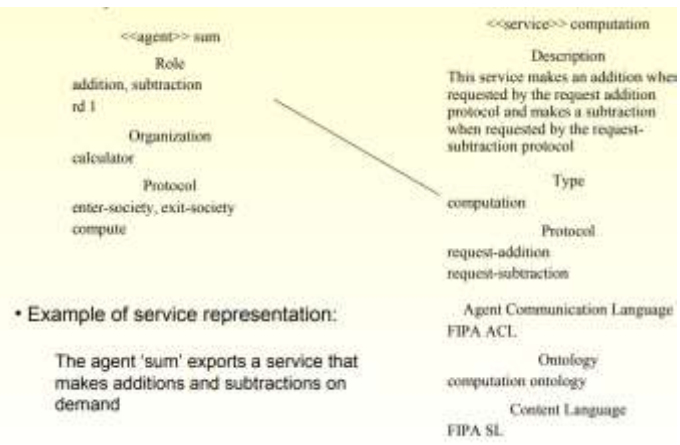
14) Explain the common features of agent and its representation.

Common features of agent are:-

- **Identifier**: identifies each agent in a multi-agent system. Representation like <<agent>> agent_name
- **Role**: defines the behaviour of an agent into the society (es. Seller, Buyer). Representation like Role1, Role2, …
- **Organization**: defines the relationships between the roles (similar to human or animal organizations such as hierarchies, markets, groups of interest or herds). Representation like organization1, organization2, …
- **Capability**: specifies what an agent is able to do and under what conditions
- **Service**: describes an activity that an agent can perform and is provided to other agents



Fig(a) Capabilities representation



Fig(b) Service representation

15) What is UML?

- UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

16) Differentiate UML from AUML.

The current UML is sometimes insufficient for modelling agents and agent-based systems. However, no formalism yet exists to sufficiently specify agent-based system development. To employ agent-based programming, a specification technique must support the whole software engineering process—from planning, through analysis and design, and finally to system construction, transition, and maintenance.

Agents share some common characteristics:

- **Identifier:** identifies each agent in a multi-agent system
- **Role:** defines the behaviour of an agent into the society (es. Seller, Buyer)
- **Organization:** defines the relationships between the roles (similar to human or animal organizations such as hierarchies, markets, groups of interest or herds)
- **Capability:** specifies what an agent is able to do and under what conditions
- **Service:** describes an activity that an agent can perform and is provided to other agents

17) What is software quality metrics?

What is Software Quality Metrics?

The word '**metrics**' refer to standards for measurements. Software Quality Metrics means measurement of attributes, pertaining to software quality along with its process of development.

The term **"software quality metrics"** illustrate the picture of measuring the software qualities by recording the number of defects or security loopholes present in the software. However, quality measurement is not restricted to counting of defects or vulnerabilities but also covers other aspects of the qualities such as maintainability, reliability, integrity, usability, customer satisfaction, etc.

Why Software Quality Metrics?

- To define and categorize elements in order to have better understanding of each and every process and attribute.
- To evaluate and assess each of these process and attribute against the given requirements and specifications.
- Predicting and planning the next move w.r.t software and business requirements.
- Improving the Overall quality of the process and product, and subsequently of project.

Software Quality Metrics: sub-category of software metrics

It is basically, a subclass of **software metrics** that mainly emphasizes on quality assets of the software product, process and project. Software metric is a broader concept that incorporates software quality metrics in it, and mainly consists of three types of metrics:

- **Product metrics:** It includes size, design, complexity, performance and other parameters that are associated with the product's quality.
- **Process metrics:** It involves parameters like, time-duration in locating and removing defects, response time for resolving issues, etc.
- **Project metrics:** It may include number of teams, developers involved, cost and duration for the project, etc.

PROCESS METRICS	PRODUCT METRICS	PROJECT METRICS
• Time in locating defect(s).	• Size and Design.	• Number of Teams.
• Resolving Time	• Complexity involved.	• Number of Developers.
• Response Time	• Performance	• Time & Cost of the Project.
• *	• Usability	• *
• *	• Security	• *
• *	• *	• *
• *	• *	• *
• *	• *	• *

18) Explain the difference between validation and verification.

Validation is the process of checking whether the specification captures the customer's needs. "Did I build what I said I would?"
Verification is the process of checking that the software meets the specification. "Did I build what I need?"

	Verification	Validation
1	Verification is a static practice of verifying documents, design, code and program.	Validation is a dynamic mechanism of validating and testing the actual product.
2	It does not involve executing the code.	It always involves executing the code.
3	It is human based checking of documents and files.	It is computer based execution of program.
4	Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
5	Verification is to check whether the software conforms to specifications.	Validation is to check whether software meets the customer expectations and requirements.
6	It can catch errors that validation cannot catch. It is low level exercise.	It can catch errors that verification cannot catch. It is High Level Exercise.
7	Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8	Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	Validation is carried out with the involvement of testing team.
9	It generally comes first-done before validation.	It generally follows after verification.

19) Explain Object Oriented Approach for Software Development.

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.

In analysis phase, OO models are used to fill the gap between problem and solution. It performs well in situation where systems are undergoing continuous design, adaption, and maintenance. It identifies the objects in problem domain, classifying them in terms of data and behavior.

The OO model is beneficial in the following ways −

- It facilitates changes in the system at low cost.
- It promotes the reuse of components.
- It simplifies the problem of integrating components to configure large system.

- It simplifies the design of distributed systems.

Elements of Object-Oriented System

Let us go through the characteristics of OO System −

- **Objects** − An object is something that is exists within problem domain and can be identified by data (attribute) or behavior. All tangible entities (student, patient) and some intangible entities (bank account) are modeled as object.
- **Attributes** − They describe information about the object.
- **Behavior** − It specifies what the object can do. It defines the operation performed on objects.
- **Class** − A class encapsulates the data and its behavior. Objects with similar meaning and purpose grouped together as class.
- **Methods** − Methods determine the behavior of a class. They are nothing more than an action that an object can perform.
- **Message** − A message is a function or procedure call from one object to another. They are information sent to objects to trigger methods. Essentially, a message is a function or procedure call from one object to another.

Features of Object-Oriented System

An object-oriented system comes with several great features which are discussed below.

Encapsulation

Encapsulation is a process of information hiding. It is simply the combination of process and data into a single entity. Data of an object is hidden from the rest of the system and available only through the services of the class. It allows improvement or modification of methods used by objects without affecting other parts of a system.

Abstraction

It is a process of taking or selecting necessary method and attributes to specify the object. It focuses on essential characteristics of an object relative to perspective of user.

Relationships

All the classes in the system are related with each other. The objects do not exist in isolation, they exist in relationship with other objects.

There are three types of object relationships −

- **Aggregation** − It indicates relationship between a whole and its parts.
- **Association** − In this, two classes are related or connected in some way such as one class works with another to perform a task or one class acts upon other class.
- **Generalization** − The child class is based on parent class. It indicates that two classes are similar but have some differences.

Inheritance

Inheritance is a great feature that allows to create sub-classes from an existing class by inheriting the attributes and/or operations of existing classes.

Polymorphism and Dynamic Binding

Polymorphism is the ability to take on many different forms. It applies to both objects and operations. A polymorphic object is one who true type hides within a super or parent class.

In polymorphic operation, the operation may be carried out differently by different classes of objects. It allows us to manipulate objects of different classes by knowing only their common properties.

Structured Approach Vs. Object-Oriented Approach

The following table explains how the object-oriented approach differs from the traditional structured approach −

Structured Approach	Object Oriented Approach
It works with Top-down approach.	It works with Bottom-up approach.
Program is divided into number of submodules or functions.	Program is organized by having number of classes and objects.
Function call is used.	Message passing is used.
Software reuse is not possible.	Reusability is possible.
Structured design programming usually left until end phases.	Object oriented design programming done concurrently with other phases.
Structured Design is more suitable for offshoring.	It is suitable for in-house development.
It shows clear transition from design to implementation.	Not so clear transition from design to implementation.
It is suitable for real time system, embedded system and projects where objects are not the most useful level of abstraction.	It is suitable for most business applications, game development projects, which are expected to customize or extended.
DFD & E-R diagram model the data.	Class diagram, sequence diagram, state chart diagram, and use cases all contribute.
In this, projects can be managed easily due to clearly identifiable phases.	In this approach, projects can be difficult to manage due to uncertain transitions between phase.

20) Develop a Use case, Class and Sequence Diagram of University Information System, Telephone Billing System, Railway Reservation System.

21) Difference between aggregation and composition with proper example.

BASIS	AGGREGATION	COMPOSITION
Basic	Exhibit a relationship where a child can exist independently of the parent.	Cannot exist independently of the parent.
Type of relationship	"has a"	"part of"
Association type	Weak association	Strong association
UML design symbol	Represented by a hollow diamond next to assembly class.	Represented by a solid diamond next to assembly class.
Function	The deletion of assembly doesn't affect its parts.	If the owning class object is deleted, it could significantly affect the containing class object.
Theme	Collection of Objects	Mixture of Objects
Association type	*Weak Association type*: If the life of contained object doesn't depends on the container object, it is called weak association)	*Strong Association Type*: If the life of contained object totally depends on the container object, it is called strong association
Example	A Company is an aggregation of People	A Company is a composition of Accounts. When a Company finishes to do business, its Accounts finish to exist but its People continue to exist.

22) What is Object Oriented Modeling ?

- Object-oriented modelling is the process of preparing and designing what the model's code will actually look like. During the construction or programming phase, the modelling techniques are implemented by using a language that supports the object-oriented programming model.
- OOM consists of progressively developing object representation through three phases: analysis, design, and implementation.
- During the initial stages of development, the model developed is abstract because the external details of the system are the central focus. The model becomes more and more detailed as it evolves, while the central focus shifts toward understanding how the system will be constructed and how it should function.

23) Why and when object oriented modelling is used?

Object model introduces several new elements which are advantageous over traditional method of structured programming. The significant benefits are:

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilent to change.
- Object model appears to the working of human cognition, many people who have no idea how a computer works find the idea of object oriented systems quite natural.
- Some of the applications of the object model are as follows: - Air traffic control - Animation - Business or insurance software - Business Data Processing - CAD - Databases - Expert Systems - Office Automation - Robotics - Telecommunication - Telemetry System etc.

24) Differentiate between AOP and OOP

The objective of Agent Oriented (AO) Technology is to build systems applicable to real world that can observe and act on changes in the environment. Such systems must be able to behave rationally and autonomously in completion of their designated tasks. AO technology is an approach for building complex real time distributed applications. This technology is built on belief that a computer system must be designed to exhibit rational goal directed behavior similar to that of a human being.

In AOP objects known as agents interact to achieve individual goals. Agents can be autonomous entities, deciding their next step without the interference of a user, or they can be controllable, serving as mediatory between user and another agent. AOP programming is performed at abstract level. The Agent-Oriented Software Engineering (AOSE) is being described as a new paradigm for the research field of Software Engineering. But in order to become a new paradigm for the software industry, robust and easy-to-use methodologies and tools have to be developed

Comparison between OOP and AOP

	OOP	AOP
Basic units	Object	Agent
Parameters defining state of basic units	Unconstrained	Beliefs, commitments, capabilities, choices
Process of computation	Message passing and response methods	Message passing and response methods
Types of message	Unconstraint	Inform, request, offer, promise, decline

Constraints or methods	None	Honesty, consistency

Basic unit

The points listed below shows differentiators between agents and objects:

- Agents may communicate using an Agent Communication Language ACL or ICL, whereas objects communicate via fixed method interfaces.
- Agents have the quality of volition. That is, using AI techniques, intelligent agents are able to judge their results, and then modify their behavior (and thus their own internal structure) to improve their perceived fitness.
- Objects are abstractions of things like invoices. Agents are abstractions of intelligent beings -- they are essentially anthropomorphic. Note that this does not mean that agents are intelligent in the human sense, only that they are modeled after an anthropomorphic architecture, with beliefs, desires, etc.

Parameters defining state of basic unit

AOP agents contain beliefs, commitments, choices, and the like and communicate with each other via a constrained set of speech type acts such as inform, request, promise, decline the state of the agent is called its mental state.

Process of computation

- An object's message may request only one operation, and that operation may only b e requested via a message formatted in a very exacting way. The OO message broker has the job of matching each message to exactly one method invocation for exactly one object.
- Agent-based communication can also use the method invocation of OO. However, the demands that many agent applications place on message content are richer than those commonly used by object technology. While agent communication languages (ACL) are formal and unambiguous, their format and content vary greatly. In short, an agent message could consist of a character string whose form can vary yet obeys a formal syntax, while the conventional OO method must contain parameters whose number and sequence are fixed..

Type of message

- Agents are commonly regarded as autonomous entities, because they can watch out for their own set of internal responsibilities. Furthermore, agents are interactive entities that are capable of using rich forms of messages. These messages can support method invocation—as well as informing the agents of particular events, asking something of the agent, or receiving a response to an earlier query. Lastly, because agents are autonomous they can initiate interaction and respond to a message in any way they choose. In other words, agents can be thought of as objects that can say —No‖—as well as —Go.‖
- Due to the interactive and autonomous nature of agents, little or no integration is required to physically launch an application.
- Objects, on the other hand, are conventionally passive—with their methods being invoked under a caller's thread of control. The term autonomy barely applies to an entity whose invocation depends solely on other components in the system.

Constrain on method

- Usually, object classes are designed to be predictable in order to facilitate buying and selling reusable components. Agents are commonly designed to determine their behavior based on individual goals and states, as well as the states of ongoing conversations with other agents. While OO implementations can be developed to include nondeterministic behavior, this is common in agent-based thinking.
- Agent behaviour can also be unpredictable because of the following. First, an agent's knowledge can be represented in a manner that is not easily translated into a set of attributes. Even if an agent's state were publicly available, it may be difficult to decipher or understand. Second, the requested behaviors that an agent performs may not even be known within an active system. This is a clear distinction from object systems, because current OO languages only let you ask an object what interfaces it supports.
- Since the programmer needs to have some idea what interface to ask for, this makes coding difficult.
- In OO, there is no provision in current languages for an object to "advertise" its interfaces. In contrast, an agent can employ other mechanisms, such as publish/subscribe, protocol registration, —yellow page‖ and —white page‖ directories. Another common mechanism provides specialized broker agents to which other agents can make themselves known for various purposes but are otherwise unlisted to the rest of the agent population. Lastly, the underlying agent communication model is usually asynchronous. This means that there is no predefined flow of control from one agent to another. An agent may autonomously initiate internal or external behavior at anytime, not just when it is sent a message [3]. There is no predefined flow of control from one agent to another. An agent may autonomously initiate internal or external behavior at anytime, not just when it is sent a message.

25. Explain the various steps of domain and application analysis with example

Domain analysis is the process of identifying, collecting, organizing, analyzing and representing a domain model from the study of existing systems, underlying theory, emerging technology and development histories within the domain of interest.

Domain analysis can be defined as: "a process by which information used in developing software systems is identified, captured and organized with the purpose of making it reusable when creating new systems" [Prieto-Diaz90]. During software development, different information is produced, and the software product delivered is only a part of this heap of data. One of the main goals of domain analysis is to analyze all these information aiming to exploit and reuse most of them in future software development projects. Following this view, domain analysis fosters software reuse in the sense that it supports the identification and definition of information and components that can be reused in applications and in contexts different from the ones for which they were originally conceived.

Domain analysis is an activity occurring prior to system analysis. It aims to identify features common to a domain of applications, selecting and abstracting the objects and operations that characterize those features. The first definition of domain analysis has been introduced by Neighbors as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain" [Neighbors81]. In his work, Neighbors draws an analogy between domain analysis and system analysis: system analysis is concerned with the specific aspects in a particular system, while domain analysis is concerned with objects and operations in a set of systems in a defined application area. The generalization of the systems in an application domain aims to define domain models that transcend specific applications. Generally speaking, domain analysis should support extraction, organization, analysis and abstraction, understanding, representation and modeling of reusable information and assets from the software process [Prieto-Diaz90]. In these last years these concepts have been exploited and different methodologies for domain analysis have been proposed. Readers interested in a more comprehensive survey on Domain Analysis should consult [Arango 93].

Considering the activities which are shared by the different Domain Analysis methods, it is possible to define a general model for the domain analysis process which can help in comparing the different methodologies (in [Arango93] it is presented an extended discussion on a Common Domain Analysis Process). Along the lines of [Arango 93], the general model can be structured into four main phases, each phase constituted by different activities:

1. *Domain Characterization and project planning*: the first step of every domain analysis method is a preparation activity, which aims to collect the minimum information concerning the problem that allows to decide if it is worth to deal with it and try to solve it, or if it is not feasible to go on with the process. The main activities that have to be carried out during this phase are:
    a. Business Analysis;
    b. Feasibility Analysis;
    c. Domain Description;
    d. Project planning and resource allocation.
2. *Data Analysis*: the information necessary to the analysis are collected and organized, then the analysis exploits domain commonalties and variations. The main activities prescribed are:
    a. Data Organization
    b. Data Exploitation
3. *Domain Modeling*: the purpose of the modeling phase is to complete the previous analysis step building suitable domain models of the domain, specifically:
    a. Modeling commonalties aspects in the domain;
    b. Refining domain models encapsulating variations possibilities;
    c. Defining frameworks and general architecture for the domain;
    d. Describing the rationale beneath domain models and tracing technical issue and relative decisions made in the analysis and modeling process.

    This phase can be considered the core activity aiming to produce reusable assets, such as components, framework and architectures. The difference between domain modeling and system modeling lies in the target chosen: for system modeling it is the specific software system that has to be built, while for domain analysis it is a class of similar systems in a specific application domain.

4. *Evaluation*: its purpose is to verify the results of each step of the domain analysis process, identifying possible errors done in building the model, and to validate the results against requirements and user expectations.

25) Explain OOS development process.

This is also known as Classic Life Cycle Model (or) Linear Sequential Model (or) Waterfall Method. This model has the following activities.

- System/Information Engineering and Modeling

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

- Software Requirement Analysis

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system. By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.. The requirement gathering process is intensified and focused

specially on software. To understand the nature of the program(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behavior, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

- System Analysis and Design

In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc.. are all defined in this phase. A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

- Code Generation

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like compilers, interpreters, debuggers etc.. are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

- Testing

Once the code is generated, the software program testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

- Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

26) Define prototyping and its types.

- A prototype isn't just a part of the product design, it is one of the most integral parts without which future steps of the startup process are nothing but useless.
- prototype is a rudimentary working sample, model, mock-up or just a simulation of the actual product based on which the other forms (MVP, final product, and variations) are developed.
- The main motive behind prototyping is to validate the design of the actual product. Sometimes, creating a prototype is called materialization as it is the first step of transforming the virtual or conceptualized design into the real physical form.
- **Prototyping** refers to an initial stage of a software release in which developmental evolution and product fixes may occur before a bigger release is initiated. These kinds of activities can also sometimes be called a beta phase or beta testing, where an initial project gets evaluated by a smaller class of users before full development.

Categories of Prototype

i. **A horizontal prototype** is a simulation of the interface (that is, it has the entire user interface that will be in the full-featured interface that will be in the full-featured system) but contains no functionality. This has the advantages of being very quick to implement, providing a good overall feel of the system, and allowing users to evaluate the interface on the basis of their normal, expected perception of the system.

ii. **A vertical prototype** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth. In practice, prototypes are a hybrid between horizontal and vertical. The major portions of the interface are established so the user can get the feel of the system, and features having a high degree of risk are prototyped with much more functionality

iii. **An analysis prototype** is an aid for exploring the problem domain. This class of prototype is used to inform, the user and demonstrate the proof of a concept. It is not used as the basis of development, It is not used as the basis of development, however, and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code.

iv. **A domain prototype** is an aid for the incremental development of the ultimate software solution. It often is used as a tool for the staged delivery of subsystems to the users or other delivery of subsystems to the users or other members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product.

From a usability perspective, the prototypes can be categorized into:

i. **Throwaway prototype**: Throwaway prototype refers to the models which are eventually discarded or thrown away rather than becoming a part of the actual product. These products are only used to represent what an actual product can do. Throwaway prototypes are also called close ended prototypes

ii. **Evolutionary Prototype**: Evolutionary prototyping uses a different approach than throwaway prototyping and involves building a basic but robust prototype in a manner which can further be improved and built upon to form an actual saleable product. This avoids wastage of resources.

Prototype Examples

Prototypes come in many types and shapes. It all depends on the reason for what a prototype is created. While some prototypes are developed just to represent or mimic the functioning or the look of the product (paper prototypes, HTML prototypes, etc.) to investors, some include showing a miniature version (3D print, single version of the lot, etc.) of the product with full or partial functionality.

27) UML Diagrams

The underlying premise of UML is that no one diagram can capture the different elements of a system in its entirety. Hence, UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system. The nine UML diagrams are:

- **Use case diagram:** The use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The use case diagram shows which actors interact with each use case.
- **Class diagram:** The class diagram is used to refine the use case diagram and define a detailed design of the system. The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. The relationship or association between the classes can be either an "is-a" or "has-a" relationship. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed "methods" of the class. Apart from this, each class may have certain "attributes" that uniquely identify the class.
- **Object diagram:** The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.
- **State diagram:** A state diagram, as the name suggests, represents the different states that objects in the system undergo during their life cycle. Objects in the system change states in response to events. In addition to this, a state diagram also captures the transition of the object's state from an initial state to a final state in response to events affecting the system.
- **Activity diagram:** The process flows in the system are captured in the activity diagram. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions.
- **Sequence diagram:** A sequence diagram represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing "messages".
- **Collaboration diagram:** A collaboration diagram groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects.
- **Component diagram:** The component diagram represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated. A component diagram depicts the components culled after the system has undergone the development or construction phase.
- **Deployment diagram:** The deployment diagram captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed.