IV/IV B.Tech (Regular) DEGREE EXAMINATION

Nov, 2019 **Common for CSE & IT**

Seventh Semester **Object Oriented Analysis and Design**

Time: Three hours **Maximum:** 60 Marks

Answer question No.1 compulsorily. (1*12=12 Marks)

Answer one question from each unit (4*12=48 Marks)

1.Answer all questions.

a. Define an Object?

A. Instance of class or variable of class.

(OR)

A single thing or concept, either in a model of an application domain or in a software system that can be represented as an encapsulation of state, behaviour and identity.

b. What is Coupling?

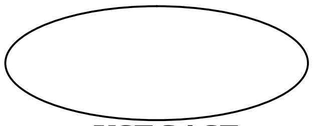
A. Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. (coupling is the indication of the relationships between modules)

c. What is the use of UML in object-oriented approach?

A.UML is a visual language that lets you to model processes, software, and systems to express the design of system architecture. It is a standard language for designing and documenting a system in an object-oriented manner that allow technical architects to communicate with developer.

d. State how use case is represented?

A. Use case diagrams are a way to capture the system's functionality and requirements in UML diagrams. A use case represents a distinct functionality of a system, a component, a package, or a class. An actor is an entity that initiates the use case from outside the scope of a use case.

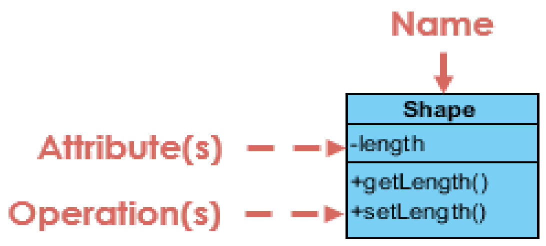


USECASE

e. How a class is represented in UML?

A. Class is represented by a rectangle having a subdivision of three compartments name,

attributes and operation.



f. Define generalization?

A. A **generalization** is a binary taxonomic (i.e. related to classification) directed relationship between a more general classifier (superclass) and a more specific classifier (subclass).

g. State analysis differs from design? .

A. Analysis is understanding the problem you're trying to solve. Design is figuring out how to organize the solution once you've done the analysis and you understand the problem.

h. Give the guide lines for naming classes?

A. Class names should be nouns in Upper case, with the first letter of every word capitalized. Use whole words — avoid acronyms and abbreviations.

i. What is aggregation?

A. Aggregation is special form of association that specifies a whole-part relationship between the **aggregate** (whole) and a component part composition.

j. What is meant by CRT cards?

A. Class-responsibility-collaboration (CRC) cards are a brainstorming tool used in the design of object-oriented software used to identify classes.

k. What is metaphor?

A. The idea that the user is carrying on a dialogue with the system is a *metaphor*.

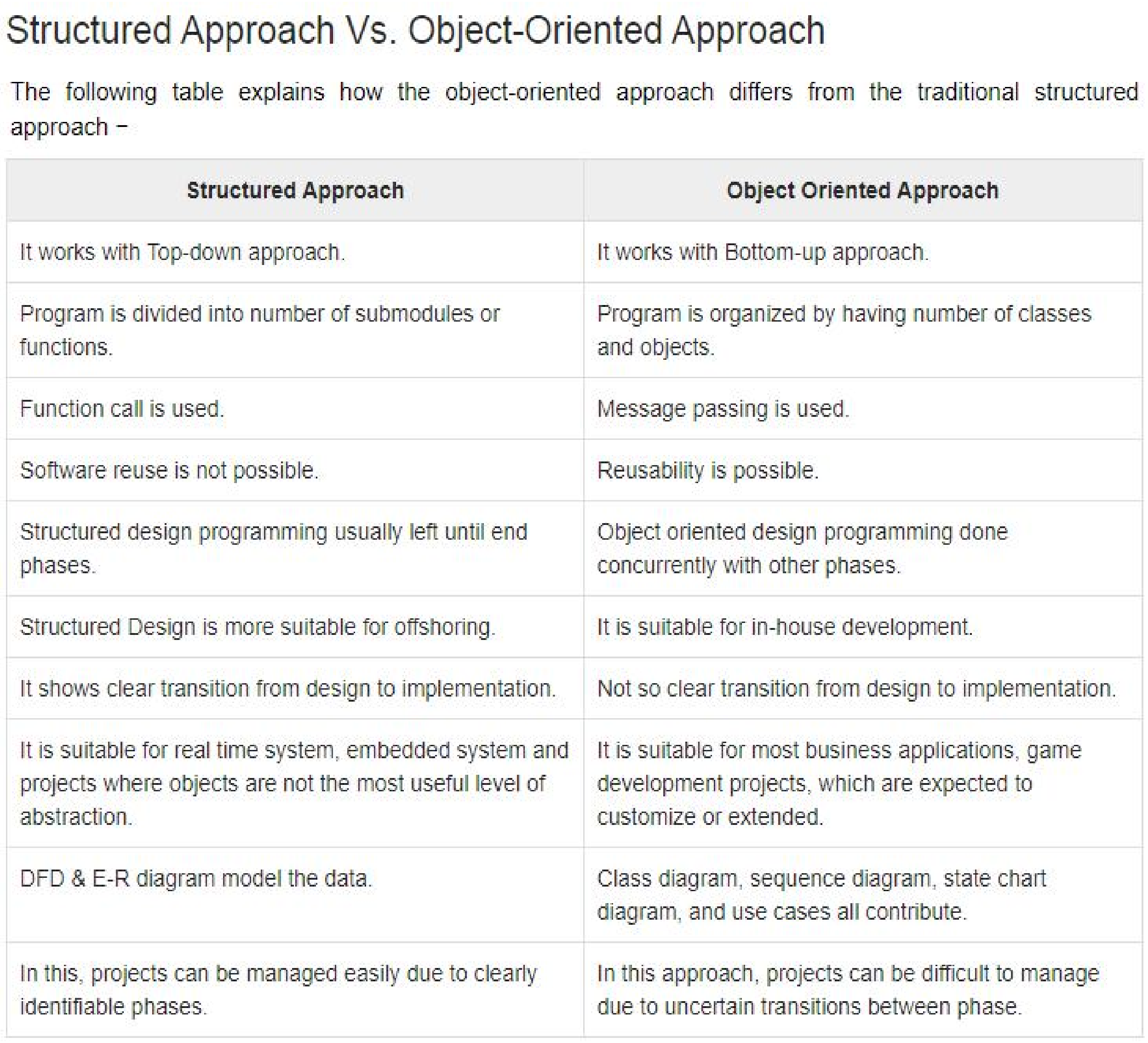
l. what is Reusability?

A. Reusability is the use of existing assets in some form within the software product development process.

UNIT-I

2.a. Compare the object oriented system development with structured approach? 6M

<u>Structured Approach ----3M Object Oriented Approaqch-----3M</u>



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.

2. b. Describe the elements of object model with example? 6M

<u>Any Six Elements can be considered------6M</u>

A. It is a way to develop a s/w by building self-contained modules or objects that can be easily replaced, modified and re-used.

1. Objects:

According to Coad and Yourdon object is define as follows. Object is an abstraction of something in a problem domain, reflecting the capabilities of the system to keep the information about it, interact with it, or both.

Abstraction in this context might be, a form of representation that includes only what is important or interesting from a view point.

Eg: a Map is an abstract representation; no map shows every detail of the territory

It covers. The intended purpose of the choice of which details to be given, or which to be suppress. Mean an Object represents only those features of a thing that are deemed relevant to the current purpose, and hides those features that are not relevant. According to Ram Baugh Object defined as a concept, abstraction, or thing with the boundaries and meaning for the problem at hand.

Objects serves for two purposes.

1. They promote understanding of the real world.

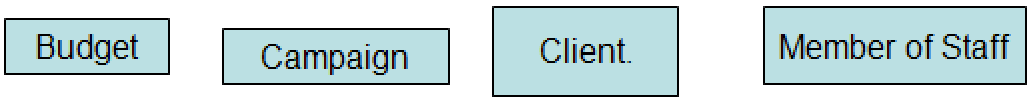
2. Provides a complete practical basis for computer implementation

2. Class and Instance:

An object represents a particular instance of a class. Objects that are sufficiently similar to each other are said to the same class. Instance is another word for a single object , but it also carries features of the class to which that object belongs : means every object is an instance of some class. So like an object , an Instance represents a single person, thing or concept in the application domain.

A Class is a abstract descriptor for a set of instances with certain logical similarities to each other.

The following CAMPAIGN is the class, which is an abstraction that could represent any one of several specific campaigns. This class represents the relevant features that all campaigns have in common. Some examples of campaigns are - A series of magazine adverts for various yellow jewelry products, a national series of TV, cinema, and Magazine adverts. Along with the campaign we have the following classes in the Agate Ltd.. case study which is advertising company.



3. Class membership

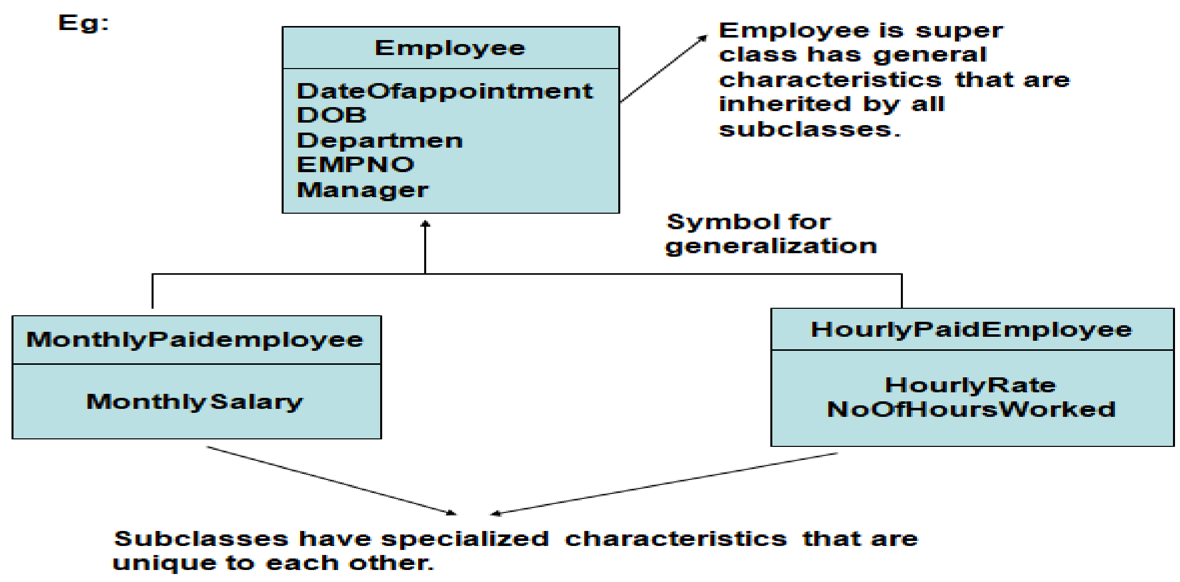
The idea that instances belong to a class logically implies that there must be a test that determines to which class an instance belongs. since membership is based on similarity, such a test will also be capable of determining whether any two instances belong to the same class. There are two types of logical similarity which must be tested.

1.Whether All the objects in a class share a common set of descriptive characteristics.

2. Whether all the objects in a class share a common set of valid behaviors or not.

4. Generalization: In the UML notation Generalization is defined as, it is a taxonomy relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and may contain additional information. Here the Taxonomy means a scheme of hierarchic classification-either an applied set of classifications, or the principles by which that set is constructed.

The main application for Generalization in OO is to describe relationships of similarity b/w classes. Object classes can be arranged into hierarchies.


Eg:
Employee
DateOfappointment
DOB
Departmen
EMPNO
Manager
Employee is super class has general characteristics that are inherited by all subclasses.
Symbol for generalization
MonthlyPaidemployee
MonthlySalary
HourlyPaidEmployee
HourlyRate
NoOfHoursWorked
Subclasses have specialized characteristics that are unique to each other.

5. Message Passing:

In an OO system, Objects communicate with each other by sending messages. In earlier approaches systems are developed tendency to separate data in a system from the process that act on the data. This method is appropriate but still has some difficulties. That is the process needs to understand the organization of the data that it uses, means process is called dependent on the structure of the data.

This dependency of process on data can also cause, if the data structures were changed for any reason, those processes which uses that data must also be changed.

OO systems avoids these problems by locating each process with the data it uses. Means this is another way of describing an OBJECT: is a data together with process that acts on the data. These Processes are called Operations, and each has a specific signature. An Operation signature is definition of its interface. In order to invoke an operation, its signature must be given.

6. Polymorphism:

When one person sends a message to another, it is often convenient to ignore many of the differences that exist b/w the various people that might receive the message.

This looks like a Polymorphism, which is important element in OO approaches, defines an ability to appear in many forms, and it refers to the possibility of identical messages being sent to the objects of different classes, each of which responds to the message in a different way.Polymorphism is a powerful concept for the information systems developer. It permits clear separation b/w different sub-systems which handles similar tasks in a different manner. This means system can be easily modified or extended to include extra features, since only the interfaces b/w classes need to be known.

7. Object state:

Objects can also occupy different states, and this affects they way that they have responded to messages. Each state is represented by the current values of data within the object , which can in turn be changed by the objects behavior in response to messages. According to BOOCH Object state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for event. This is fundamental concept to an understanding of the way that the behavior of an OO s/w system is controlled, so that the system responds in an correct way when an external event occurs.

3.a. Explain with an example how the classes can be identified using noun phrase approach? 6M

<u>Explanation-----4M Example----2M</u>

A. In this method, analyst read through the requirements or uses cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes All plurals are changed to singular, the nouns are listed, and the list divided into three categories relevant classes, fuzzy classes (the "fuzzy area," classes we are not sure about), and irrelevant classes as shown below.

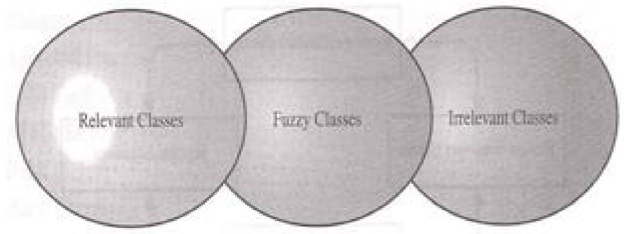


FIGURE 7-2
Using the noun phrase strategy, candidate classes can be divided into three categories: Relevant Classes, Fuzzy Area or Fuzzy Classes (those classes that we are not sure about), and Irrelevant Classes.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Here identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model .Analyst must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

1 Identifying Tentative Classes: The following are guidelines for selecting classes in an application:

- Look for nouns and noun phrases in the use cases.
- Some classes are implicit or taken from general knowledge.
- All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage.
- Carefully choose and define class names.

Identifying classes is an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages.

2 Selecting Classes from the Relevant and Fuzzy Categories:

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

a) **Redundant classes.** Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.

   Eg: Registrar, University I/C

b) **Adjectives classes.** "Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class".

   For example : Single account holders behave differently than Joint account holders, so the two should be classified as different classes.

c) **Attribute classes:** Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Details of Client are not classes but attributes of the Client class.

d) **Irrelevant classes:** Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

   Eg: Initial List of Noun Phrases for ATM BANK: Candidate Classes

The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Example List:

Account, Account Balance, Amount , Approval Process, ATM Card , ATM Machine, Bank

Bank Client, Card Cash, Check, Checking, Checking Account, Client, Client's Account,

Currency, Dollar, **envelope, Four Digits,** Fund, Invalid PIN, Message, Money, Password, PIN,

PIN Code, Record, Savings, Savings Account, **step,** Transaction, Transaction History

from the list bolded irreverent nouns can be eliminated

3. b. What is CRC? How is it used to identify classes? Explain with an example? 6M

<u>Explanation-----4M Example----2M</u>

A. *Class Responsibility Collaboration (CRC)* cards provide an effective technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfill the responsibilities.

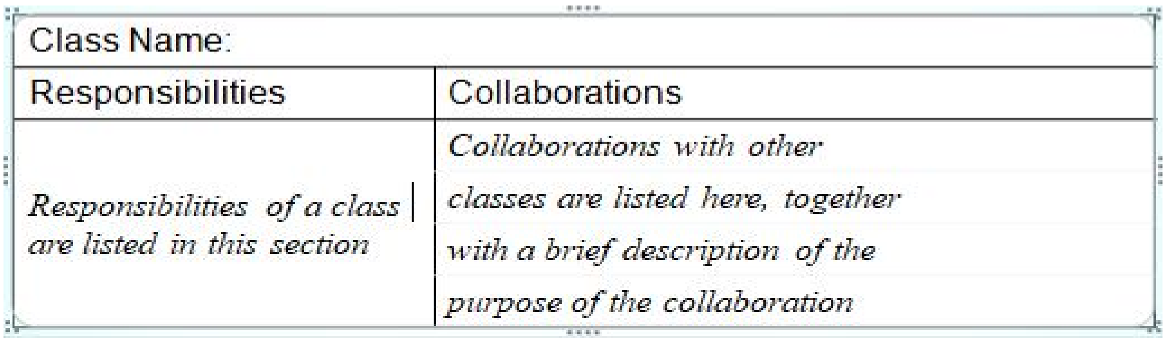
CRC cards can be used at several different stages of a project for different purposes.

1. They can be used early in a project to help the production of an initial class diagram .

2.To develop a shared understanding of user requirements among the members of the team.

3. CRCs are helpful in modelling object interaction.

The format of a typical CRC card is shown below



Class Name:	
Responsibilities	Collaborations
Responsibilities of a class are listed in this section	*Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration*

CRC cards are an aid to a group role-playing activity . Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class.

A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent. For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility. From a UML perspective, use of CRC cards is in analyzing the object interaction that is triggered by a particular use case scenario. The process of using CRC cards is usually structured as follows.

1.Conduct a session to identify which objects are involved in the use case.

2. Allocate each object to a team member who will play the role of that object.

3. Act out the use case.

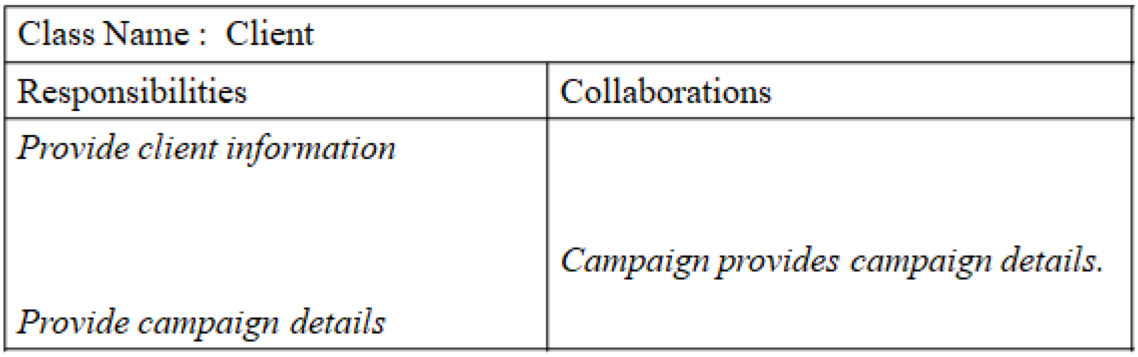
4. Identify and record any missing or redundant objects.

Before beginning a CRC session it is important that all team members are briefed on the organization of the session and a CRC session should be preceded by a separate exercise that identifies all the classes for that part of the application to be analyzed.

The team members to whom these classes are allocated can then prepare for the role playing exercise by considering in advance a first-cut allocation of responsibilities and identification of collaborations. Here , it is important to ensure that the environment in which the sessions take place is free from interruptions and free for the flow of ideas among team members.

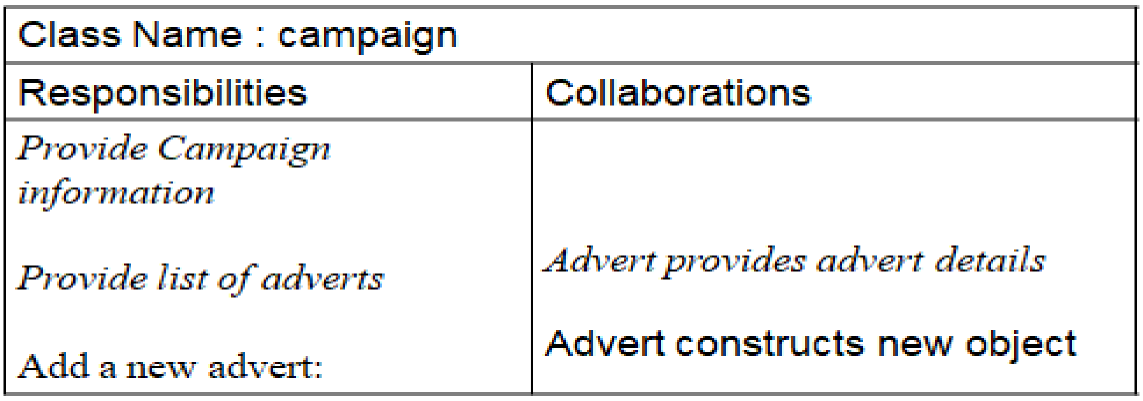
During a CRC card session, there must be an explicit strategy that helps to achieve an appropriate distribution of responsibilities among the classes. One simple but effective approach is to apply the rule that each object should be as lazy as possible, refusing to take on any additional responsibility unless instructed to do so by its fellow objects.

Example:



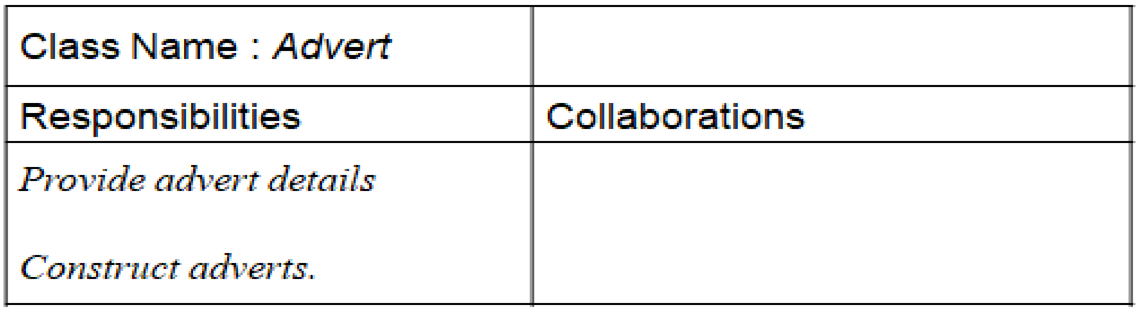
Class Name : Client	
Responsibilities	Collaborations
Provide client information	
Provide campaign details	*Campaign provides campaign details.*

CRC card for Client class in ADD A NEW ADVERT CAMPAIGN



Class Name : campaign	
Responsibilities	Collaborations
Provide Campaign information	
Provide list of adverts	*Advert provides advert details*
Add a new advert:	Advert constructs new object

CRC card for Campaign class in ADD A NEW ADVERT CAMPAIGN



Class Name : *Advert*	
Responsibilities	Collaborations
Provide advert details	
Construct adverts.	

CRC card for Advert class in ADD A NEW ADVERT CAMPAIGN

UNIT-II

4.a. Explain about a unified approach to software development patterns? 6M

Pattern definition----1M different patterns explanation ---5M

A. Pattern: Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Coplien identifies the critical aspects of a pattern as follows.

- It solves a problem.
- It is a proven concept.
- The solution is not obvious.
- It describes a relationship.

The pattern has a significant human component.

Architectural patterns – Responsibilities

1. Addresses some of the issues concerning the structural organization of software systems.
2. Architectural patterns also describe the structure and relationship of major components of a software system.
3. These patterns identify subsystems, their responsibilities and their interrelationships.

Design patterns – Responsibilities

1. These patterns identify the interrelationships among a group of software components describing their responsibilities, collaborations and structural relationships.
2. Idioms describe how to implement particular aspects of a software system in a given programming language.

Analysis patterns: Responsibilities

1. *These* are defined as describing groups of concepts that represent common constructions in domain modelling. These patterns may be applicable in one domain or in many domains.
2. The use of analysis patterns is an advanced approach that is principally of use to experienced analysts, They are closely related to design patterns also.
3. An analysis pattern is essentially a structure of classes and associations that is found to occur over and over again in many different modelling situations.

4.b. Explain the components of activity diagram with an example? 6M

Notations-----3M Example-----3M

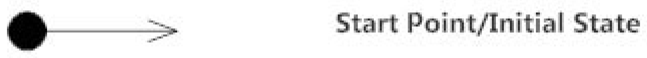
A. An activity diagram visually presents a series of actions or flow of control in a system similar to a flowchart or a data flow diagram. Activity diagrams are often used in business process modeling. They can also describe the steps in a use case diagram. Activities modeled can be sequential and concurrent. In both cases an activity diagram will have a beginning (an initial state) and an end (a final state).

Activity diagrams are inherently very flexible in their use, and therefore a little care should be exercised when they are employed in operation specification. A diagram may be drawn to represent a single operation on an object, but it may just as easily be drawn to represent a collaboration between several objects

Basic Activity Diagram Notations and Symbols

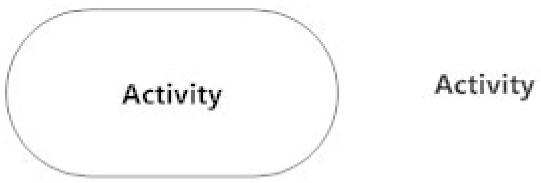
Initial State or Start Point

A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram. For activity diagram using swimlanes, make sure the start point is placed in the top left corner of the first column.



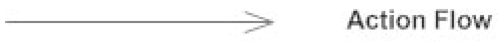
Activity or Action State

An action state represents the non-interruptible action of objects. You can draw an action state in SmartDraw using a rectangle with rounded corners.



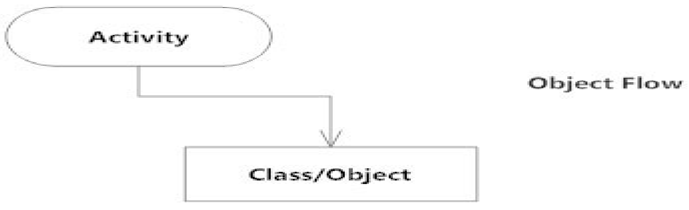
Action Flow

Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.



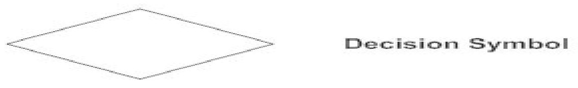
Object Flow

Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.



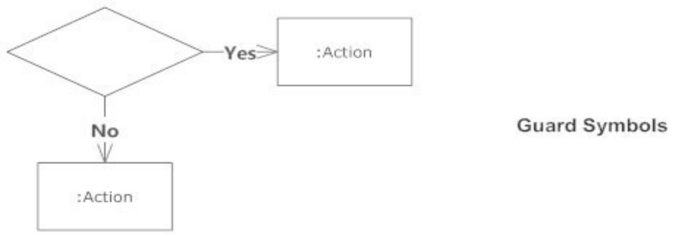
Decisions and Branching

A diamond represents a decision with alternate paths. When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities. The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Guards

In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity. These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.

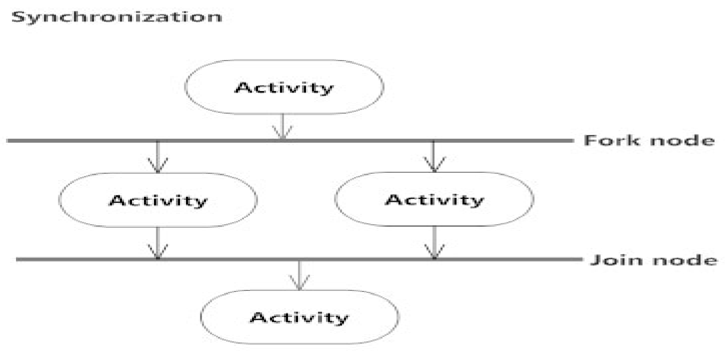


Synchronization

A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.

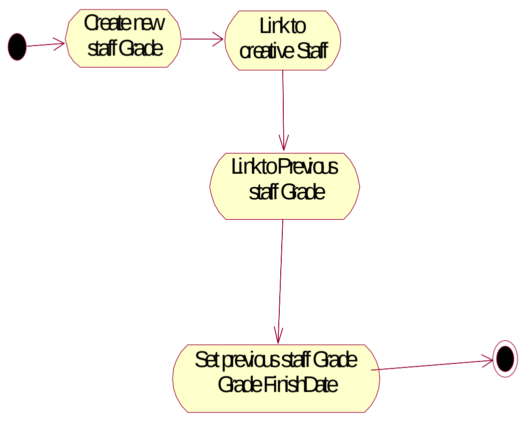
A join node joins multiple concurrent flows back into a single outgoing flow.

A fork and join mode used together are often referred to as synchronization.

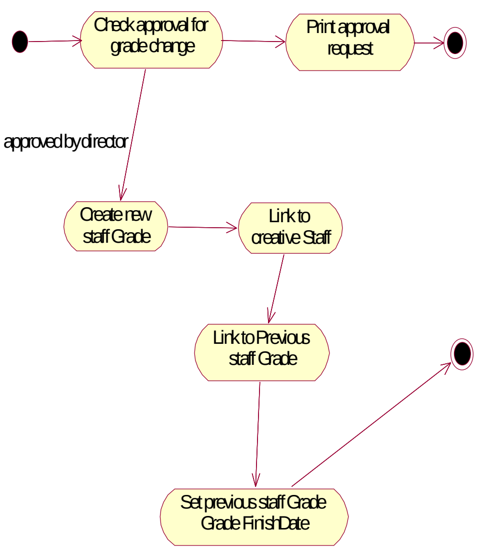


The following figure shows activity diagram for the operation CreativeStaff.changeGrade(). This example contains a single selection.

Example


Create new staff Grade
Link to creative Staff
Link to Previous staff Grade
Set previous staff Grade Grade Finish Date

There is no complex behaviour to be shown at this level of abstraction. However, if we consider the operation logic at a more detailed level some selection logic may become apparent. For this purpose the following figure shows more complex activity diagram for CreativeStaff. changeGrade () with an initial selection to check that approval has been given .



Check approval for grade change
Print approval request
approved by director
Create new staff Grade
Link to creative Staff
Link to Previous staff Grade
Set previous staff Grade Grade FinishDate

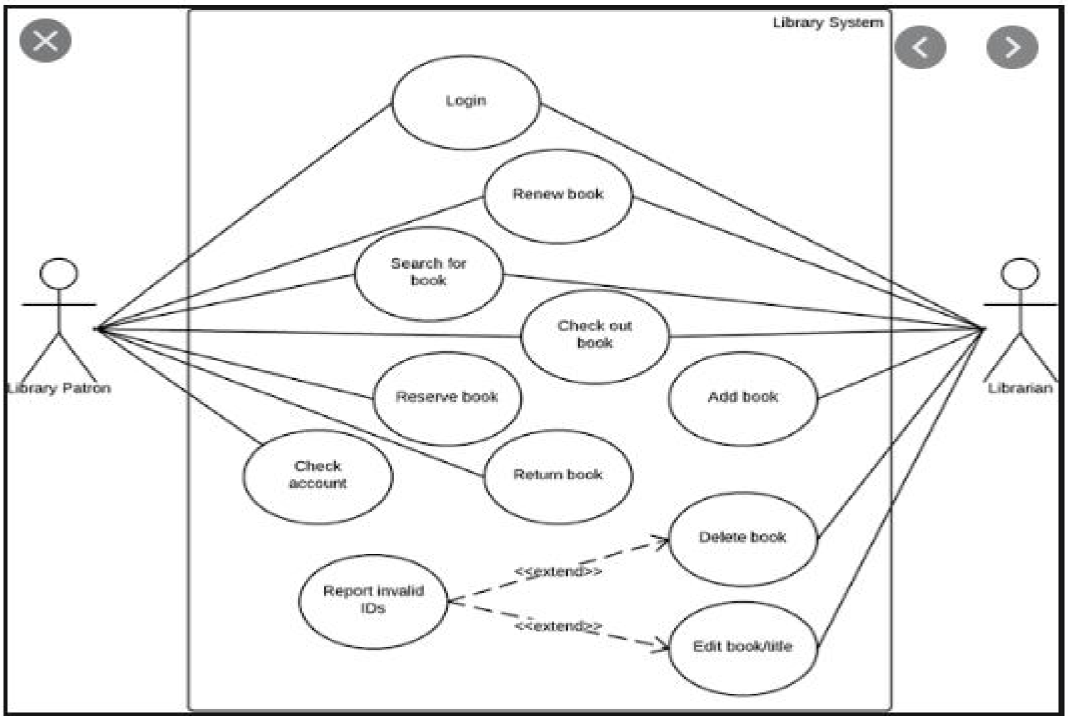
5. Consider a digital library system. Draw the following UML diagrams for the above mention system and explain 12M

i. Use case diagram

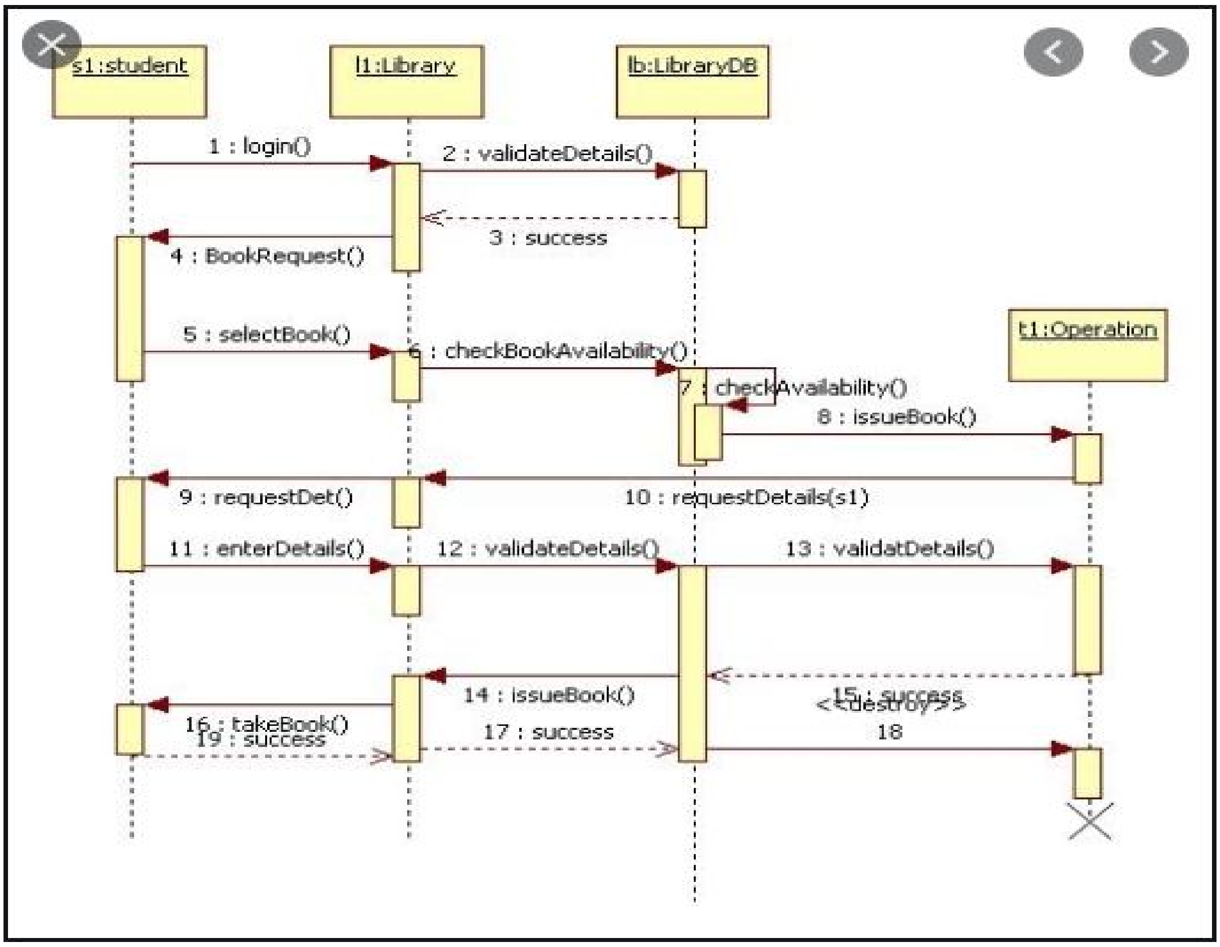
ii. Sequence diagram

iii. State chart diagram

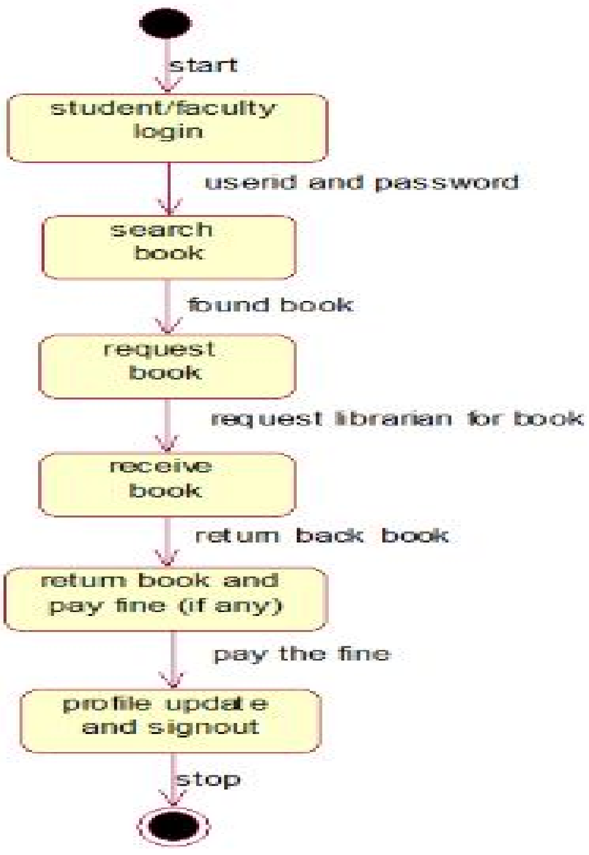
Any related diagram example can be considered each diagram carries 4----M



Use case diagram



Sequence diagram



State chart diagram

UNIT-III

6.a. Explain in detail the criteria for good design? 6M

Coupling----3M Cohesion----3M

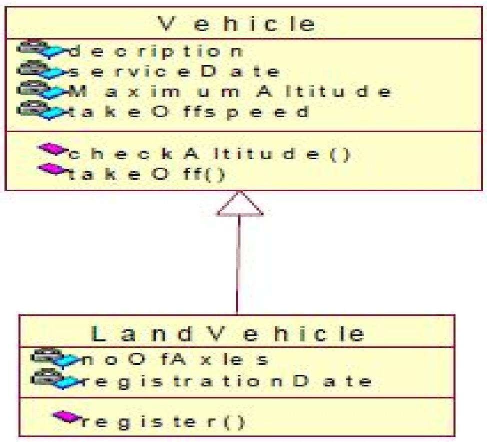
A. Criteria for Good Design

Coupling and cohesion – These factors coupling and cohesion are important factors for good design.

Coupling describes the degree of interconnectedness between design components and is reflected by the number of links an object has and by the degree of interaction the object has with other objects.

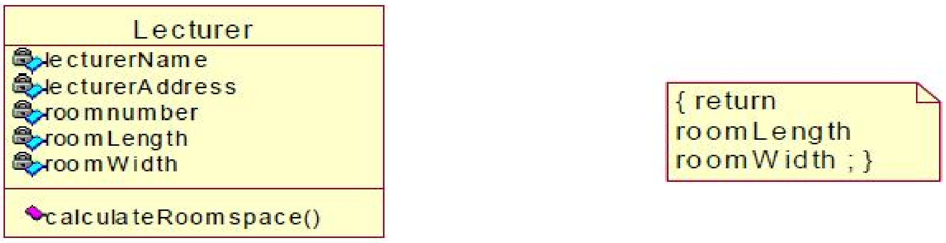
Cohesion is a measure of the degree to which an element contributes to a single purpose. The concepts of coupling and cohesion are not mutually exclusive but support each other. This criterion can be used within object-orientation as described below.

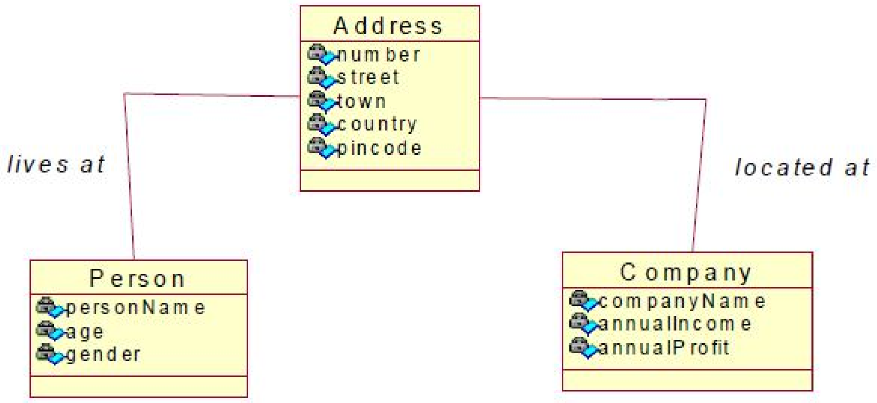
Interaction Coupling *is* a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types. Interaction coupling should be kept to a minimum to reduce the possibility of changes rippling through the interfaces and to make reuse easier. When an object is reused in another application it will still need to send these messages and hence needs objects in the new application that provide these services.
Inheritance Coupling describes the degree to which a subclass actually needs the features it inherits from its base class. For example, in the above figure, the inheritance hierarchy exhibits low inheritance coupling and is poorly designed. The subclass LandVehicle needs neither the attributes maximumAltitude and takeOff Speed nor the operations checkAltitude () and takeOff(). They have been inherited unnecessarily.
Operation Cohesion measures the degree to which an operation focuses on a single functional requirement. Good design produces highly cohesive operations, each of which deals with a single functional requirement. For example in the following figure , the operation calculateRoomSpace () is highly cohesive.



Class Cohesion reflects the degree to which a class is focused on a single requirement. The class Lecturer in the previous figure exhibits low levels of cohesion as it has three attributes (roomNumber, roomLength and roomWidth and one operation calculate RoomSpace ()) that would be more appropriate in a class Room. The class Lecturer should only have attributes that describe a Lecturer object (e.g.lecturerName and lecturerAddress) and operations that use them.

Specialization Cohesion addresses the semantic cohesion of inheritance hierarchies. For example in the following figure all the attributes and operations of the Address base class are used by the derived classes - this hierarchy has high inheritance coupling. However, it is neither true that a person is a kind of address nor that a company is a kind of address. The example is only using inheritance as a syntactic structure for sharing attributes and operations. This structure has low specialization cohesion and is poor design. It does not reflect meaningful inheritance in the problemdomain.





6. b. Explain about major elements of system design. **6M**

Explanation-----6M

A. The major elements of system Design

The system design activity specifies the context within which detailed design will occur. A major part of system design is defining the *system architecture.* The meaning and scope of the term architecture for computerized information systems is much debated but it is generally accepted that it is an important feature of the delivered system.

The architecture of a system is concerned with its overall structure, the relationships among its major components and their interactions. If the system being considered contains human, software and hardware elements then its architecture includes how these elements are structured and how they interact.

On the other hand, if the system being considered comprises software and hardware, then its architecture only concerns these elements. It is important to consider the structure of the software elements of the system and this is termed the *software architecture.* The hardware architecture of a system describes computers and peripherals required for the system and how software is allocated to them.

The architecture of the information system is first considered early in the project during the requirements capture and analysis activities. This first view of the system architecture is driven significantly by the use cases and then informs the continuing requirements capture and analysis activities. This forms a useful basis from which to develop the design architecture.

The detailed software architecture of a computerized information system develops as the design process continues into object design but it is important to identify an overall system architecture within which the detail can be refined. High-level architectural decisions that are made during system design determine how successfully the system will meet its non-functional objectives (e.g. performance, extensibility) and thus its long-term utility for the client.

Reuse is one of the much wanted benefits of object-orientation and poor software architecture usually reduces both the reusability of the components produced and the opportunity to reuse existing components.

During system design we need to consider the following activities.

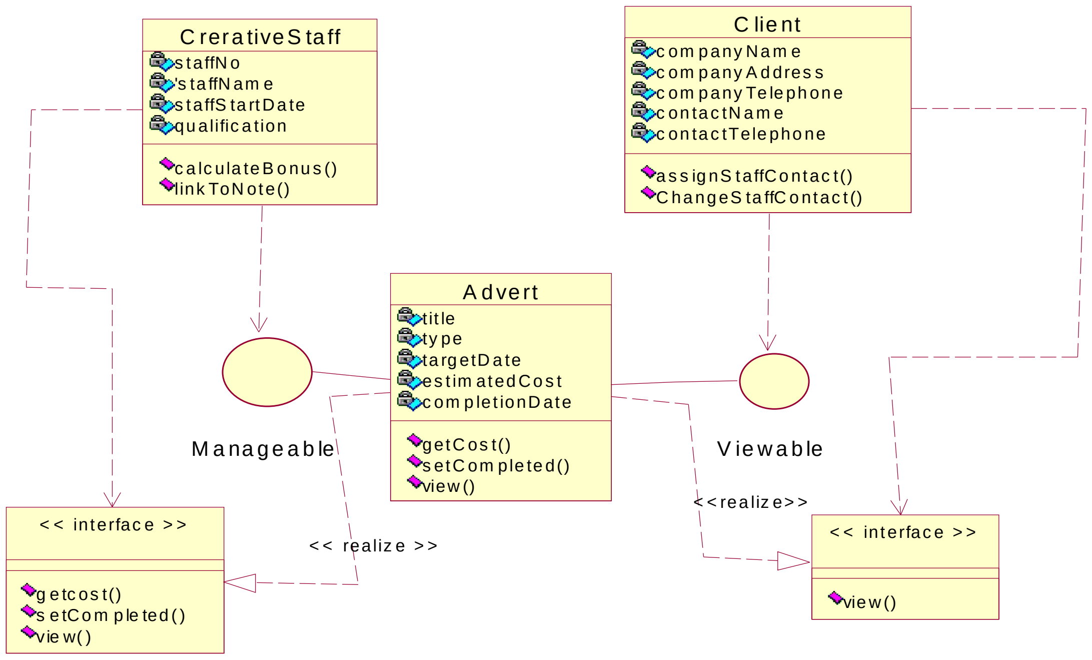
- ❖ Sub-systems and major components are identified.
- ❖ Any inherent concurrency is identified.
- ❖ Sub-systems are allocated to processors.
- ❖ A data management strategy is selected.
- ❖ A strategy and standards for human-computer interaction are chosen.
- ❖ Code development standards are specified.
- ❖ The control aspects of the application are planned.
- ❖ Test plans are produced.
- ❖ Priorities are set for design trade-offs.
- ❖ Implementation requirements are identified (for example, data conversion).

7. a. What are interface objects? Explain how to design them? 6M

A. Generally a class may present more than one external interface to other classes or the same interface may be required from more than one class. An interface in UML is a group of externally visible (i.e. public) operations. The interface contains no internal structure, it has no attributes, no associations and the implementation of the operations is not defined. Formally, an interface is equivalent to an abstract class that has no attributes, no associations and only abstract operations.

The following figure shows two alternative notations for an interface. The simpler of the two UML interface notations is a circle. This is attached by a solid line to the classes that support the interface. For example, in Figure the Advert class supports two interfaces, Manageable and Viewable, that is, it provides all of the operations specified by the interface. The circle notation does not include a list of the operations provided by the interface type, though they should be listed in the repository. The dashed arrow from the CreativeStaff class to the Manageable interface circle icon indicates that it uses or needs, at most, the operations provided by the interface.

The alternative notation uses a stereotyped class icon. As an interface only specifies the operations and has no internal structure, the attributes compartment is omitted. This notation lists the operations on the diagram. The *realize* relationship, represented by the dashed line with a triangular arrowhead, indicates that the client class (e.g. Advert) supports at least the operations listed in the interface .Again the dashed arrow from CreativeStaff means that the class needs or uses no more than the operations listed in the interface.



7. b. What is design pattern? How to use design patterns and explain the benefits of design patterns? 6M

<u>Design pattern----1M Usage-----3M Benefits-----2M</u>

A. Design pattern: In software engineering, ad design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

How to Use Design Patterns

The use of a pattern requires careful analysis of the problem that is to be addressed and the context in which it occurs. Before contemplating the application of patterns within a software development environment it is important to ensure that all members of the team receive appropriate training.

The following are the issues to be considered before applying a pattern to resolve the problem.

- Is there a pattern that addresses a similar problem?
- Does the pattern trigger an alternative solution that may be more acceptable?
- Is there a simpler solution? Patterns should not be used just for the sake of it.
- Is the context of the pattern consistent with that of the problem?
- Are the consequences of using the pattern acceptable?
- Are constraints imposed by the software environment that would conflict with the use of the pattern?

Gamma et al. suggested seven-part procedure that should be followed after an appropriate pattern has been selected in order to apply it successfully.

1. Read the pattern to get a complete overview.
2. Study the Structure, Participants and Collaborations of the pattern in detail.
3. Examine the Sample Code to see an example of the pattern in use.
4. Choose names for the pattern's participants that is classes that are meaningful to the application.

5) Define the classes.
6) Choose application specific names for the operations.
7) Implement operations that perform the responsibilities and collaborations in

   the pattern.

Benefits

One of the most important benefits of object-orientation is reuse. Reuse at the object and class level has proved more tangible than was initially expected. Patterns provide a mechanism for the reuse of generic solutions for object-oriented and other approaches. They provide a strong reuse culture. Within the design context, patterns suggest reusable elements of design and, most significantly, reusable elements of demonstrably successful designs. This reuse permits the transfer of expertise to less experienced developers so that a pattern can be applied again and again

Another benefit gained from patterns is that they offer a vocabulary for discussing the problem domain means whether it be analysis, design or some other aspect of information systems development at a higher level of abstraction than the class and object making it easier to consider micro-architectural issues. Pattern catalogues and pattern languages offer a rich source of experience that can be explored and provide patterns that can be used together to generate effective systems.

UNIT-IV

8. a. Explain the architecture of presentation layer? 6M

Explanation----6M

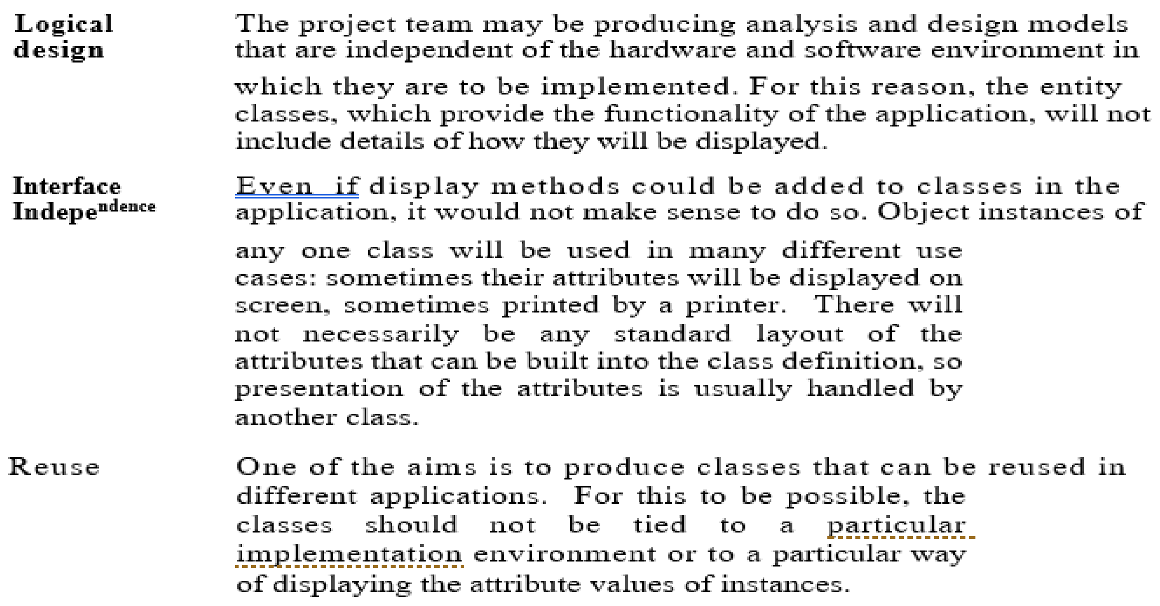
The architecture of presentation layer

The idea of boundary classes was introduced, a layered model of the system was presented the three tier architecture was presented in a way to separate out user interface classes from business and application logic. classes and from mechanisms for data storage. There are number of reasons for doing this, and these are shown in Figure 17.1.

This is not to say that classes should contain no means of displaying their contents to the outside world. It is common practice to include in each class a print ( ) method that can be used to test the classes before the presentation layer has been developed. Such methods typically take an output stream (a file or a terminal window) as a parameter and produce a string representation of their attributes on that stream. This enables the programmer to check the results of operations carried out by classes without needing the full system in place.

The three-tier architecture was discussed in Sections Different approaches to object-oriented development use different names for the layers of the three-tier architecture. The Unified Process uses the terms boundary, control and entity classes for the three types of classes, and these are the terms that we have used. Coad and Yourdon (1991) call the presentation layer the Human Interaction Component and keep it separate from what they call the Problem Domain Component, which contains what we have called the entity classes. Developers using Smalltalk to implement systems have for many years adopted a similar approach using the Model-View-Controller (MVC) approach that was described in Chapter 13. In the (MVC) approach a system is divided into three components:

- ■ Model—the classes that provide the functionality of the system;



Logical design	The project team may be producing analysis and design models that are independent of the hardware and software environment in which they are to be implemented. For this reason, the entity classes, which provide the functionality of the application, will not include details of how they will be displayed.
Interface Independence	Even if display methods could be added to classes in the application, it would not make sense to do so. Object instances of any one class will be used in many different use cases: sometimes their attributes will be displayed on screen, sometimes printed by a printer. There will not necessarily be any standard layout of the attributes that can be built into the class definition, so presentation of the attributes is usually handled by another class.
Reuse	One of the aims is to produce classes that can be reused in different applications. For this to be possible, the classes should not be tied to a particular implementation environment or to a particular way of displaying the attribute values of instances.

Figure 17 .1 Reasons for separating business and user interface classes.

■ View—the classes that provide the display to the user;

■ Controller—the classes that handle the input from the user and send messages to the other two components to tell them what operations to carry out.

Whatever approach is chosen in a project, all these approaches share the objective of keeping the behaviour of the interface separate from the behaviour of the classes that provide the main functionality of the system. To use the anthropomorphic style of some authors about object-oriented systems, the entity classes 'know' nothing about how they will be displayed.

Taking a three-tier architectural approach does not necessarily mean that the dif ferent types of classes will end up running on different machines or even that they will be completely separate. It is useful to distinguish between the logical architecture of the system and the physical architecture. The physical architecture may combine layers of the logical architecture on a single physical platform or it may split logical layers across physical systems. If you are designing Java applets, some of the responsibilities for control will be located in the applet class itself, together with the responsibilities of the boundary class, while other control responsibilities may be located in many classes a server together with entity classes. In a distributed system the entity class may exist in different databases on different servers and the control classes would pull the data from these different sources in order to deliver to the boundary classes.

8. b. Explain the modelling the interface using state chart? 6M

The user may choose to check more than one budget. What happens if they select a different client—how does that affect the other fields where data has already been selected? All these issues can be modelled using a state chart diagram. State charts were introduced in Chapter 11, and were used there to model the way that events affect instances of a class over its lifetime. They can also be used to model the short-term effects of events in the user interface. Browne (1994) uses state charts in this way to model the user interface as part of the STUDIO methodology. Horrocks (1999) uses state charts in a more rigorous way than Browne in his user interface-control-model (UCM) architecture and relates the use of state charts to coding and testing of the user interface. Browne's approach leads to a bottom-up design of the interface, assembling state charts for components into a complete model of an interface; Horrocks develops his state charts in a top-down way, successively introducing nested sub-states where they are necessary. We are using Horrocks' approach in what follows.

For the example that follows, we are using the original design for the user interface with dropdowns for client and Campaign, as in the prototype of Figure 17.6. As a design principle in our user interfaces, we want to prevent users from making errors wherever possible rather than having to carry out a lot of validation of data entry in order to pick up errors that have been made. One way of doing this is to constrain what users can do when they are interacting with the interface. For example, in the Check campaign budget user interface it makes no sense to click the check button until both a client and a campaign have been selected. Rather than check whether a client and campaign have been selected every time the button is clicked, we can choose only to enable the button when we know that both have been selected. To do this we need to model the state of the user interface and it is this that we model using state charts. This process involves five tasks.

- ■ Describe the high-level requirements and main user tasks.
- ■ Describe the user interface behaviour.
- ■ Define user interface rules
- ■ Draw the state char and successively refine it.
- ■ Prepare an event action table.

Describe the high level requirements and main user tasks

The requirement here is that the users must be able to check whether the budget for an advertising campaign has been exceeded or not. This is calculated by summing the cost of all the adverts in a campaign, adding a percentage for overheads and sub-
tracting the result from the planned budget. A negative value indicates that the budget has been overspent. This information is used by a campaign manager.

Describe the user interface behaviour

There are five elements of the user interface: the client dropdown, the campaign drop-down, the budget text field, the check button and the close button. These are shown in Figure 17.6.

The ***client* dropdown** displays a list of clients. When a client is selected, their campaigns will be displayed in the campaign dropdown.

The **campaign dropdown** displays a list of campaigns belonging to the client selected in the client dropdown. When a campaign is selected the check, button is enabled.

The **budget text field** displays the result of the calculation to check the budget. The **check button** causes the calculation of the budget balance to take place. The **close button** closes the window and exits the use case.

Define user interface rules

User interface objects with constant behaviour

■ The client dropdown has constant behaviour. Whenever a client is selected, a list of campaigns is loaded into the campaign dropdown.

■ The budget text field is initially empty. It is cleared whenever a new client is selected, or a new campaign is selected. It is not editable.

■ The close button may be pressed at any time to close the window.

User interface objects with varying behaviour

■ The campaign dropdown is initially disabled. No campaign can be selected until a client has been selected. Once it has been loaded with a list of campaigns it is enabled.

■ The check button is initially disabled. It is enabled when a campaign is selected. **It is** disabled whenever a new client is selected.

Entry and exit events

The window is entered from the main window when the Check campaign budget menu item is selected.

9. a. What are implementation diagrams? Explain any one with example? 6M

<u>Types ---1M any one diagram explanation 3M Example---2M</u>

Two types implementation diagrams in UML Terminology are

1. Component Diagrams
2. Deployment diagrams

In a large project there will be many files that make up the system. These files will have dependencies on one another. The nature of these dependencies will depend on the language or languages used for the development and may exist at compile-time, at link-time or at run-time. There are also dependencies between source code files and the executable files or bytecode files that are derived from them by compilation. Component diagrams are one of the two types of implementation diagram in UML.

Component diagrams show these dependencies between software components in the system. Stereotypes can be used to show dependencies that are specific to languages also. A component diagram shows the allocation of classes and objects to components in the physical design of a system. A component diagram may represent all or part of the component architecture of a system along with dependency relationships.

The dependency relationship indicates that one entity in a component diagram uses the services or facilities of another.

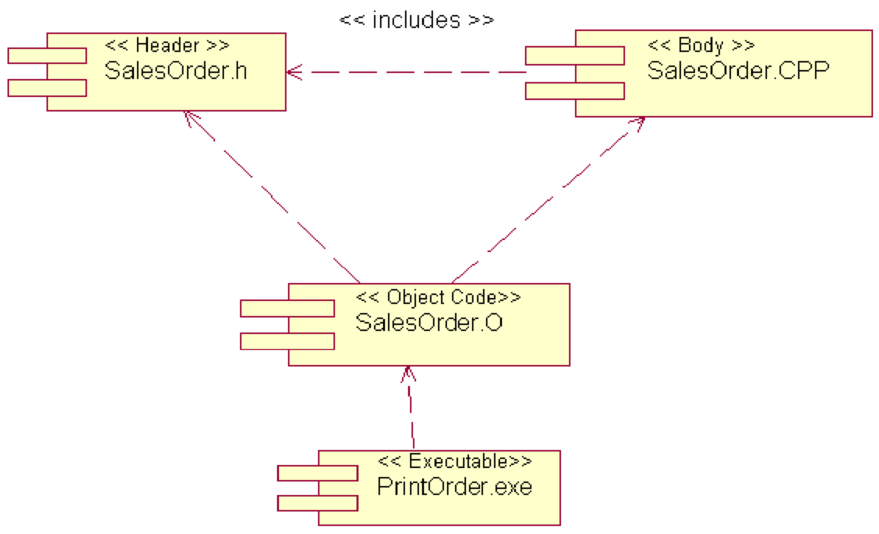
- ➢ Dependencies in the component diagram represent compilation dependencies.
- ➢ The dependency relationship may also be used to show calling dependencies among components, using dependency arrows from components to interfaces on other components.

Here We have the following distinction between them.

- Components in a component diagram should be the physical components of a system.
- During analysis and the early stages of design, package diagrams can be used to show the logical grouping of class diagrams or of models that use other kinds of diagrams into packages relating to sub-systems.
- During implementation, package diagrams can be used to show the grouping of physical components into sub-systems .
- component diagrams can also be combined with deployment diagrams to show the physical location of components of the system. The classes in one logical package may be distributed across physical locations in a physical system, and the component diagram and deployment diagram can be used to show this.

The following figure shows a component diagram that represents the dependency of a C++ source code file on the associated header file, the dependency of the object file on both and the dependency of an executable on the object file. Stereotypes can be used to show the types of different components.

Example



Component diagram

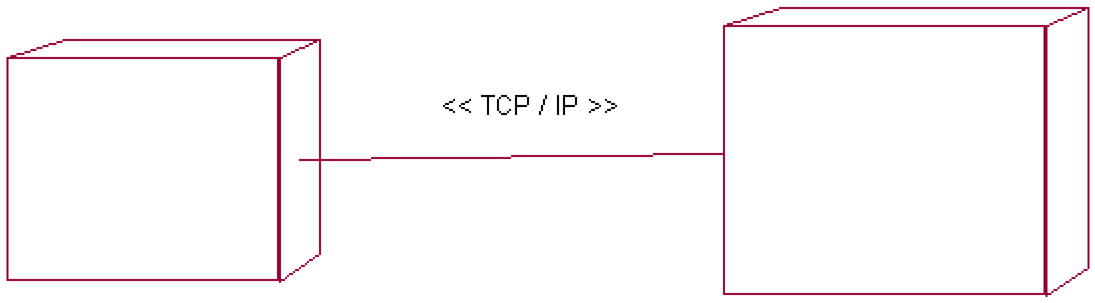
Deployment diagram.

The second type of implementation diagram provided by UML is the deployment diagram. Deployment diagrams are used to show the configuration of run-time processing elements and the software components and processes that are located on them.

Deployment diagrams are made up of nodes and communication associations. Nodes are typically used to show computers and the communication associations show the network and protocols that are used to communicate between nodes. Nodes can be used to show other processing resources such as people or mechanical resources.

Nodes are drawn as 3D views of cubes or rectangular prisms, and the following figure shows a simplest deployment diagram where the nodes connected by communication associations.

Example



Deployment diagram

9. b. What is reusability? Explain the planning strategy for reusability? 6M

<u>Reusability—1M Planning ---4M Diagram—1M</u>

A. Reuse: Reusability is the use of existing assets in some form within the software product development process. Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Other industries have long profited from reusable components.

Planning a strategy for Reuse

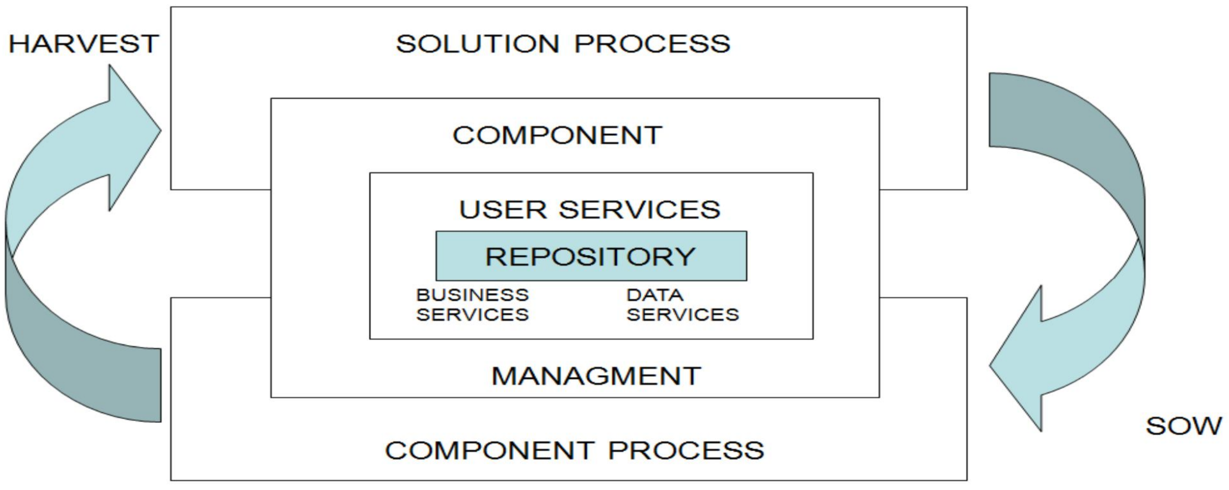
The following are two approaches to the introduction of a reuse strategy.
The SELECT Perspective

Allen and Frost describes the SELECT Perspective approach to the developers of reusable components. At the level of practical techniques, this includes guidelines for the modelling of business-oriented components and for wrapping legacy software in component wrappers. They distinguish between reuse at the level of component packages, which consist of executable components grouped together, and service packages, which are abstractions of components that group together business services.

In order to develop reusable components while achieving the development of a system to meet users' needs, the Perspective approach breaks the development process into two parts:

1. The solution process

2. The component process.



The SELECT Perspective service-based process

The **solution process** focuses on specific business needs and delivering services to meet the users' requirements. Its products have immediately definable business value. During the solution process, developers will draw on the component process in their search for reusable components that can be applied to the project.

The **component process** focuses on developing reusable components in packages that group together families of classes to deliver generic business services. During the component process, the developers produce components that can be reused in the solution process. The component process also searches out opportunities to reuse services from existing legacy systems and legacy databases and from other packages of components.

Software support is needed for effective component reuse to take place. This support takes the form of repository-based component management software. Components are placed in the repository as a means of publishing them and making them available to other users. The repository is made up of catalogues and the catalogues contain details of components, their specifications and their interfaces. Component management software tools provide the functionality for adding components to the repository and for browsing and searching for components. Component management software may be integrated with CASE tools to allow the storage of analysis and design models as well as source code and executables.

Scheme prepared by   Signature of the HOD, IT Dept.

Paper Evaluators:

S.No	**Name Of the College**	**Name of the Faculty**	**Signature**