Q.1

What is Object Orientation? Explain the concept of class, objects, instance, generalization, and associations .

Ans :--

In the past, information systems used to be defined primarily by their functionality: data and functions were kept separate and linked together by means of input and output relations.

The object-oriented approach, however, focuses on objects that represent abstract or concrete things of the real world. These objects are first defined by their character and their properties which are represented by their internal structure and their attributes (data). The behaviour of these objects is described by methods (functionality).

Objects

Objects are instances of classes. They contain data and provides services. The data forms the attributes of the object. The services are known as methods (also known as operations or functions). Typically, methods operate on private data (the attributes, or state of the object), which is only visible to the methods of the object. Thus the attributes of an object cannot be changed directly by the user, but only by the methods of the object. This guarantees the internal consistency of the object.

Classes

Classes describe objects. From a technical point of view, objects are runtime instances of a class. In theory, you can create any number of objects based on a single class. Each instance (object) of a class has a unique identity and its own set of values for its attributes.

Instance:-: a composite data type, or object, created based on
the rules

set forth in the class definitionThis break between class and instance is not new, it is just that before objects, all data classes were hard coded into the parser and you could just make use of them while creating variables that were instances of those classes. Someone, somewhere, had to write the code to define the integer data type as being a numeric value with no fractional component. Whenever you declare an integer variable, you make use of this definition to create, or instantiate, an integer. Fortunately for us, it all happens behind the scenes.

Association:-

Association is a (***a***) relationship between two classes. It allows one object instance to cause another to perform an action on its behalf. Association is the more general term that define the relationship between two classes, where as the aggregation and composition are relatively special.

Generalization:-

A **generalization** (or **generalisation**) of a concept is an extension of the concept to less-specific criteria. It is a foundational element of logic and human reasoning.[*citation needed*] Generalizations posit the existence of a domain or set of elements, as well as one or more common characteristics shared by those elements. As such, they are the essential basis of all valid deductive inferences. The process of verification is necessary to determine whether a generalization holds true for any given situation.

Q:-2What is UML? Briefly explaindifferent UML diagrams

.Ans:---

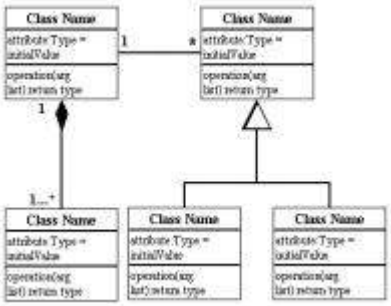
UML stands for Unified Modeling Language. This object-oriented system of notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists fused their respective technologies into a single, standardized model. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs.

Types of UML Diagrams

UML defines nine types of diagrams: class (package), object, use case, sequence, collaboration, statechart, activity, component, and deployment.

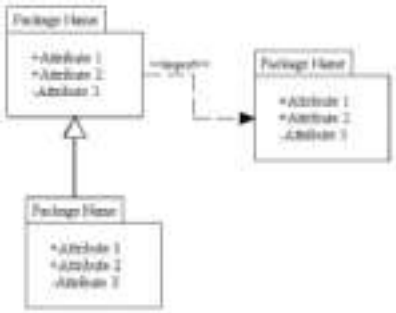
Class Diagrams

Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.


Class Name
attribute:Type = initialValue
operation(arg list):return type
1
*
Class Name
attribute:Type = initialValue
operation(arg list):return type
1
1...*
Class Name
attribute:Type = initialValue
operation(arg list):return type
Class Name
attribute:Type = initialValue
operation(arg list):return type
Class Name
attribute:Type = initialValue
operation(arg list):return type

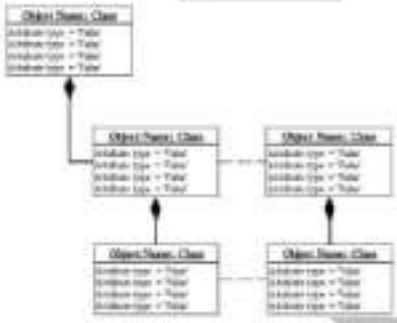
Package Diagrams

Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



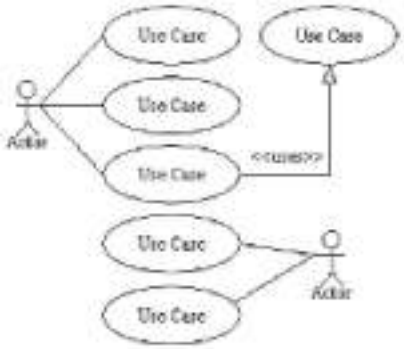
Object Diagrams

Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.



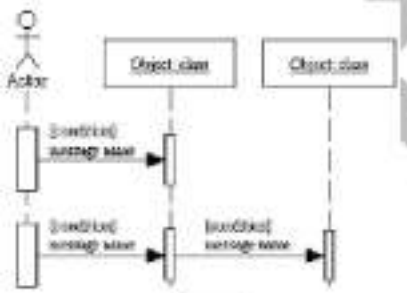
Use Case Diagrams

Use case diagrams model the functionality of system using actors and use cases.



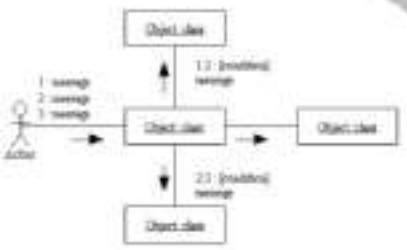
Sequence Diagrams

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.



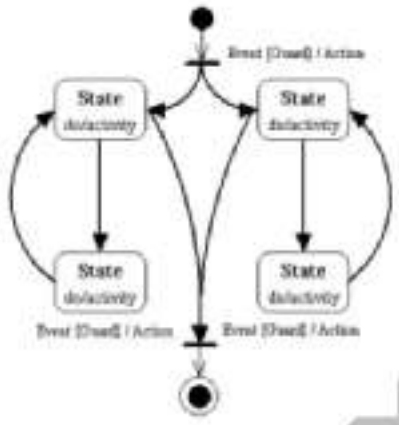
Collaboration Diagrams

Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.



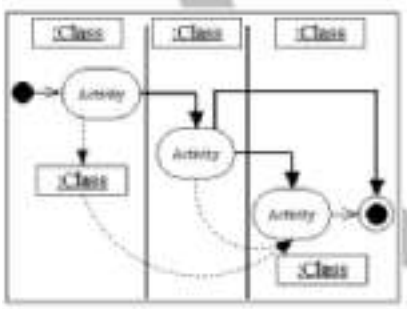
Statechart Diagrams

Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.



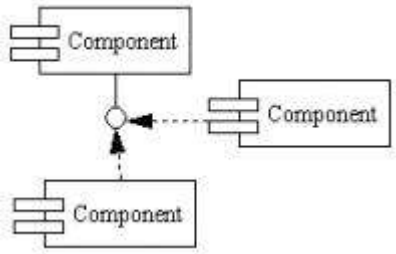
Activity Diagrams

Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.



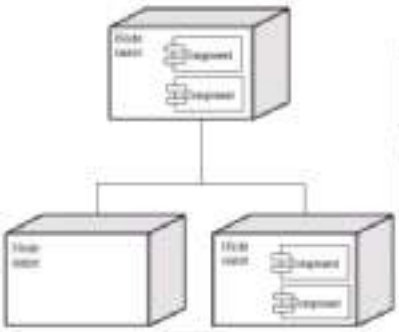
Component Diagrams

Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.



Deployment Diagrams

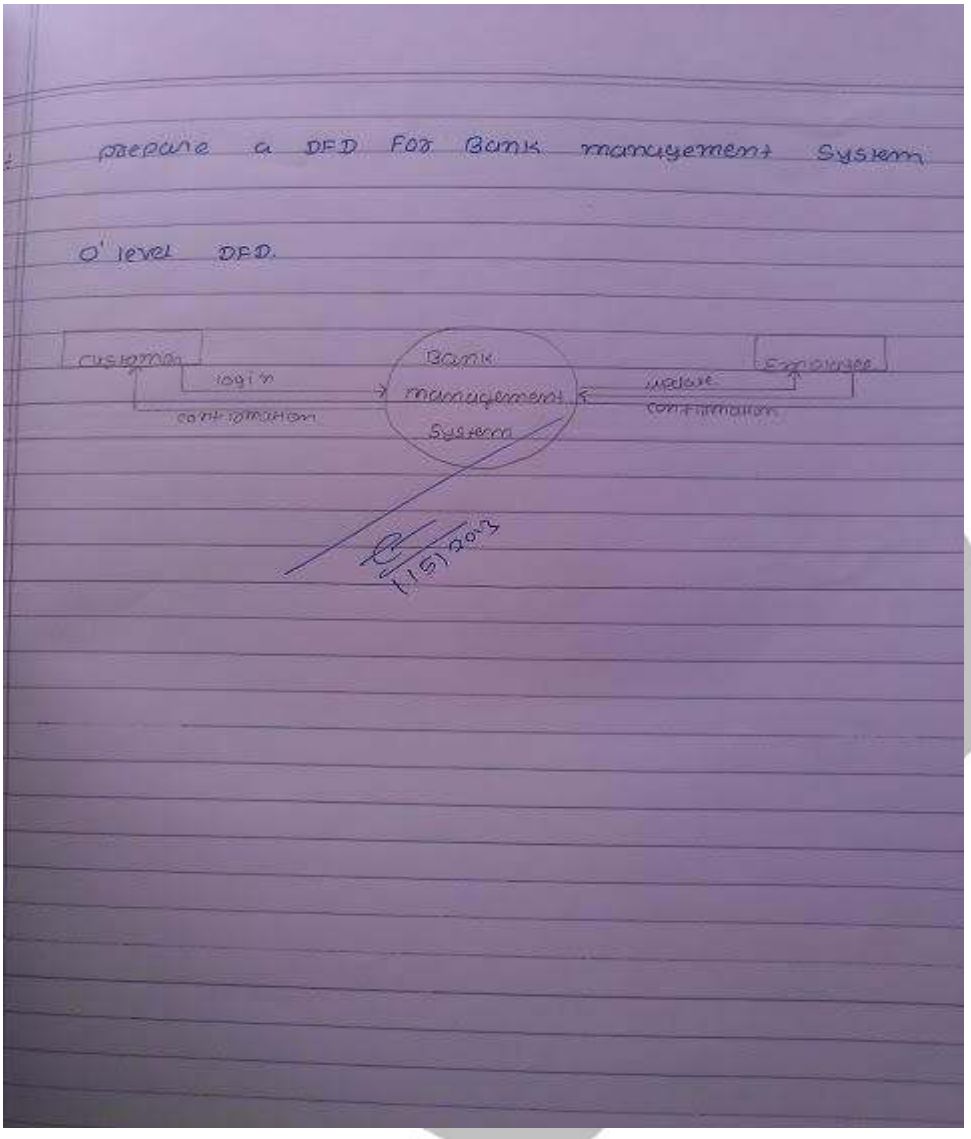
Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

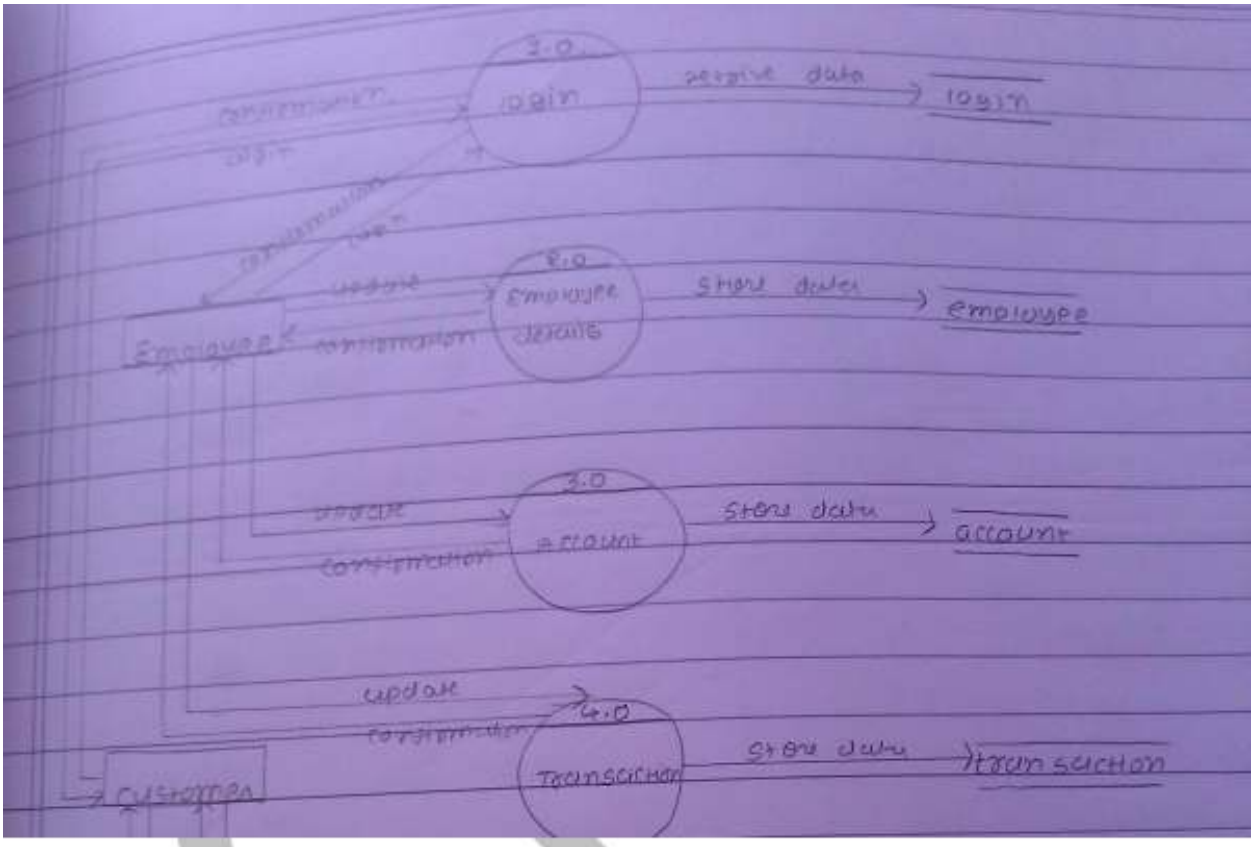


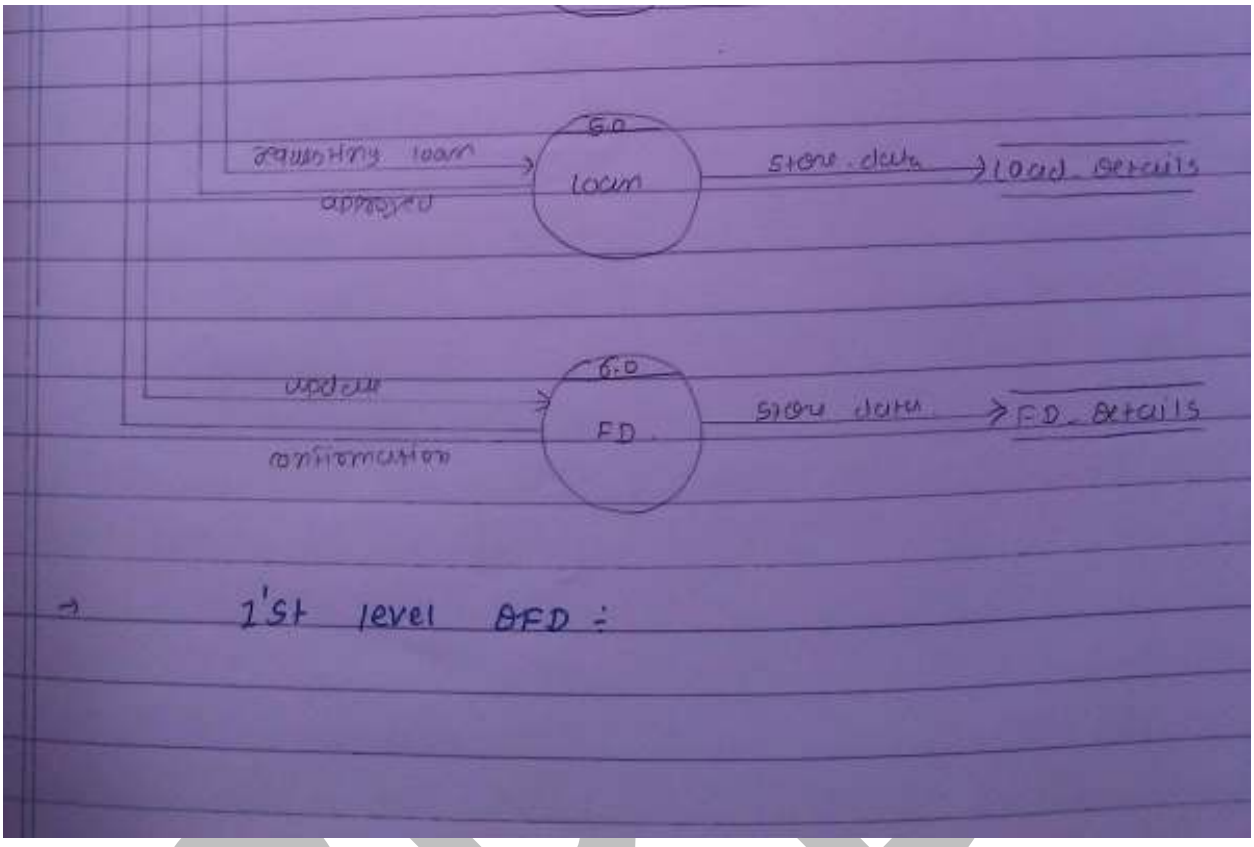
Q.3:--

Draw a DFD for Bank Accounts Management System

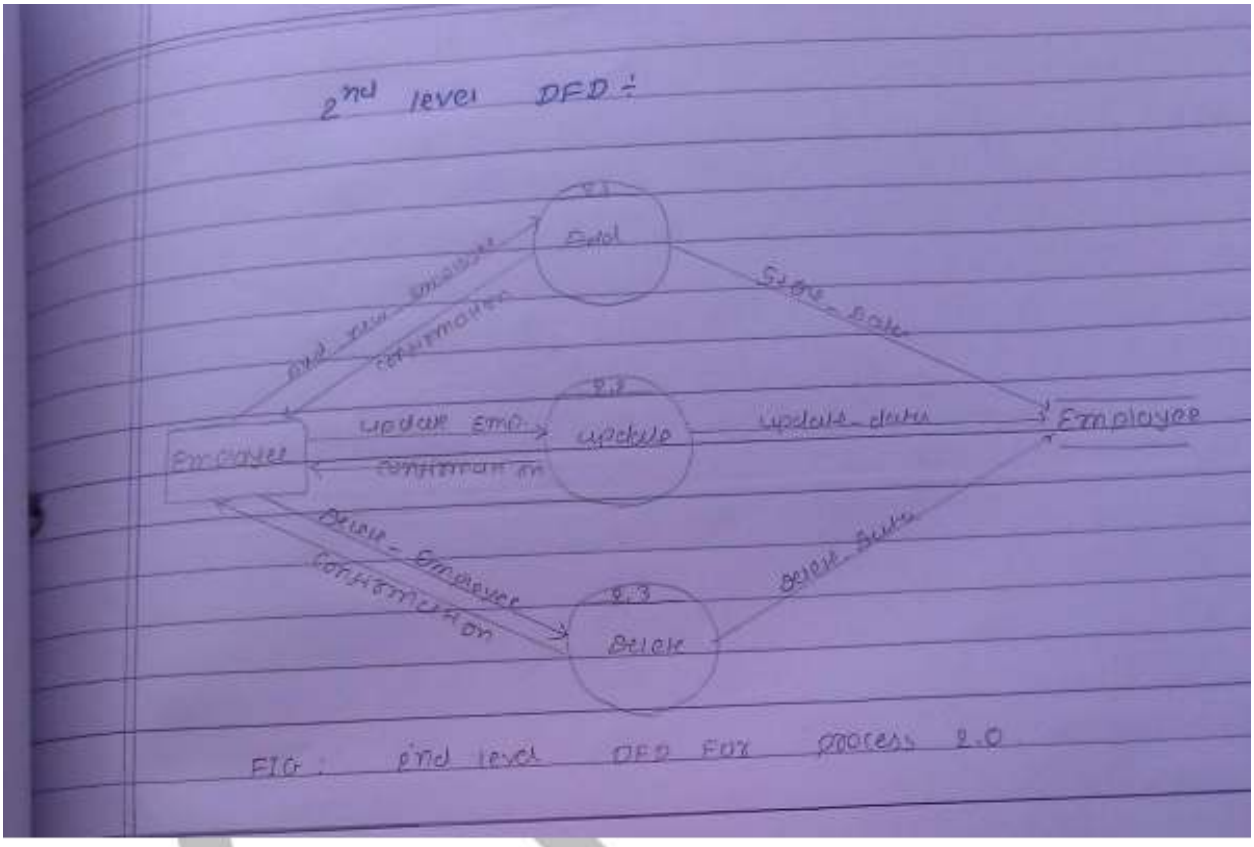
Ans:--







→ 1'st level DFD :



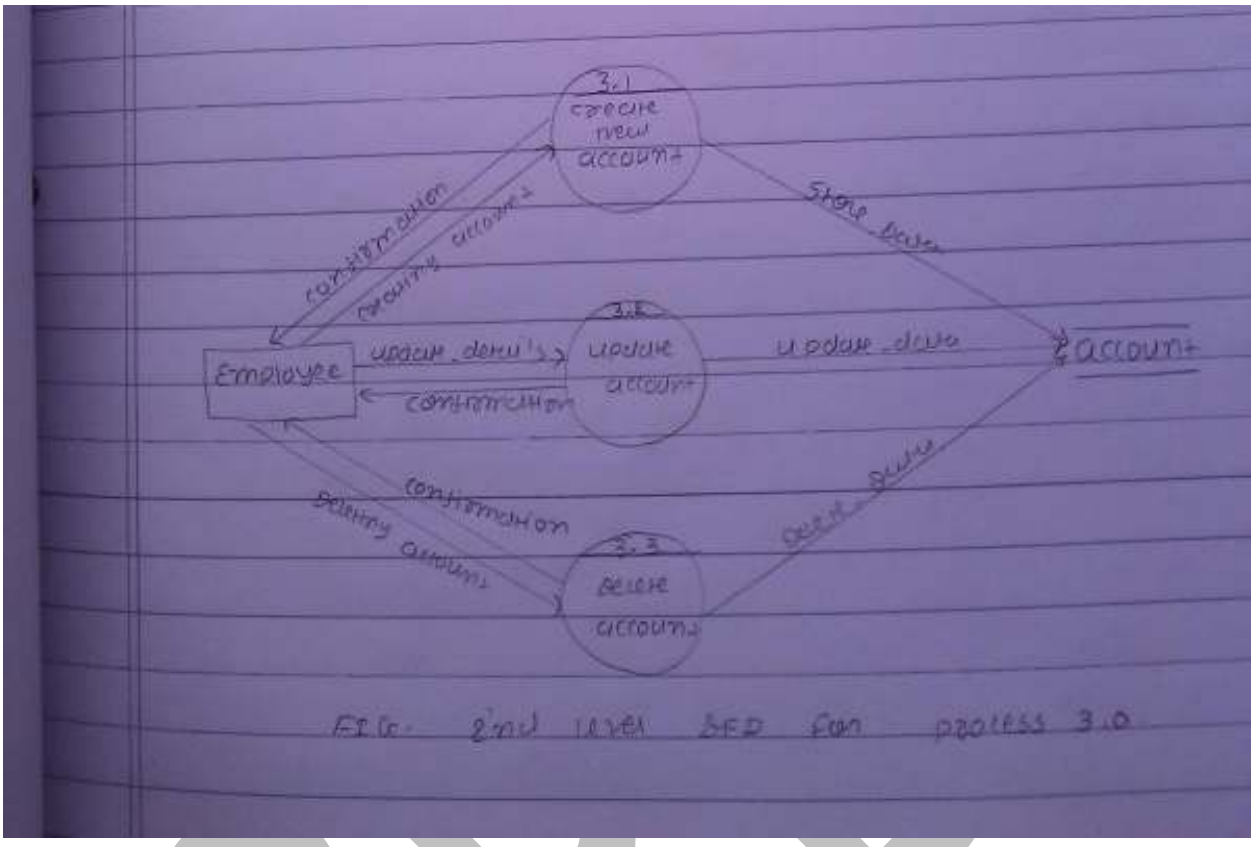
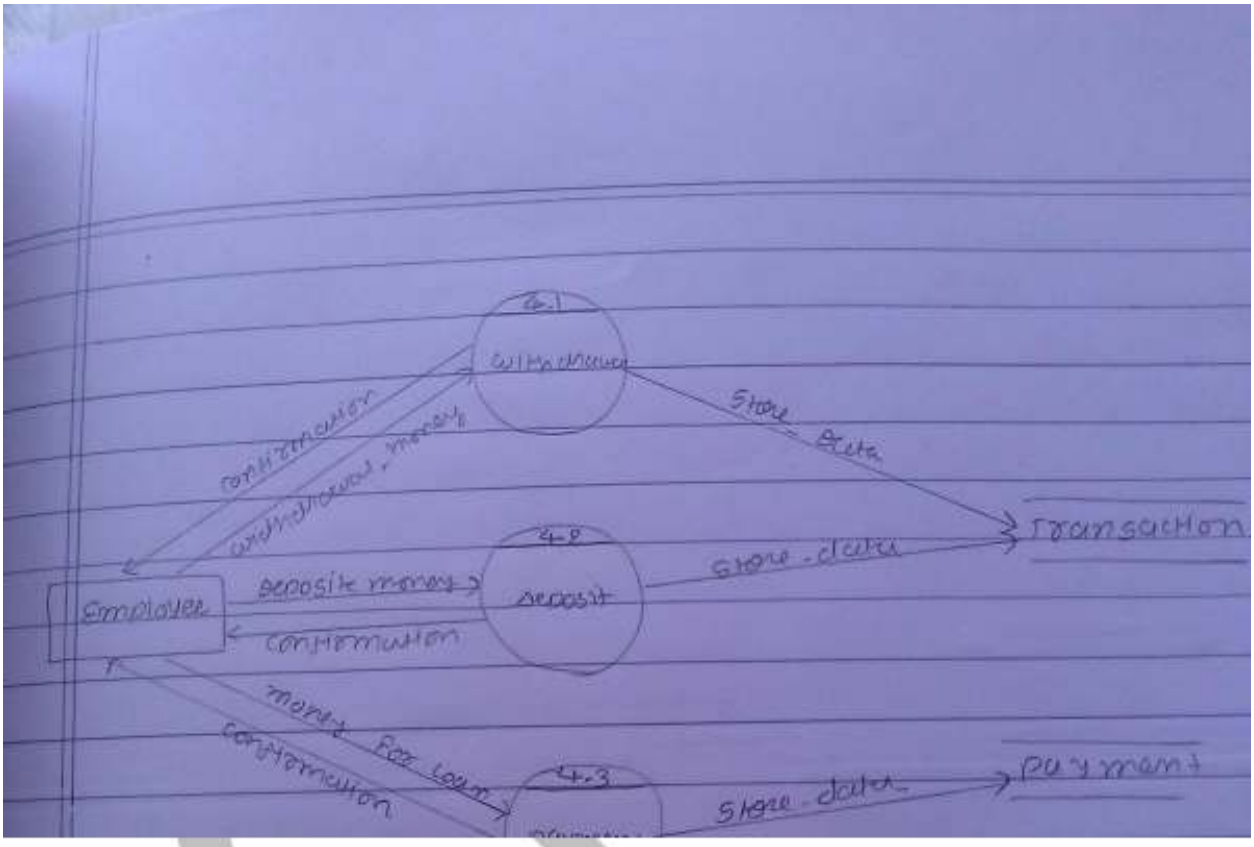
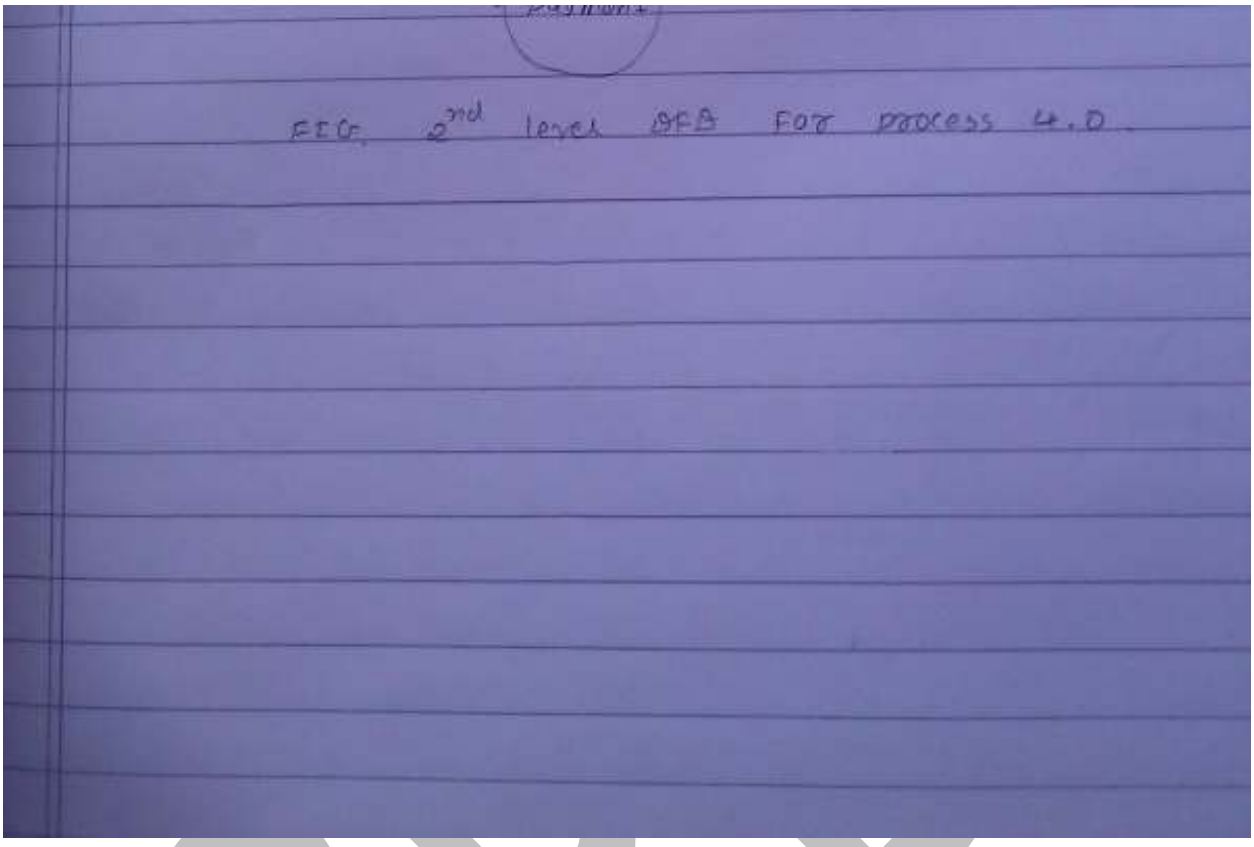


Fig: 2nd level DFD for process 3.0




FEC 2nd level OFB for process 4.0

Q 4:--

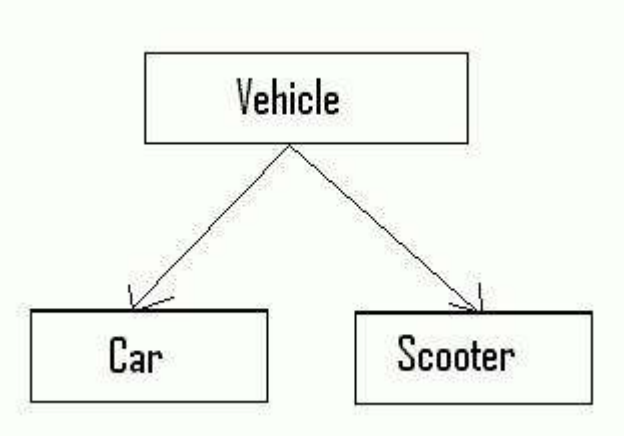
What is specialization ? Explain how it is different from generalisation with the help of an example ?

Ans:--

Specialization is a property of inheritance used for creating specialized attributes, methods that only apply to objects of that class. Specialized classes are subclasses placed below super classes. Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass.

Shared characteristics can be attributes, associations, or methods. Generalized classes are generally parent classes of specialized classes. In other words generalization is bottom up approach whereas specialization is Top to bottom approach

Example:



In Figure , the classes car and scooter partially share the same attributes. From a domain perspective, the two classes are also very similar. During generalization, the shared characteristics are combined and used to create a new superclass vehicle. car and scooter become subclasses of the class vehicle.

Similarly, the class vehicle has the a special attribute, which is needed only for car, but not for

scooter. Additionally there is a special attribute, which is needed only for scooter Obviously, here two similar but different domain concepts are combined into one class. Through specialization the two special cases vehicles are formed The epecial attribute is placed where it belongs—in car /scooter.

Q-5. What is Object Oriented modeling? Explain different types of model and their requirement in system design?

Ans:---

The Object Modeling Technique (OMT) is an object-oriented analysis, design, and implementation methodology that focuses creating a model of objects from the real world and then using this model to develop object-oriented software. OMT was developed by James Rumbaugh, et. al., at General Electric's Research and Development Center. The comments in this review are based on their book,Object-oriented Modeling and Design [Rumbaugh-1991]..

The OMT methodology uses three kinds of models to describe a system:

1.Object model- describes the objects in the system and their relationships;
2.Dynamic model- describes the interactions among objects in the system; and
3.Functional model- describes the data transformations of the system.

Object Model ;-

Object;-An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. All objects have identity and are distinguishable.

Class:- A class describes a group of objects with similar properties (attributes), common behavior(operations), common relationships to other objects, and common semantics

Attribute:-An attribute is (represents) a data value which is held by the objects in a class. An attribute should be a pure data value, not an object. Pure data values do not have identity Operation:-An operation is a function or transformation that may be applied to or by objects in a class. The same operation may be defined for several different classes. However, the signature (i.e. output type and formal parameter list) must be the same

Method:-A method is the implementation of an operation for a class. The choice of which method is invoked it solely determined by the target object. The parameters are not a factor in determining which method is selected

Dynamic model-

Events:-

An event is something that happens at a point in time. It has no duration. Information can be passed on an event An event can cause a change in state. This is called an event transition. Event transitions can have conditions and actions associated with them.

States:

A state is an abstraction of the attribute values and links of an object A state corresponds to the interval between two events received by an object

A state diagram relates states and events State diagrams can be nested. A single state can have subdiagrams associated with it A nested state diagram is actually a form of generalization on states .

Functional model components

Action: An action is a type of operation

An action is a transformation that has side effects on the target object or other objects in the system reachable from the target object. An action has no duration in time; it is logically instantaneous

Activity: An activity is a type of operation

An activity is an operation to or by an object that has duration in time

Actor:

An actor is an active object that drives the data flow graph by producing or consuming values

Question 6 :

(a)

Explain the two strategies to implement state charts with the help of an example of each. ?

Ans:--

Events

In the most general terms, an **event** is something that happens that affects the system. Strictly speaking, in the UML specification,[1] the term event refers to the type of occurrence rather than to any concrete instance of that occurrence. For example, Keystroke is an event for the keyboard, but each press of a key is not an event but a concrete instance of the Keystroke event. Another event of interest for the keyboard might be Power-on, but turning the power on tomorrow at 10:05:36 will be just an instance of the Power-on event.

An event can have associated **parameters**, allowing the event instance to convey not only the occurrence of some interesting incident but also quantitative information regarding that occurrence. For example, the Keystroke event generated by pressing a key on a computer keyboard has associated parameters that convey the character scan code as well as the status of the Shift, Ctrl, and Alt keys.

An event instance outlives the instantaneous occurrence that generated it and might convey this occurrence to one or more state machines. Once generated, the event instance goes through a processing life cycle that can consist of up to three stages. First, the event instance is **received** when it is accepted and waiting for processing (e.g., it is placed on the event queue). Later, the event instance is **dispatched** to the state machine, at which point it becomes the current event. Finally, it is **consumed** when the state machine finishes processing the event instance. A consumed event instance is no longer available for processing.

States

A **state** captures the relevant aspects of the system's history very efficiently. For example, when you strike a key on a keyboard, the character code generated will be either an uppercase or a lowercase character, depending on whether the Caps Lock is active. Therefore, the keyboard's behavior can be divided into two states: the "default" state and the "caps_locked" state. (Most keyboards include an LED that indicates that the keyboard is in the "caps_locked" state.) The behavior of a keyboard depends only on certain aspects of its history, namely whether the Caps Lock key has been pressed, but not, for example, on how many and exactly which other keys have been pressed previously. A state can abstract away all possible (but irrelevant) event sequences and capture only the relevant ones.

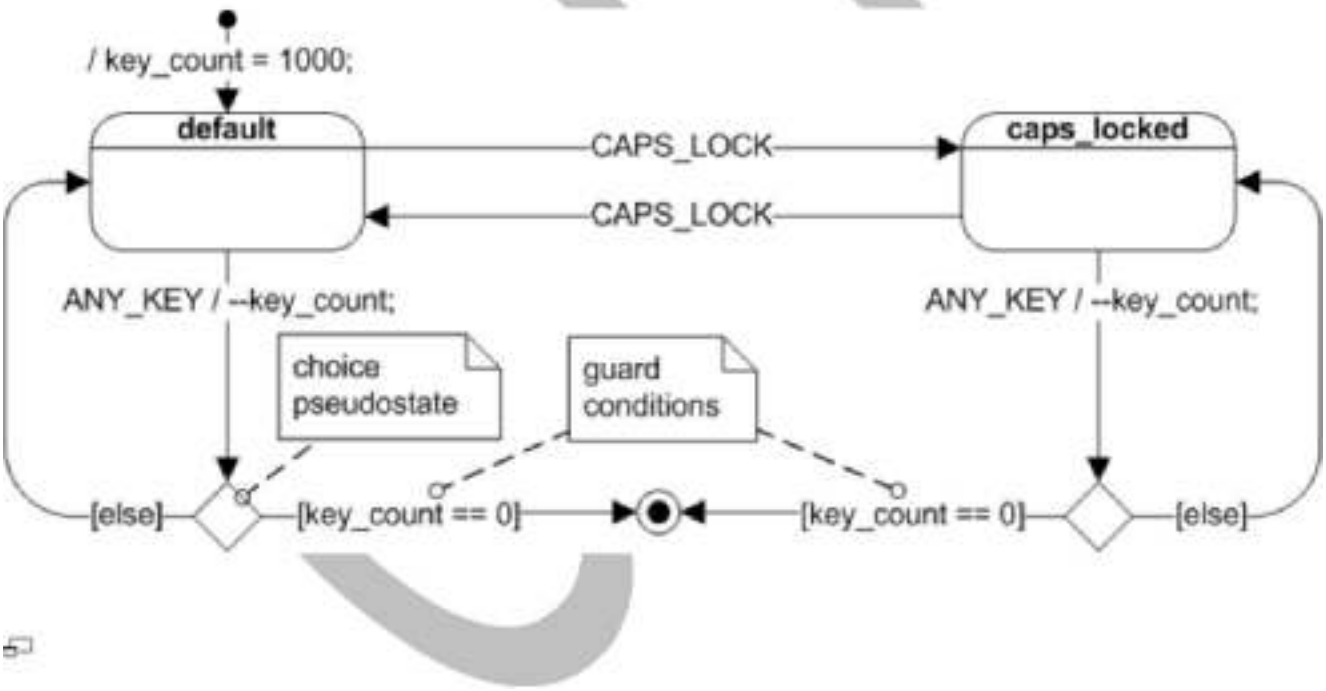
Extended states

One possible interpretation of state for software systems is that each state represents one distinct set of valid values of the whole program memory. Even for simple programs with only a few

elementary variables, this interpretation leads to an astronomical number of states. For example, a single 32-bit integer could contribute to over 4 billion different states. Clearly, this

interpretation is not practical, so program variables are commonly dissociated from states. Rather, the complete condition of the system (called the **extended state**) is the combination of a qualitative aspect (the state) and the quantitative aspects (the extended state variables). In this interpretation, a change of variable does not always imply a change of the qualitative aspects of the system behavior and therefore does not lead to a change of state.[5]

the keyboard breaks down and enters the final state. To model this behavior in a state machine without memory, you would need to introduce 1,000 states (e.g., pressing a key in state stroke123 would lead to state stroke124, and so on), which is clearly an impractical proposition. Alternatively, you could construct an extended state machine with a key_count down-counter variable. The counter would be initialized to 1,000 and decremented by every keystroke without changing state. When the counter reached zero, the state machine would enter the final state.



The obvious advantage of extended state machines is flexibility. For example, extending the lifespan of the "cheap keyboard" from 1,000 to 10,000 keystrokes would not complicate the extended state machine at all. The only modification required would be changing the initialization value of the `key_count` down-counter in the initial transition.

This flexibility of extended state machines comes with a price, however, because of the complex coupling between the "qualitative" and the "quantitative" aspects of the extended state. The coupling occurs through the guard conditions attached to transitions,

Q6.(b)

Draw a sequence diagram for sending a audio and a video file to your fried using email to your friend. ?

Ans:--

Question 7 :

(a) What is Bi-directional Implementation? Explain advantages of Bi-directional Implementation with the help of an example.

Ans:---- A bi-directional association is indicated by a solid line between the two classes. At either end of the

line, you place a role name and a multiplicity value. Figure above shows that thestudent is associated with a course, and the Flight class knows about this association. . The multiplicity value next to the student class of 0..* means that when an instance of a student exists, it can either have one instance of a course associated with it or no student associated with it (i.e., maybe a plane has not yet been assigned) or one or many students associated. similarly multiplicity value next to course class is * (many) which means when a one instance of student is associated with more than one courses(many) Advantage of Bi-directional Implementation 1. Independent of Classes 2. Useful for existing predefined classes which are not modified.

Q:7. (b) How do you identify concurrency? Explain the important issues related to concurrency.?

Ans :-

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as concurrent software

. Concurrency is the execution of two or more independent, interacting programs over the same period of time; their execution can be interleaved or even simultaneous. Concurrency is used in many kinds of systems, both

small and large.
EXAMPLE 1
The typical home computer is full of examples. In separate windows, a user runs a web browser, a text editor, and a music player all at once. The operating system interacts with each of them. Almost unnoticed, the clock and virus scanner are also running. The operating system waits for the user to ask more programs to start, while also handling underlying tasks such as resolving what information from the Internet goes to which program. EXAMPLE 2 Using a web-based airline reservation system, Joe and Sue each want to buy a ticket. Ea ch has a separate computer, and thus a separate web browser. Their two web browsers plus the airline's webserver together comprise the concurrent program. As this example illustrates, the termconcurrent system might be more appropriate. However, for verification purposes, we will view them as a single, distributed entity to be modeled.A concurrent system may be implemented via processes and/or threads. Although details can vary upon platform, the fundamental difference is that processes have separate address spaces, whereas threads share address spaces. We will follow the common theoretical practice and ignore this distinction, using shared variables when desired, and using the two terms interchangeably. We view these threads as executing relatively independently. However, since they are acting together towards some goal, they must need to communicate and coordinate. The two most common communication techniques in processes and threads are through, respectively, message passing and shared variables

. In order to communicate at the right times, they must synchronize , together arriving at agreed -upon control points. Often, one or more threads block, or stop and wait for some external event. In this module, we will use only shared variables, although the tools we use also allow message passing. Even though we have multiple flows of control, that doesn't imply we need multiple processors. Concurrent programs may be executed on a single processor by interleaving their control flows. In order to understand what happens in a concurrent program over time, we must understand how the individual operations of the threads can interleave. This is independent of how many processesors we use, although it is convenient to think of only having one processor. To consider the possible interleavings, we need to know which actions are atomic, or indivisible. If an atomic operation begins, we know that it won't be interrupted by a context switch until it is done. We generally assume that each hardware machine instruction is atomic, but a programming language might guarantee that even more complicated operations are atomic as well.

DEFINITION 1:
State

A state captures all the current information in a program. This includes all local and global variables' values and each thread's current program counter. More generally, this includes all the memory and register contents.

SEE ALSO:
trace,state-space
DEFINITION 2:
Trace

A trace is the sequence of operations (and their data) performed in a particular run of a concurrent program. Equivalently, it is the sequence of states during a run —i.e., the collection of variable and program counter values.

SEE ALSO:

State ,ω-trace
DEFINITION 3:
Atomic

An atomic operation is indivisible. A context switch to another process cannot happen during this operation.

EXAMPLE:

Statement

-

level Atomicity Let's examine the possible interleavings of the code fragments for two threads. First,let's assume that each assignment statement is atomic.

1 thread0:

2 {
3 x=x+1
4 }
5 thread1:
6 {
7 x=x*2
8 }
(a)
(b)
Figure 1:
There are two possible traces, as either