# Anna University
# Solved Question Papers

## B.E./B.Tech. 6th Semester

## Computer Science and Engineering

PEARSON

Chennai • Delhi

ISBN 978-93-325-4248-8

**First Impression**

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128, formerly known as TutorVista Global Pvt. Ltd, licensee of Pearson Education in South Asia.

Head Office: 7th Floor, Knowledge Boulevard, A-8(A), Sector 62, Noida 201 309, U.P., India.
Registered Office: Module G4, Ground Floor, Elnet Software City, TS-140, Blocks 2 and 9,
Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India. Fax: 080-30461003,
Phone: 080-30461060, www.pearson.co.in, Email: companysecretary.india@pearson.com.

# Semester-VI

# Object Oriented Analysis and Design

**B.E./B.Tech DEGREE EXAMINATION, MAY/JUNE 2012**

**Sixth Semester**

**(Regulation 2008)**

**Computer Science and Engineering**

**CS 2353/CS 63/10144 CS 603-OBJECT ORIENTED ANALYSIS AND DESIGN**

**(Common to Information Technology)**

**Time: Three hours** **Maximum: 100 marks**

**Answer All Questions.**

**PART A – (10 × 2 = 20 marks)**

1. What is UML?
2. List the relationships used in use cases.
3. What is Elaboration?
4. Define Aggregation and Composition.
5. What is the use of System sequence diagram?
6. Define Package and draw the UML notation for Package.
7. When to use Patterns.
8. Define Coupling.
9. What is the use of component diagram?
10. Give the meaning of Event, State and Transition.

## PART B – (5 × 16 = 80 marks)

11. (a) Explain the different phases of Unified Process.

Or

(b) Explain with an example, how usecase modeling is used to describe functional requirements. Identify the actors, scenario and use cases for the example.

12. (a) Describe the strategies used to identify conceptual classes. Describe the steps to create a domain model used for representing conceptual classes.

Or

(b) When to use Activity diagram. Describe the situations with an example.

13. (a) Compare Sequence Versus Collaboration diagram with suitable example.

Or

(b) (i) Describe the UML notation for class diagram with an example. (8)

(ii) Explain the concept of Link, Association and Inheritance. (8)

14. (a) (i) Describe the concept of Creator. (7)

(ii) Explain about Low coupling, Controller and High cohesion. (3 × 3 = 9)

Or

(b) Write short notes on adapter, singleton, factory and observer patterns. (4 × 4 = 16)

15. (a) Explain about Operation contracts.

Or

(b) Discuss about UML deployment and component diagrams with suitable example.

# Solutions

## PART A

1. • UML is Unified Modeling Language
   • It is a visual language for specifying constructing and documenting the artifacts of systems.
   • UML defines various UML profiles.

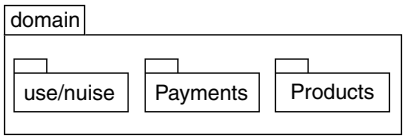
2. (1) The **include** Relationship :
   • It is common to have some partial behaviour that is common across several use cases.
   • It is desirable to separate it into its own sub function.
   (2) The **extend** Relationship:-
   To create an extending or addition use case, and within it, describe where and under what condition it extends the behaviour of some base use case.

3. Elaboration is a refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.

4. **Aggregation:**
   Is a Vague kind of association in the UML that loosely suggests whole part relationships.
   **Composition:**
   Is a Composite aggregation, is a strong instance of the part.

5. • Used to illustrate how objects interact via messages.
   • They are used for dynamic object modeling.
   • used for object designing.
   • Excellent for documentation.
   • They are space efficient.

6. The same subject area closely related by concept or purpose in a class hierarchy in the same use cases & strongly associated called as package.



7. • When related alternatives or behaviours vary by type (Polymorphism)
   • When highly cohesive set of responsibilities to an artificial or convenience (fabrication)
   • To avoid direct coupling between 2 things patterns are used.

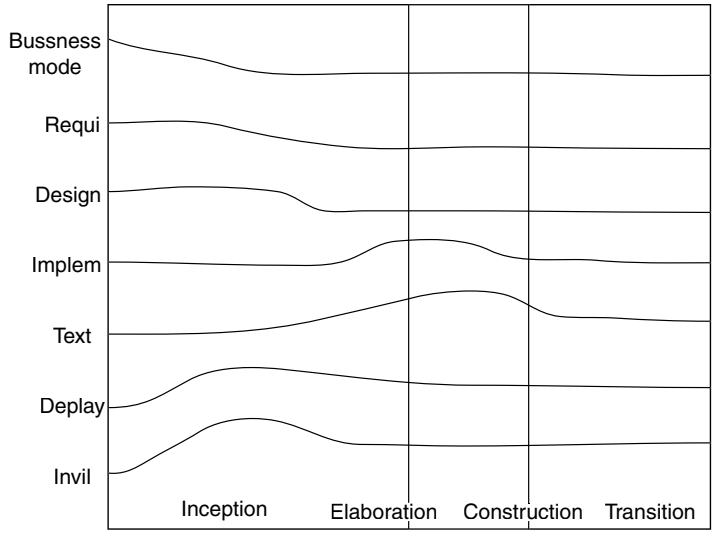
8. Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

9. • Component diagrams are used as a design level perspective.
   • It is modular, self-contained and replacable.
   • It leads to no dependency it is a relatively stand alone module.

10. **State**: is the condition of an object at a moment in time.
    **Event:** is a significant or noteworthy occurrence
    **Transition:** relationship between two states when an event occurs.

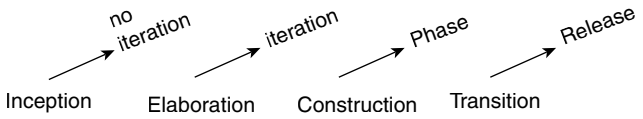
## PART B

11. (a) Phases:
    (1) Inception
    (2) Ellaboration
    (3) Construction
    (4) Transition
    (1) **Inception**: approximate vision, business case scope, vague estimates.
    (2) **Elaboration:** refined vision, iterative implement of the core architecture, resolution of high risks.
    (3) **Construction:** iterative implementation of the remaining lower risk and easier elements & preparation for deployment.
    (4) Transition:
    Beta tests, deployment.
    • Inception is not a requirements phase.
    • Investigation is done to support a decision to continue or stop.
    • Elaboration is not a requirements phase. Core architecture is iteratively.




(b) UP disciplines

Impelemented, and high risk issues are mitigated.




UP Disciplines:

(1) **Business Modeling:**
Domain model artifact, to visualize noteworthy concepts in the application domain.

(2) **Requirements:**
The Use-case model and supplementary specification artifacts to capture function and non functional requirements.

(3) **Design:**
The design model artifact, to design the software objects.
Implementation means programming is building the system, not deploying it.

The environment discipline refers to establishing the tools & customizing the process for projects.

(b)

- Use cases are not diagrams, they are text.

Use cases often need to be more detailed or structured than this eg, but discovering, recording functional requirements, by writing stories of using a system to fulfill user goals.

**Actor:**

Something with behaviour such as person, computer system or organization. eg: cashier
Scenerio:
Specific sequence of actions and interactions L/W actors & the system also called as use case instance.
Use case is a collection of related success & failure scenario describe an actor using a system
UP development case customize process.

- Three optinal artifacts in the UP.
- Models, diagrams, documents are optional.
- The choices & UP artifacts for a project may be written up in a short document called Development case containing UP phases,
- Tackle, high risk & high value issues in early iterations
- Continuously engage users for evaluation, feedback, requirements
- Build a cohesive, core architecture in early iterations.
- Apply Use cases where appropriate.
- Carefully manage requirements
- Practice change requests & configuration management.
  Support the goal.
- Use Case Model within the requirements discipline.
- Set of all written use cases, is a model of the system's functionality and environment.
- Use cases are text document, not diagrams.
- It is mainly for showing context b/w objects.
  (1) to capture goals
  (2) easier for customers
  (3) Important for avoid risk.
  (4) Simple to design.

**Functional Requirements:**

- Primarily functional or behavioral requirements that indicate what the system will do.
- FURPS +requirements emphasizes the functional Requirements.
- Related viewport is that a use case defines a contract of how a system will behave.

**Primary actor:** has user goes fulfilled through services
**Supporting actor:** provides service to the SUD.

**Off stage user:** interest in the behaviour of the use case.
**Brief:** one paragraph summary.
**Casual:** multiple paragraphs.
**Fully dressed:** All steps & variations are written in detail.
Fully dresses Use case:

1. Use Case Name:
2. Scope
3. Level
4. Primary actor
5. Stakeholders
6. Precondition
7. Success scenerio

8 Extensions
9 Special Requirements

Eg:

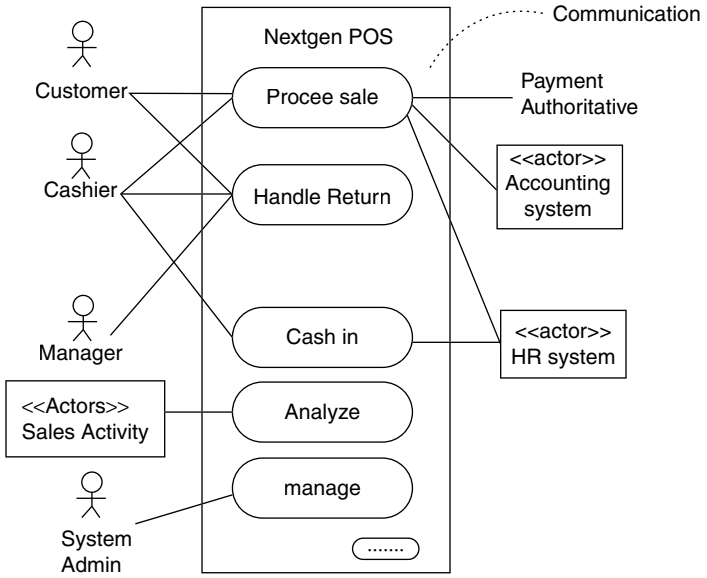



**Fig:** Use case context diagram

12 (a) **Three Strategies:**

1. Reuse or modify existing models
2. Use a category list
3. Identity noun phrases

**Use a category Lists:**

Making a list of Candidate conceptual classes.

- Common categories that are usually worth considering, with an emphasis on business information.

Eg

1. Pos
2. Monopoly
3. Airline

1. Business transactions
2. Translation line items
3. product or service related to a transaction or a transaction line item.
4. Where is the transaction recorded,
5. Catalogs
6. Contains of things
7. Things in container
8. Other collaborating systems
9. Financial instruments

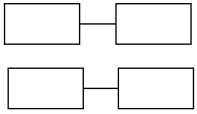
**Finding conceptual class with**

- Useful technique suggested in is linguistic analysis.
- Candidate conceptual classes or attributes.

**Draw Conceptual Classes:**

Category, & noun phrase analysis, list generated of candidate conceptual classes for the domain.

Sale Cachier
CashPayment customer
Sales LineItem store.
Item Product Distribute



Description class:-

Contains info that describes some else. Price, picture, text descriptions.

- Represents a physical item in a store.
- Duplicate data avoided by this.

**Steps for creating Domain Model:**

(1) Find the conceptual classes
(2) Draw them as classes
(3) Add associations and attributes.

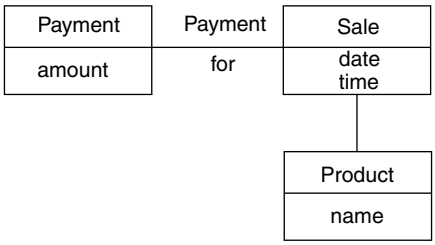
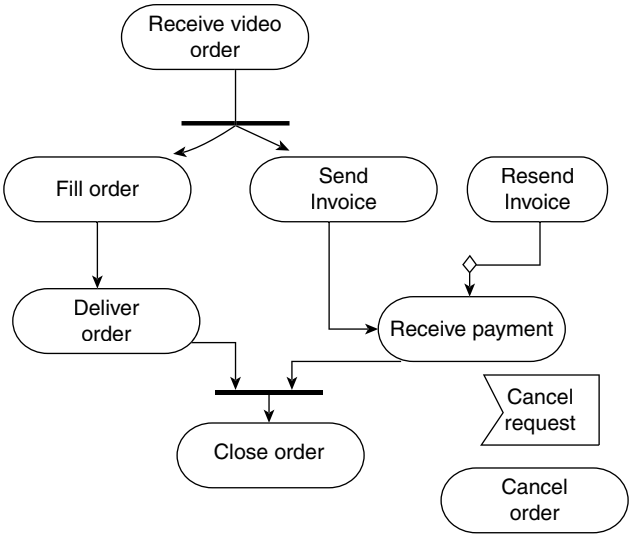



**Fig:** Domain Model

(b) To understand the data flow of the objects & messages the Activity diagram used.

- It also used in data flow modelling
- DFD were useful to document the major data flows.
- Many UML notations are used in activity diagrams.
- Rate symbol used for expanded activity diagram.
- Any branch happen represent mutual exclusion diamond symbol used.
- Merge symbol is used for contrast to a join.
- One of the UP disciplines is Business modeling.
- It's purpose is to understand the communication b/w objects & dynamics of the organization.
- During a system is to be deployed activity diagram is used.
- Key artifact of the business modeling discipline is the Business object model.
- Visualizes how a business works, using UML class, sequence, & activity diagrams

eg



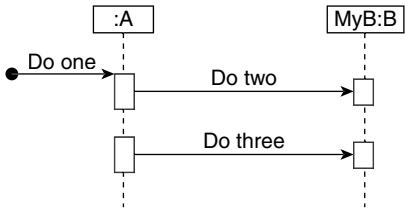

Creating an object model from an existing relational data must be database layout is often referred to as reverse engineering.

Conversely creating relational schema from an existing object model often is referred to as forward engineering.

Blend is an example of such a tool. Java Blend allows the develop access to relational data as Java objects, thus avoiding the mismatch between the relational and object data models.

13 (a) Sequence diagram:

Illustrate interactions in a kind of fence format, in which each new object is added to the right.




```
Public class A
{
Private B myB = new B();
Public void do one()
{
myB. do Two();
```

myB. do Three();
}}

**Strengths & weakness:**

- greater notational power
- simply read top to bottom
- excellent for documentation.

**Space efficient**

- Modification is difficult.
- Vertical expansion for new objects.
- new Object should add at right.
- large set of detailed notation options.

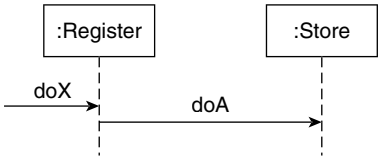
Lifeline : informally represent the related parts existing in the sequence operation.

**Message Expression:-**

return = message (Parameter = Parameter type) = return type

**Singleton Objects:**

Onely one instance of a class. Instantiated –never two.




doX doA are messages.

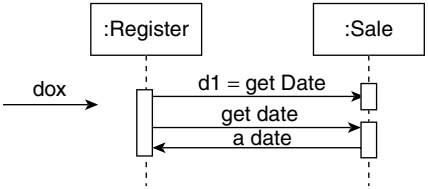
**Specification bars:**

also called as activation bar or execution bar.

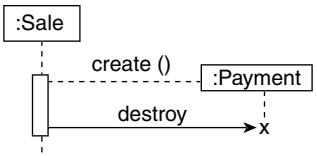
- the bar is optional.

**Reply or Return:**

Return var = message (Parameter)

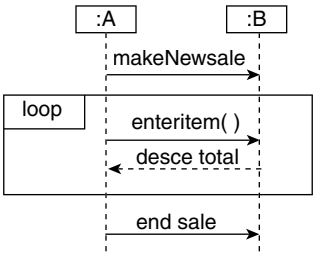


**Object Destruction:**
represents explicit destruction of an

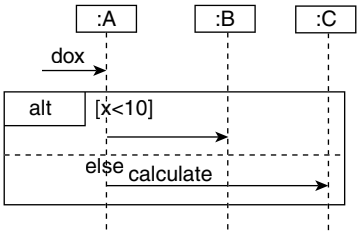



**Frames:**

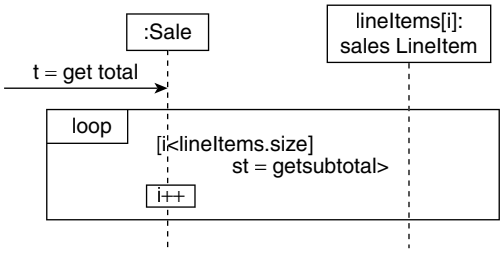
- Conditional and looping constructs, the UML uses frames.
- Frames are regions or fragments.




**Mutually Exclusive Conditional Messages:**
Alt frame is placed around the mutually exclusive alternatives.

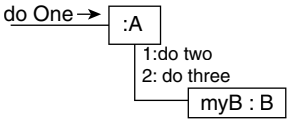



**Iteration Over a Collection:**
A common algorithm is to over all members of a collection sending the same message.
Implementation of java util. Iterator.



**Collaboration diagram:-**

Illustrate object interactions in a group or network format, in which object can be placed anywhere on the diagram.

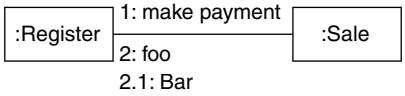



**Strength & Weakness:-**

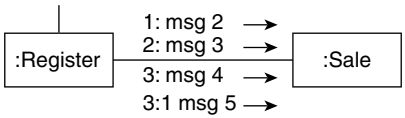
- Less notational power
- Call flow sequence is not effective
- Not excellent for documentation.
- Much more space efficient.
- We can draw a new object where even we want.

**Links:**

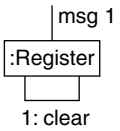
- A link is a connection path between 2 objects.
- It indicates some form of navigation and visibility between the objects is possible.



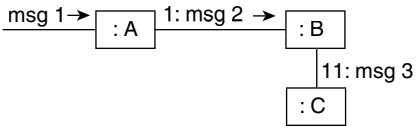

**Messages:**
Message expression and small arrow indicating the direction of the message.



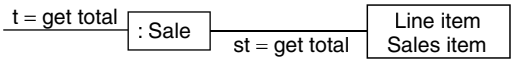

**Message to self or this:**
A message can be sent from an object to itself.



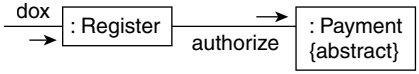
**Message number sequencing:**



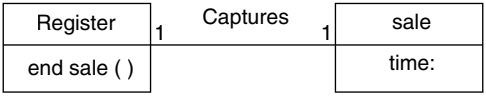

**Iteration Or Looping:**
**Over a collection:**




Polymorphic Messages & classes:




13. (b) (i) 3 Common Compartments
(1) Classifier name
(2) attributes
(3) operations
In the UP, the set of all DCDS form part of the Design model. Other parts of the Design model include UML interacts and package diagrams.

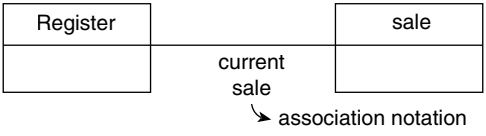



**Classifier:**

A classifier is a model element that describes behavioural and structure feature
- It can be specialized.
- UML including classes, interfaces, use cases, and actors.

**UML attributes:**

attribute text: notation, currentSale :Sale associationline : notation both together.

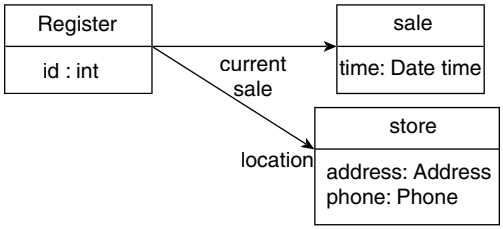



Visibility name : type multiplicity = default { Property- String

Visibility marks include + (Public), - (Private),---
A navigability arrow pointing from the source(Register) to target (Sale) object, indicating a Reg object has an attribute of one sale.
A Multiplicity at the target end, but not the source end.
Rolename only at the target end, to show the attribute name
No association name.

**Data type:-**

A data type refers to objects for which unique identity is not important.
Boolean, Date, Number, Character, String, time, Address, Color, Geometrics.

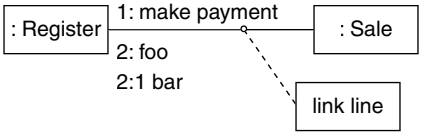



```
Public Class Register
{
Private int id;
Private Sale CurrentSale;
}
```

Keyword is a textual adanment to categorize a model element.

13. (b) (ii) **Link**: A Link is a connection path between the objects. It indicates some; form of navigation and visibility between the objects

- is possible.
- a link is an instance of an association.

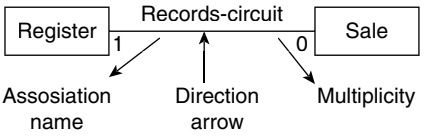



**Association:**

- Associations are needed to satisfy the information requirements of the current scenarios under development & it is aid in understanding the domain.
- Association is a relationship between a classes that indicates some meaningful and interesting connection.

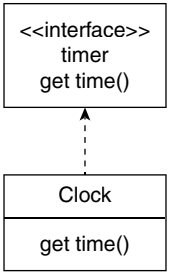
the semantic relationship b/w 2 classific

- Association for which knowledge of the relationship needs to be preserved for some duration.
- Associations derived from the common Associations List.
- In domain modeling, an association is not a statement about data flows, database foreign Key relationships, instance variables, or object connections in a software solution.
- Association represented by line.




**Inheritance:**

UML provides several ways to show inheritance implementation, providing an interface to clients, & interface dependency



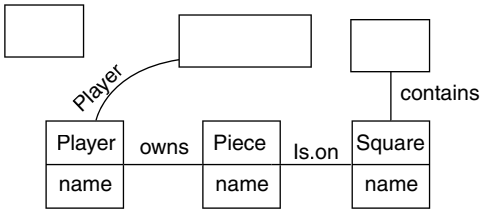

14. (a) (i) Name : creator

Problem: who creates an A?

Solution: Assign class B the responsibility to create an instance of class A if one of there is tree :

- B "contains" or compositely aggregates A
- B records A
- B Closely uses A
- B and A refer to software objects, not domain model objects.
- Board Contains squares, that's conceptual perspective, not a software one, but of course we can minor it in the Design Model so that a software Board object contains software square square objects.
- Squares will always be a part of One Board, and Board manages their creation and destruction.

- Agile modeling practice is to create parallel Complementary dynamic and static object models.
- Ignore the side issues of drawing the loop to create all 40 squares.

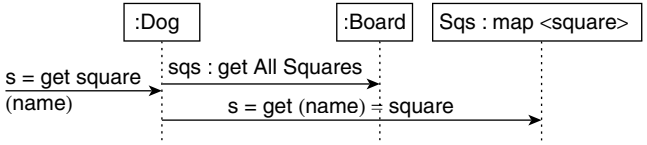


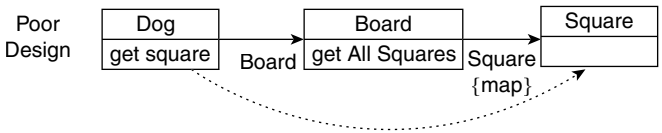

14. (a) (ii) Low Coupling:

Name : Low Coupling

Problem : How to reduce the impact of change

Solution : Assign responsibilities so that coupling remains low.




**Controller:**

Name : Controller

Problem : What 1st object beyond the UI layer receives and coordinates a system of creation:

Solution: Assign the responsibility to an object representing one of these conditions

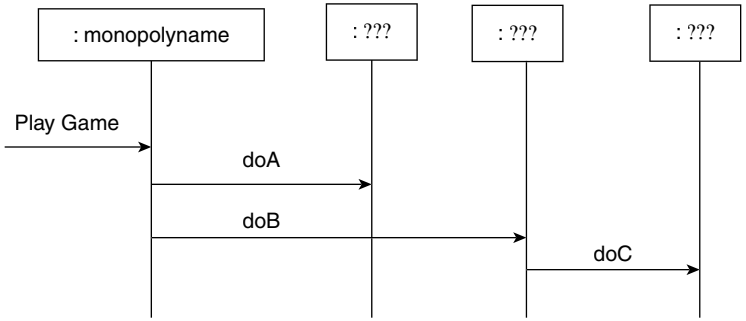
- Represents the Overall system a root object.
- Represents a use case scenario within which the system operation occurs.

**High Cohesion:**

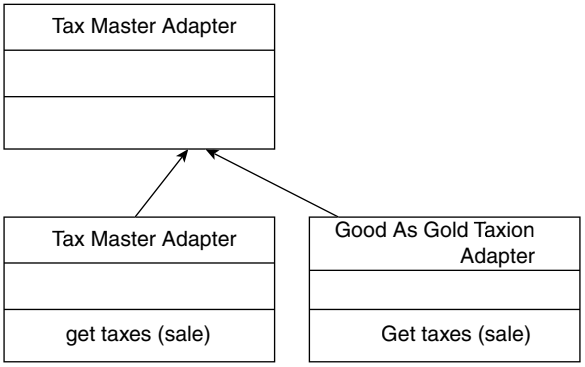
Name: High Cohesion

Problem : How to keep objects focused, understandable and manageable, and as a side effect.

Solution: Assign responsibilities so that cohesion remains high, use this to evaluates alternatives.




14. (b) Adapter:
Name : Adapter
Problem: How to resolve incompatible interfaces.
Solution: Convert the original interface of a component into another interface through an intermediate adapter object.




- A particular adapter instance will be instatiated for the chosen external service, such as SAP for accounting, and will adapt the post sale request to the external interface

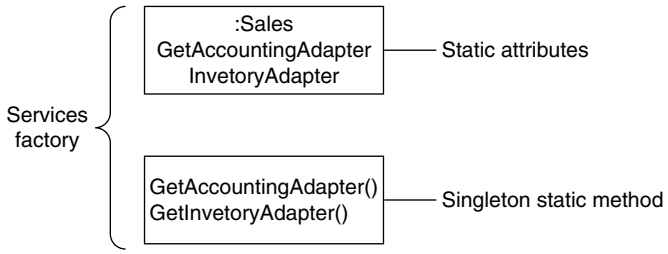
**Singleton:**

Singleton is desirable to support global Visibility or a single access point to a single instance of a class rather than some other form of visibility.

Name : Singleton

Problem: Exactly one instance of a class is allowed it is a singleton object.

**Solution:**




Eg Public class Register
{
Public void initialize()
{
-- do some work----
// accessing the singleton Factory via the get Instance call accounting Adapter = Services Factory. get Instance(). get Accounting Adapter();
-- do some work----
}
// other methods
} // end of class.

**One can write**

SingletonClass. Get Instance()
in order to Obtain visibility to the singleton instance

- Instance side methods.
- Remote enabling of instance methods.
- not always a singleton in all application contexts.

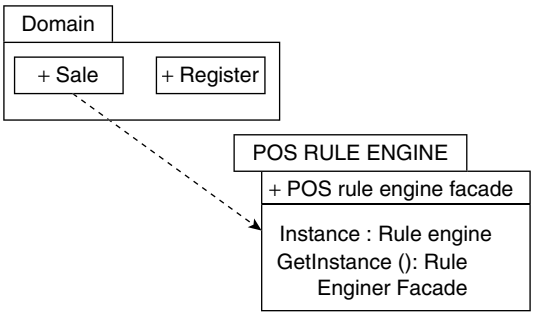
**Observer Pattern:**

Name : Observer

Problem: Different kinds of Subscribe objects are interested in the state changes or events of a publisher.

Solution : Define a "subscriber" or "listened interface.

- GUI window to refresh its display of the sale total when the total changes.
- When the sale changes its total, the sale object sends a message to a window, asking it to refresh its display.




Factory:

- Simple Factory or concrete Factory.
- is not a GOF design pattern widespread.
- Underscores another fundamental design

Principle: Design to maintain a separation of concerns.
Separate the responsibility of complex creation into cohesive helper objects.
Hide potentially complex creation logic
Name : Factory
Problem: who would be responsible for Creating Objects when there are special considerations for better cohesion?
Solution : Create a face fabrication object.

15. (a) Operation contracts we a pre-and postcondition form to describe detailed changes to objects in a domain model, as the result of a system operation.
    - Operation contracts may be part of the UP use case model detail of the System operation.
    - The inputs to the contracts are SSDS, domain model, and domain insight from experts.

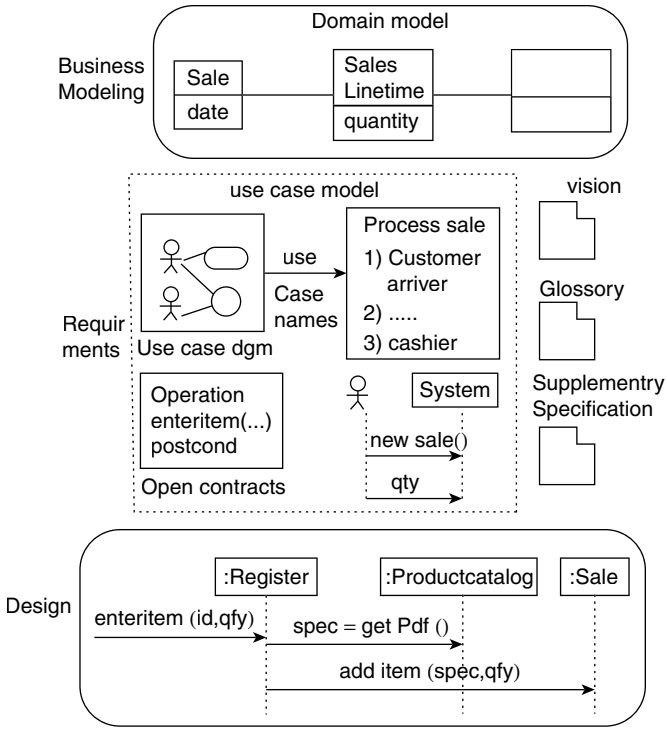
    **Sections of a contract:**

    Operation: Name of operation, and parameter
    Cross References: use cases this operation can Occur within.
    Preconditions : State of the system or Objects before execution.
    Postconditions: State of Objects after execution.

**System operation:**

Operations that the system as a black box component offers in its public interface.

**Post conditions:**




Design (overall)

Describes changes in the state of objects in the domain model.

- Post conditions are not actions to be performed during the operation rather, they are observations about the domain model objects.

**Categories:**

- Instance creation and deletion
- Attribute change of value.
- Associations to be precise, OMC links) formed and broken.

**Related to Domain Model:**

Post Conditions are expressed in the context of the domain model objects.

- What associations can be formed are analyzed.
- A contract is an excellent tool of requirements analysis or OOA that describe in great detail the changes required by a system operation.

**Writing Post condition:**

Express in Past tense
Eg A sales Line Item was created.
Eg EnterItem Post conditions:

**Instance Creation and Deletion:**

After itemid and quanting of an item have been entered, what new object shock have been created.

**Attribute Modification:**

After id & quantity of an item have been entered by cashier, what attributes of new or existing objects should have been modified.

**Associations formed and broken:**

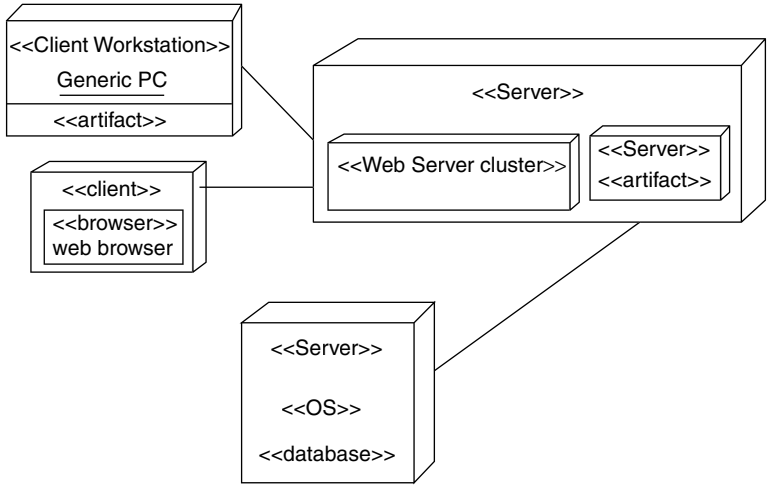
After item id & qty of an item have been entered by the cashier, what associations between new or existing objects should have been formed
Eg : Contract Col: make NewSale
Operation : Make newSale()
Cross Reference : Use cases: Process sale
Preconditions: none
Post conditions: Sale instance was created.
Eg 2 : Contract Co3: end Sale
Operation : end Sale()
Cross Reference : Use cases: Process sale
Preconditions: sale underway
Post conditions: Sale is completed.

15. (b) **Deployment Diagram**

- Communication between physical elements.
- Deployment diagrams are useful to communicate the Physical or deployment architecture.
  Device Node-Physical computing resource with processing and memory services to execute software
  Execution Environment Node (EEN)
  Software computing resource that sums within an outer node.
- OS (operating system) its software that hosts and executes pro-

grams.

- Virtual machine hosts & executing program.
- Database engine receives SQL program requests and executes them.
- Web browser.
- Workflow engine.
- a servlet container or EJB container.
- there are not official predefined UML sterotypes.
- normal connection between node is a communication path.
- A node may contain and show an artifact.
- Set of instances are shown with an signifies a class rather than an instance.

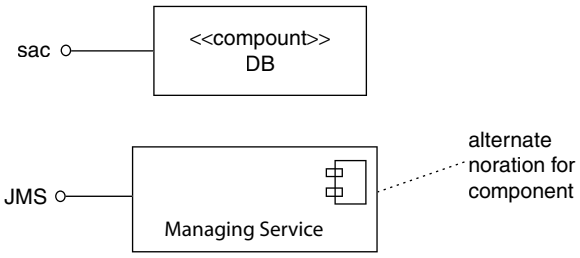



Deployment diagram

**Component Diagrams:**

Fuzzy concept in the UML, Deployment diagram because both classes and components can be used to model the same thing.

- Component represents a modular part of a system that encapsulates its contents and whole manifestation is replacable.
- UML components are a design level perspects they don't exist in the concrete software perspective, but map to concrete artifacts such as a set of files.
- A SQL database engine can be modeled as a component.
- Emphasis of component-based modeling is replaceable parts.
- It is difficult to think about or design for many small, fine grained replaceable parts.




Component diagram

# B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2011

**Sixth Semester**

**Computer Science and Engineering**

**CS 2353 — OBJECT ORIENTED ANALYSIS AND DESIGN**

**(Common to Information Technology)**

**(Regulation 2008)**

**Time: Three hours** **Maximum: 100 marks**

**Answer ALL questions.**

**PART A – (10 × 2 = 20 marks)**

1. List out any four reasons for the complexity of software.
2. What do you mean by use cases and actors?
3. Give the hint to identify the attributes of a class.
4. Define swim lane.
5. What do you mean by sequence diagram? Mention its use.
6. What do you mean by sequence number in UML? Where and for what it is used?
7. Distinguish between coupling and cohesion.
8. Write a note on Patterns.
9. Define component with an example.
10. How will you reflect the version control information in UML diagram?

**PART B – (5 × 16 = 80 marks)**

11. (a) What do you mean by Unified Process in OOAD? Explain the phases with suitable diagrams. (16)

Or

(b) By considering the Library Management system, perform the Object Oriented System Development and give the use case model for the same (use include, extend and generalization). (16)

12. (a) Explain the **relationships** that are possible among the classes in the UML representation with your own example. (16)

Or

(b) Explain the following with an example:
(i) Conceptual class diagram
(ii) Activity Diagram. (8 + 8)

13. (a) With a suitable example explain how to design a class. Give all possible representation in a class (name, attribute, visibility, methods, responsibilities). (16)

Or

(b) What do you mean by interaction diagrams? Explain them with a suitable example. (16)

14. (a) What is GRASP? Explain the design patterns and the principles used in it. (16)

Or

(b) What is design pattern? Explain the GoF design patterns. (16)

15. (a) Explain the state chart diagram with a suitable example. Also define its components and use. (16)

Or

(b) Consider the Hospital Management System application with the following requirements
(i) System should handle the in-patient, out-patient information through receptionist.
(ii) Doctors are allowed to view the patient history and give their prescription.
(iii) There should be a information system to provide the required information.
Give the state chart, component and deployment diagrams.
(6 + 6 + 4)

# Solutions

## PART A

1. The reasons for the complexity of software are
   - Nature of the problem domain
   - Complexity of process
   - Dangerous potential for flexibility in software systems
   - Characterizing behavior of discrete systems

2. Use cases are scenarios that describe how actors use the system. It is an interaction between users and a system. It captures the goal of the users and the responsibility of the system to its users. It is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system.
   An actor is someone or something that must interact with the system under development. It is a user playing a role with respect to the system. Actor initiates the process. It is represented as a stickman.

3. The guidelines for identifying attributes of classes are:
   - Attributes usually correspond to nouns followed by preposition phrases.
   - Attributes also may correspond to adjectives or adverbs.
   - Keep the class simple; state only enough attributes to define the object state.
   - Attributes are less likely to be fully described in the problem statement.
   - Omit derived attributes. They should be expressed as a method.
   - Do not carry excess identification.

4. Swim lane is a kind of package for organizing responsibility for activities within a class. It shows the actions and activities being executed by a unit, an object or a class, mostly concurrent to other actions/activities.

5. A sequence diagram in a Unified Modeling Language is a kind of interaction diagram that shows how processes operate with one another and in what order. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Also called as event diagrams, event scenarios, and timing diagrams.

   The sequence diagrams assist us in defining services that the objects must provide. These services are implemented as the methods for your objects. In a sequence diagram the events that occur between objects are drawn between the vertical object lines. An event is considered to be an action that transmits information; therefore these actions are the operations that the objects must perform.

6. Sequence diagrams show how objects in the system interact over time, but they don't show the associations between objects. They are read from top to bottom and time unfolds as we read downward. Like sequence diagrams, collaboration diagrams show how objects in the system interact, but the emphasis is on the associations. Message sequencing is still visible, by the sequence numbers on each message.

   In collaboration diagram the method call sequence is indicated by numbering technique. The number indicates how the methods are called one after another. The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization where as the collaboration diagram shows the object organization.

   If the time sequence is important then sequence diagram is used and if organization is required then collaboration diagram is used.

**7.** Coupling is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship. Coupling is important when evaluating a design because it helps us focus on an important issue in design. Coupling deals with interactions between objects or software components

   Cohesion can be defined as the interactions within a single object or software component. Cohesion reflects the "single-purposeness" of an object. Cohesion helps in designing classes that have very specific goals and clearly defined purposes. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes.

8. Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

   A pattern is instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces. Pattern solves a problem, is a proven concept, describes relationships, and has significant human component. The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems.

9. A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture is

called as a component. A component is a collection of related classes that together provide a larger set of services. A component may be

- A source code component
- A run time components
- An executable component

Components in your system might include applications, libraries, ActiveX controls, JavaBeans, daemons, and services. In the .NET environment, most of your projects will require component development.

10. The tool will make the comparison, analyze and report about the versions of the diagrams. If you change classes under version control, you can evaluate how your changes affect the model on a UML diagram.

## PART B

11. (a) The unified approach (UA) establishes a unifying and unitary framework around their works by utilizing the unified modeling language to describe, model, and document the software development process. The unified approach to software development revolves around the following processes and concepts. The processes are:
    - Use-case driven development
    - Object-oriented analysis
    - Object-oriented design
    - Incremental development and prototyping
    - Continuous testing

    The Methods and Technology include
    - Unified modeling language used for modeling.
    - Layered approach.
    - Repository for object-oriented system development patterns and frameworks.
    - Component-based development.

    **Object-oriented analysis:**

    Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how it does it. OOA Process consists of the following steps:

    1. Identify the Actors.
    2. Develop a simple business process model using UML Activity diagram.

3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

**Object-oriented design:**

Booch provides the most comprehensive object-oriented design method. Rumbaugh et al.'s and Jacobson et al.'s high-level models provide good avenues. UA combines these by utilizing Jacobson et al.'s analysis and interaction diagrams; Booch's object diagrams, and Rumbaugh et al.'s domain models. OOD Process consists of:

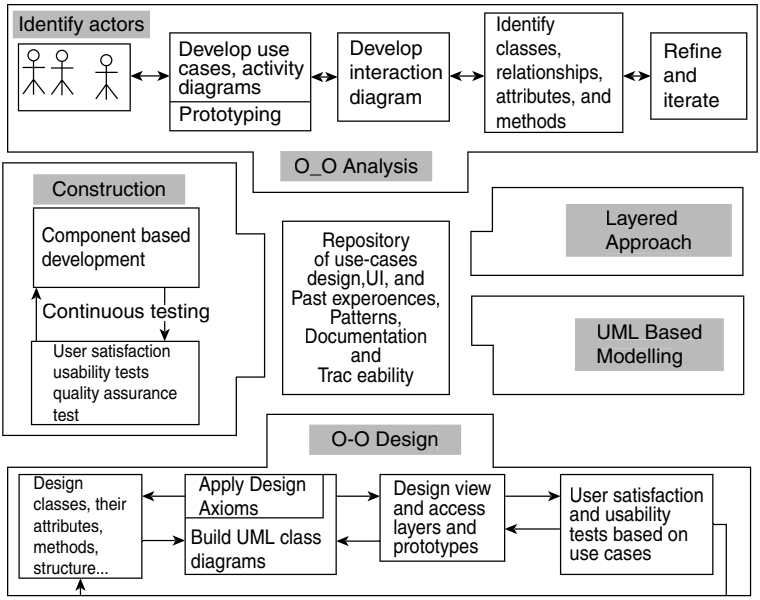
- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterate and refine the design

**3. Iterative Development and Continuous Testing:**

You must iterate and reiterate until, you are satisfied with the system. Since testing often uncovers design weaknesses, repeat the entire process, taking what you have learned and reworking your design or moving on the prototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results. During this iterative process, your prototypes will be incrementally transformed into the actual application. The UA encourages the integration of testing plans from day 1 of the project. Usage scenarios can become test scenarios; therefore, use case will drive the usability testing. Usability testing is the process in which the functionality of software is measured.

**4. Modeling based on the unified modeling language**

The unified modeling language was developed by the joint efforts of the leading object technologists Grady Booch, Ivar Jacobson, and James Rumbaugh with contributions from many others. The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch's methodology, Jacobson et al.'s use case, and Rumbaugh et al.'s object modeling technique. The UML is becoming the universal language for modeling systems. The UML is the standard notation for object-oriented modeling systems.

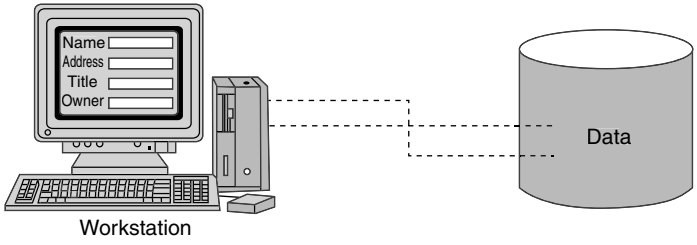



## 5. The UA Proposed Repository

The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format. Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository.

The advantage of repositories is that, if your organization has done projects in the past, objects in the repositories from those projects might be useful. Design and develop applications based on previous experience, creating additional applications will require no more than assembling components from the library.

These repositories contain all objects that have been previously defined and can be reused for putting together a new software system for a new application. The repository should be accessible to many people. Figure shows the Two-layered architecture (i.e.) interface and data.



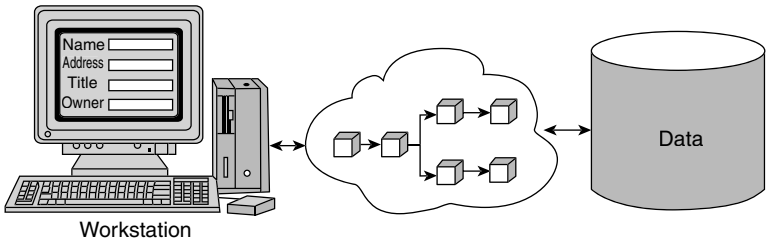
**6. The layered approach to software development:**

Most systems developed with today's CASE tools or client-server application development environments are two-layered architecture.

In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens. With every interface you create, you must re-create the business logic needed to run the screen. The routines required to access the data must exist within every screen. Any change to the business logic must be accomplished in every screen that deals with that portion of the business. This approach results in objects that are very specialized and cannot be reused easily in other projects.

A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business. This approach also isolates the business from the details of the data access

Objects are completely independent of how they are represented or stored.



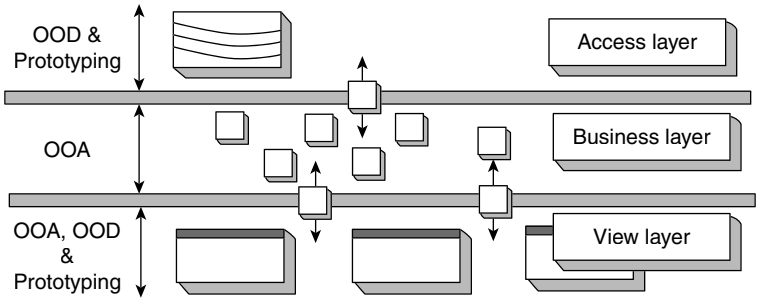

Business objects represent tangible elements of the application. They should be completely independent of how they are represented to the user or how they are physically stored. In layered approach, you are able to create objects that represent tangible elements of your business yet are completely independent of how they are represented to the user (through an interface) or how they are physically stored (in a database).

The three-layered approach consists of a view or user interface layer, a business layer, and an access layer.

**The Business Layer:** The business layer contains all the objects that represent the business (both data and behavior). The responsibilities of the business layer are very straightforward. Model the objects of the business and how they interact to accomplish the business processes. These objects should not be responsible for the following:

- Displaying details
- Data access details

**The User Interface (View) Layer:** The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The user interface layer also is called the view layer. This layer is responsible for two major aspects of the applications:

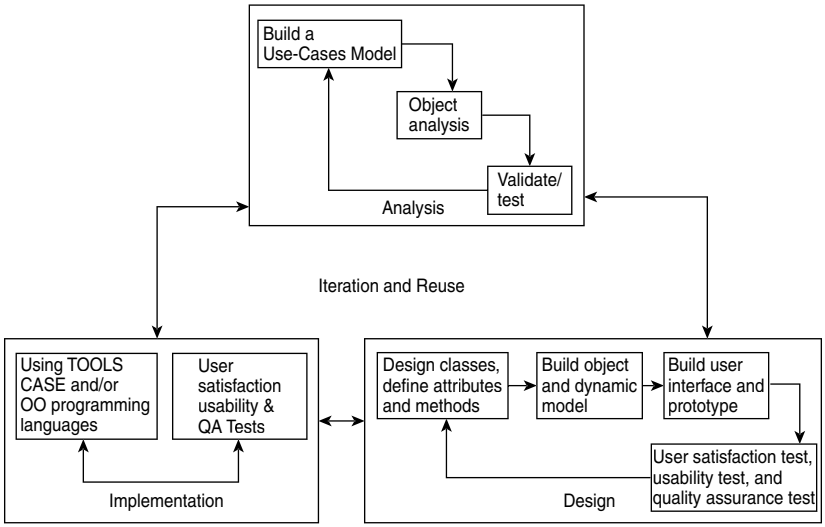



- Responding to user interaction
- Displaying business objects

**The Access Layer:** The access layer contains objects that know how to communicate with the place where the data actually reside, whether it will be a relational database, mainframe, internet or file. The two major responsibilities are

- Translate request
- Translate results

11. (b) The object-oriented software development life cycle (SDLC) consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation.

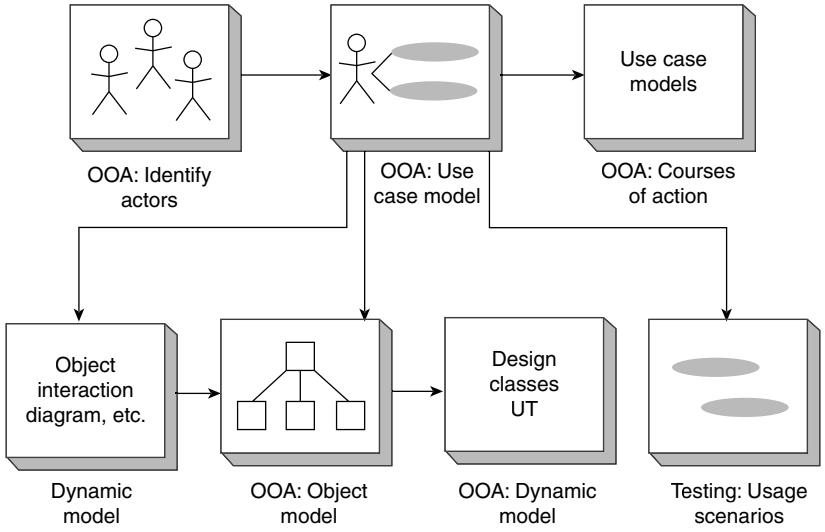



The use-case model can be employed throughout most activities of software development. The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can

become test scenarios. Object-oriented system development includes these activities:

- Object-oriented analysis-use case driven
- Object-oriented design
- Prototyping
- Component-based development
- Incremental testing

The life cycle model of Jacobson produces designs that are traceable across requirements, analysis, implementation, and testing.




**Object-Oriented Analysis-Use-Case Driven:**

The object-oriented analysis phase of software development is concerned with determining the system requirements and identifying classes and their re1ationship to other classes in the problem domain. To understand the system requirements, we need to identify the users or the actors. In object-oriented as well as traditional development, scenarios are used to help analysts understand requirements. However, these scenarios may be treated informally or not fully documented. The use-case model represents the users' view of the system or users' needs.

**Object-oriented design:**

Object-oriented development is highly incremental. First, build the object model based on objects and their relationships, then iterate and refine the model:

- Design and refine classes.
- Design and refine attributes.
- Design and refine methods.
- Design and refine structures.
- Design and refine associations.

A few guidelines to use in your object-oriented design:

- Reuse, rather than build, a new class. Know the existing classes.
- Design a large number of simple classes, rather than a small number of complex classes.
- Design methods.
- Critique what you have proposed. If possible, go back and refine the classes.

**Prototyping:**

A prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. A prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system. It also can give users a chance to comment on the usability and usefulness of the user interface design and lets you assess the fit between the software tools selected, the functional specification, and the user needs. Prototyping define the use cases, and it actually makes use-case modeling much easier. Prototypes have been categorized in various ways.

- Horizontal Prototype
- Vertical Prototype
- Analysis Prototype
- Domain prototype

A **horizontal prototype** is a simulation of the interface 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.

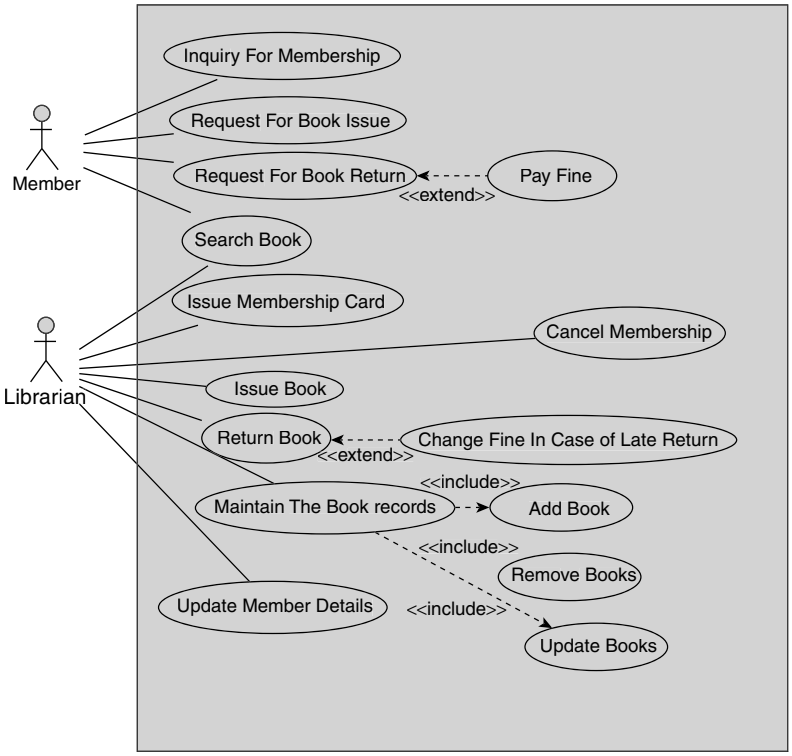
A **vertical prototype** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth. Prototypes are a hybrid between horizontal and vertical: The major portions of the interface are established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

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,

however, and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code.

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 members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product.

**Example:** Library management system




12. (a) In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements. Relationships in class diagrams show the interaction between classes and classifiers. Such relationships indicate the classifiers that are associated with each other, those that are generalizations and realizations, and those that have dependencies on other classes and classifiers.

The following topics describe the relationships that you can use in class diagrams:

- **Abstraction relationships:** An abstraction relationship is a dependency between model elements that represents the same concept at different levels of abstraction or from different viewpoints. Abstraction relationships to a model in several diagrams, including use-case, class, and component diagrams can be added.
- **Aggregation relationships:** In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier.
- **Association relationships:** In UML models, an association is a relationship between two classifiers, such as classes or use cases, which describes the reasons for the relationship and the rules that govern the relationship.
- **Association classes:** In UML diagrams, an association class is a class that is part of an association relationship between two other classes.
- **Binding relationships:** In UML models, a binding relationship is a relationship that assigns values to template parameters and generates a new model element from the template.
- **Composition association relationships:** A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier.
- **Dependency relationships:** In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier. Dependency relationships in class diagrams, component diagrams, deployment diagrams, and use-case diagrams to indicate that a change to the supplier might require a change to the client can be used.
- **Directed association relationships:** In UML models, directed association relationships are associations that are navigable in only one direction.
- **Element import relationships:** In UML diagrams, an element import relationship identifies a model element in another package, and allows the element in the other package to be referenced by using its name without a qualifier.
- **Generalization relationships:** In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.

- **Interface realization relationships:** In UML diagrams, an interface realization relationship is a specialized type of implementation relationship between a classifier and a provided interface. The interface realization relationship specifies that the realizing classifier must conform to the contract that the provided interface specifies.
- **Instantiation relationships:** In UML diagrams, an instantiation relationship is a type of usage dependency between classifiers that indicates that the operations in one classifier create instances of the other classifier.
- **Package import relationship:** In UML diagrams, a package import relationship allows other namespaces to use unqualified names to refer to package members.
- **Realization relationships:** In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Several clients can realize the behavior of a single supplier. Realization relationships in class diagrams and component diagrams can also be used.
- **Usage relationships:** In UML modeling, a usage relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation.

12. (b) (i) **Conceptual class diagram:**

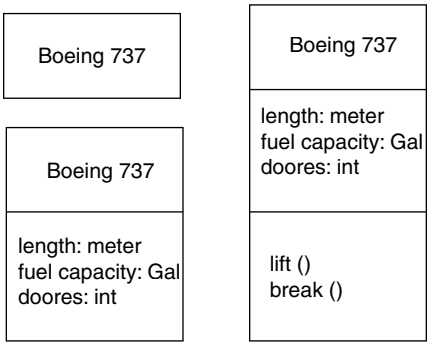
The UML class diagram, also referred to as object modeling, is the main static analysis diagram. These diagrams show the static structure of the model. A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents. Class diagrams do not show temporal information, which is required in dynamic modeling. Object modeling is the process by which the logical objects in the real world are represented by the actual objects in the program.

The main task of object modeling is to graphically show what each object will do in the problem domain, describe the structure and the relationships among objects by visual notation, and determine what behaviors fall within and outside the problem domain.

**Class Notation: Static Structure**

A class is drawn as a rectangle with three components separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes,

are in the middle compartment, and the bottom compartment holds a list of operations.



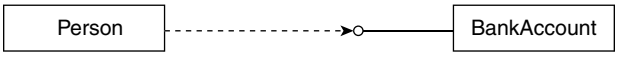

Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it. The class name and other properties should be displayed in up to three sections.

**Object Diagram:**

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.

**Class Interface Notation**

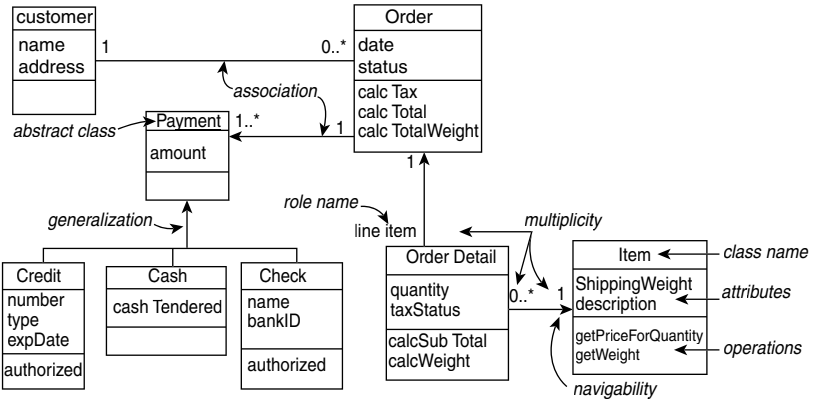
Class interface notation is used to describe the externally visible behavior of a class. Identifying class interfaces is a design activity of object-oriented system development. The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations. Example: A Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Fig with UML class interface notation.



**Binary Association Notation**

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. The association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name. The end of an association, where it connects to a class, is called the association role.

**Example:**




**(ii) Activity Diagram:**

An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. An activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. It can also be used to represent a class's method implementation.

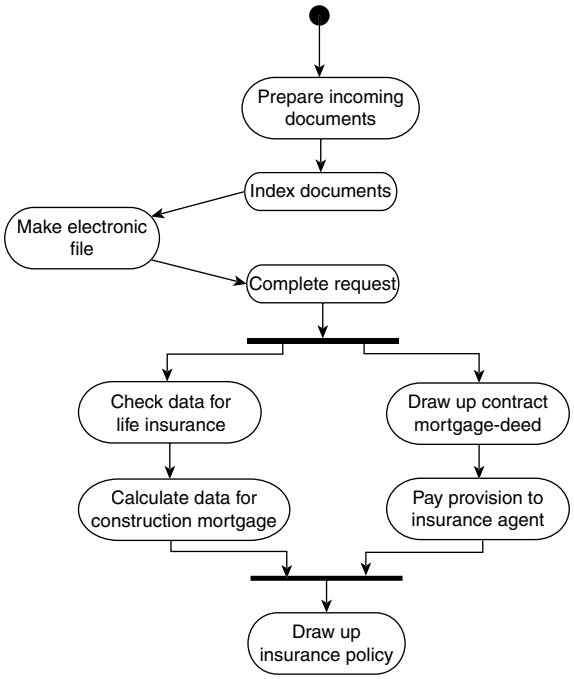
An activity model is similar to a state chart diagram, where a token, shown by a black dot, represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation.

An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "wait state," a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result

of an activity by another object. The two states are wait state and activity state. More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path.

**Example:** An activity diagram for processing mortgage requests




13. (a) Object-oriented design requires taking the object identified during object-oriented analysis and designing classes to represent them. As a class designer, we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other classes.

**Class visibility: Designing well-defined public, private and protected protocols**

In designing methods or attributes for classes, we are confronted with two problems. One is the protocol or interface to the class operations and its visibility and the other is how it is implemented. The class's protocol or the messages that a class understands, can be hidden from other objects (private protocol) or made available to other objects (public protocol). Public protocols define the functionality and external messages of an object. Private protocols define the implementation of an object.

A class might have a set of methods that it uses only internally, messages to itself. This private protocol of the class, includes messages that normally should not be sent from other objects. Here only the class itself can use the methods.The public protocol defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes. If the methods or attributes can be used by the class itself (or its subclasses) a protected protocol can be used. Here subclasses can used the method in addition to the class itself. The lack of well-designed protocol can manifest itself as encapsulation leakage. It happens when details about a class's internal implementation are disclosed through the interface.

**Designing classes: Refining attributes**

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute is enough. But in the design phase, detailed information must be added to the model. The 3 basic types of attributes are:

- Single-value attributes
- Multiplicity or multivalue attributes
- Reference to another object or instance connection

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes. Single value attribute has only one value or state. E.g.: Name, address, salary. Multiplicity or multivalue attribute can have a collection of many values at any time. E.g.: If we want to keep tact of the names of people who have called a customer support line for help. Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities. E.g.: A person may have one or more bank accounts. A person has zero to many instance

connections to Accounts. Similarly, an Account can be assigned to one or more persons(i.e.) joint account. So an Account has zero to many instance connection to Persons.

**Designing methods and protocols:**

A class can provide several types of methods:

- Constructor: Method that creates instances (objects) of the class
- Destructor: The method that destroys instances
- Conversion Method: The method that converts a value from one unit of measure to another.
- Copy Method: The method that copies the contents of one instance to another instance
- Attribute set: The method that sets the values of one or more attributes
- Attribute get: The method that returns the values of one or more attributes
- I/O methods: The methods that provide or receive data to or from a device
- Domain specific: The method specific to the application.

**Packages and Managing Classes**

A package groups and manages the modeling elements, such as classes, their associations and their structures. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model elements. A package provides a hierarchy of different system components and can reference other packages. Classes can be packaged based on the services they provide or grouped into the business classes, access classes and view classes.

13. (b) Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done. Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. There are two kinds of interaction models are:

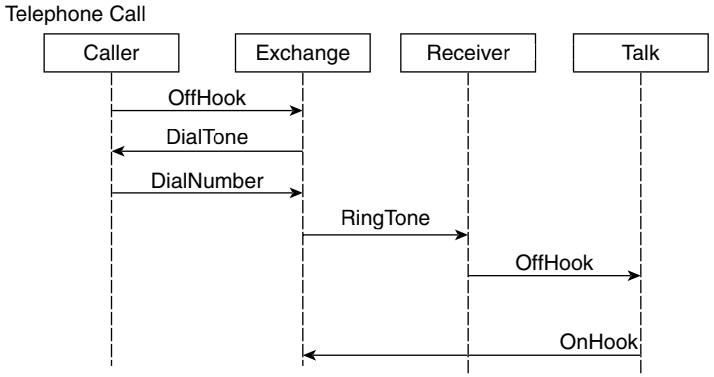
- Sequence diagrams
- Collaboration diagrams.

**Sequence diagrams:**

UML Sequence Diagram Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. It shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they

exchange, arranged in a time sequence. A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction.

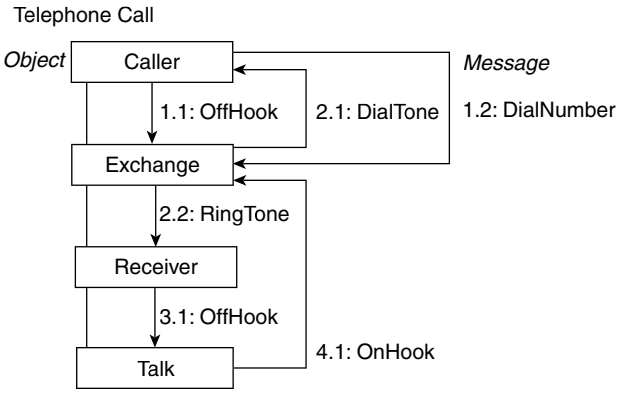
An object is shown as a box at the top of a dashed vertical line. A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles. A sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.




Each message is represented by an arrow between the lifelines of two objects. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary.

**Collaboration diagrams:**

A collaboration diagram represents collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figures. Arrows indicate the message sent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages. A collaboration diagram provides several numbering schemes.




14. (a) General Responsibility Assignment Software Patterns (GRASP), consists of guidelines for assigning responsibility to classes and objects in object-oriented design. GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.The different patterns and principles used in GRASP are:
    - Controller
    - Creator
    - Indirection
    - Information Expert
    - High Cohesion
    - Low Coupling
    - Polymorphism
    - Protected Variations
    - Pure Fabrication

    Larman states that, the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology. Thus, GRASP is really a mental toolset, a learning aid to help in the design of object-oriented software.

    Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

    The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications. It provides a scheme for refining the subsystems or components of a software system or

the relationships among them. Design patterns are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context. The main idea is to provide documentation to help categorize and communicate about solutions to recurring problems.

Design patterns are more abstract than frameworks. Design patterns are smaller architectural elements than frameworks. Design patterns are less specialized than frameworks.

**Controller:** The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event. A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case

**Creator:** Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes. Creator pattern is responsible for creating an object of class.

**High Cohesion:** It is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change.

**Indirection:** The Indirection pattern supports low coupling between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.

**Information Expert:** Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on. Using the principle of Information Expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. It

will lead to placing the responsibility on the class with the most information required to fulfill it.

**Low Coupling:** Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

- low dependency between classes,
- low impact in a class of changes in other classes,
- high reuse potential.

**Polymorphism:** According to Polymorphism, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

**Protected Variations:** The Protected Variations pattern protects elements from the variations on other elements by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

**Pure Fabrication:** A Pure Fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived.

14. (b) Design patterns are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context. A design pattern provides a scheme for refining the subsystems or components of a software system or the relationships among them. Design patterns are documented by writing essays in a fairly well-defined form.

Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

The GoF patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral.

- Creational Patterns: Abstract object instantiation
- Structural Patterns: Compose and organize objects into larger structures
- Behavioral Patterns: Algorithms, interactions and control flow between objects.

**Creational Patterns:**

1. **Abstract Factory**: Creates an instance of several families of classes. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Builder**: Separates object construction from its representation. Separate the construction of a complex object from its representation so that the same construction processes can create different representations.
3. **Factory Method**: Creates an instance of several derived classes. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
4. **Prototype**: A fully initialized instance to be copied or cloned. Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
5. **Singleton:** A class of which only a single instance can exist. Ensure a class only has one instance, and provide a global point of access to it.

**Structural Patterns:**

1. **Adapter**: Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
2. **Bridge**: Separates an object's interface from its implementation. Decouple an abstraction from its implementation so that the two can vary independently.
3. **Composite:** A tree structure of simple and composite objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
4. **Decorator**: Add responsibilities to objects dynamically. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
5. **Facade:** A single class that represents an entire subsystem. Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.
6. **Flyweight:** A fine-grained instance used for efficient sharing. Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it's indistinguishable from an instance of the object that's not shared.

7. **Proxy:** An object representing another object. Provide a surrogate or placeholder for another object to control access to it.

**Behavioral Patterns:**

1. **Chain of Resp**: A way of passing a request between a chain of objects. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
2. **Command:** Encapsulate a command request as an object. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
3. **Interpreter**: A way to include language elements in a program. Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
4. **Iterator**: Sequentially access the elements of a collection. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
5. **Mediator**: Defines simplified communication between classes. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
6. **Memento**: Capture and restore an object's internal state. Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
7. **Observer**: A way of notifying change to a number of classes. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
8. **State**: Alter an object's behavior when its state changes. Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
9. **Strategy**: Encapsulates an algorithm inside a class. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
10. **Template:** Defer the exact steps of an algorithm to a subclass. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses

redefine certain steps of an algorithm without changing the algorithm's structure.

11. **Visitor:** Defines a new operation to a class without change. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

15. (a) A state chart diagram shows the sequence of states that an object goes through during its life in response to outside stimuli and messages. The state is the set of values that describes an object at a specific point in time and is represented by state symbols and the transitions are represented by arrows connecting the state symbols. A state chart diagram may contain sub-diagrams. A state diagram represents the state of the method execution, and the activities in the diagram represent the activities of the object that performs the method. The purpose of the state diagram is to understand the algorithm involved in performing a method.

A state chart diagram is similar to a Petri net diagram, where a token, shown by a solid black dot represents an activity symbol. When an activity symbol appears within a state symbol, it indicates the execution of an operation. Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in a different phase. An outgoing solid arrow attached to a state chart symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but conditions that depend on the result of the activity or other values may be included. An event occurs at the instant in time when the value is changed. A message is data passed from one object to another.

A state is represented as a rounded box, which may contain one or more compartments. The compartments are all optional. The name compartment and the internal transition compartment are two such compartments:

- The **name compartment** holds the optional name of the state. States without names are "anonymous" and all are distinct. Do not show the same named state twice in the same diagram, since it will be very confusing.
- The **internal transition compartment** holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing states

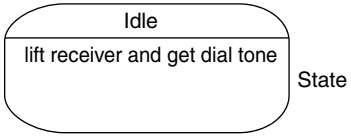
The syntax is
event-name argument-list / action-expression;
Example:
help / display help.

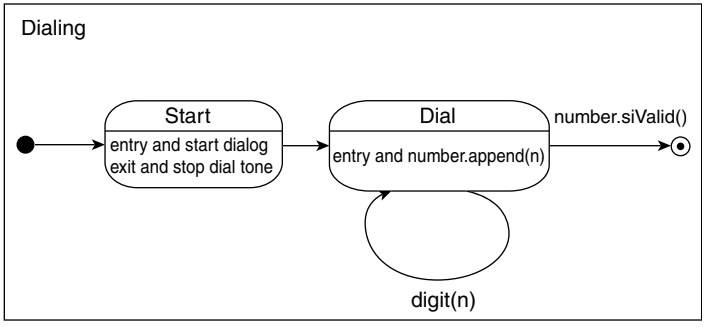
Two special events are **entry** and **exit,** which are reserved words and cannot be used for event names. The state chart supports nested state machines; to activate a sub-state machine use the keyword **do**. If this state is entered, after the entry action is completed, the nested state machine will be executed with its initial state. When the nested state machine reaches its final state, it will exit the action of the current state, and the current state will be considered completed.

An initial state is shown as a small dot, and the transition from the initial state may be labeled with the event that creates the objects; otherwise, it is unlabeled. If unlabeled, it represents any transition to the enclosing state.

A final state is shown as a circle surrounding a small dot, a bull's-eye. This represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition. The transition can be simple or complex. A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain actions when a specific event occurs; if the specified.

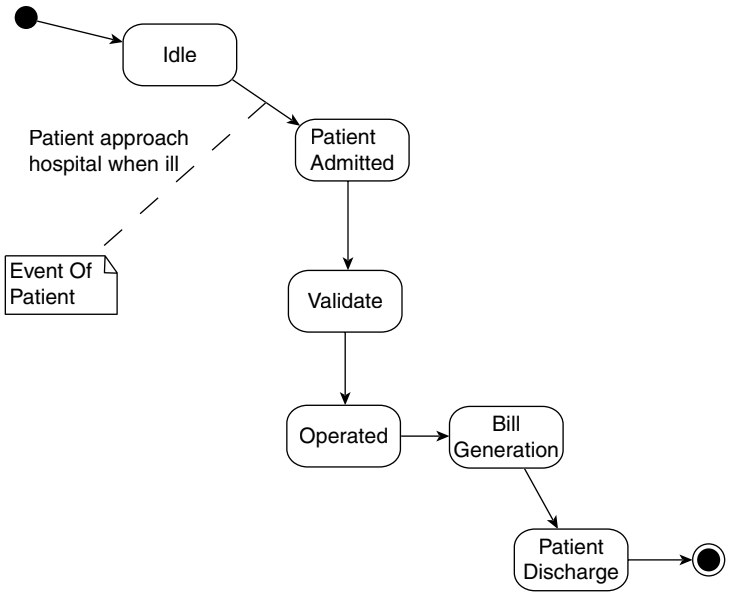
**Example:**



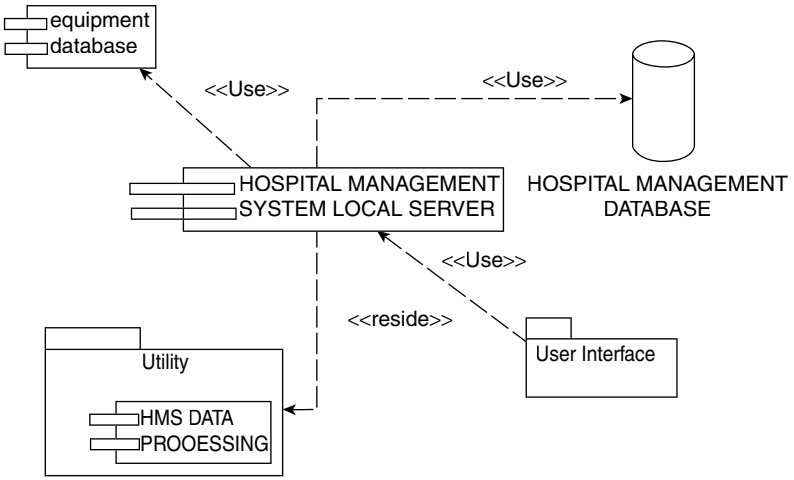




A simple state idle and a nested state. The dialing state contains sub-states, which consist of start and dial states.

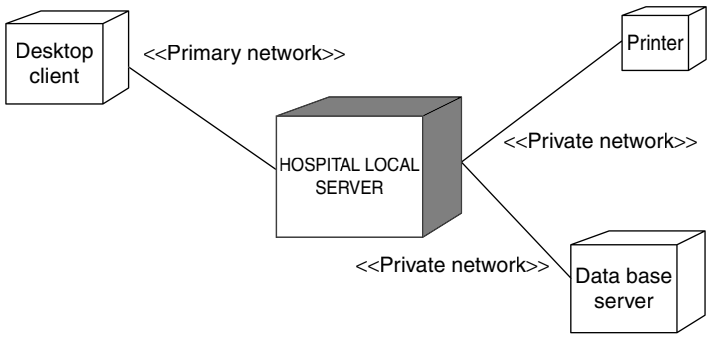
15. (b) **State chart diagram:**




**Component diagram:**



**Deployment diagram:**



# ANNA UNIVERSITY

## B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011

**Sixth Semester**

**Computer Science and Engineering**

**CS 2353 — OBJECT ORIENTED ANALYSIS AND DESIGN**

**(Common to Information Technology)**

**(Regulation 2008)**

**Time: Three hours** **Maximum: 100 marks**

**Answer ALL questions**

**PART A – (10 × 2 = 20 marks)**

1. What is Object Oriented Analysis and Design?
2. Define the inception step.
3. What is a Domain Model?
4. Define Aggregation and Composition.
5. What is the use of System sequence diagram?
6. List the relationships used in class diagram.
7. What is Design Pattern?
8. Define Coupling.
9. What is the use of operation contracts?
10. Give the meaning of Event, State and transition.

**PART B – (5 × 16 = 80 marks)**

11. (a) Briefly explain the different phases of Unified Process.

Or

(b) Explain with an example, how use case modeling is used to describe functional requirements. Identify the actors, scenario and use cases for the example.

12. (a) Describe the strategies used to identify conceptual classes. Describe the steps to create a domain model used for representing conceptual classes.

Or

(b) Explain about activity diagram with an example.

13. (a) Illustrate with an example, the relationship between sequence diagram and use cases.

Or

(b) Explain with an example Interaction diagram

14. (a) Explain about GRASP Patterns.

Or

(b) Write short notes on adapter, singleton, factory and observer patterns.

15. (a) Explain about implementation model (mapping design to code).

Or

(b) Discuss about UML deployment and component diagrams. Draw the diagrams for a banking application.

# Solutions

## PART A

1. Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state, and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

   Object-oriented analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system. It is like other kinds of system analysis, clarifies and documents the requirements of a computer system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOD is viewed as an extension of analysis, more so than a distinct activity OOA focuses on what the system does, OOD on how the system does it.

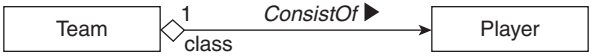
2. Inception is a feasibility phase, where just enough investigation is done to support a decision to continue or stop. Inception phase is the initial short step to envision the product scope, vision, and business case. The intent of inception is to establish some initial common vision:
   - for the objectives of the project,
   - determine if it is feasible, and
   - decide if it is worth some serious investigation in elaboration

   It may include the:
   - first requirements workshop
   - planning for the first iteration

3. Domain Model is a class diagram with conceptual classes in the problem domain Domain model maps real world object into the domain object model. It is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain.

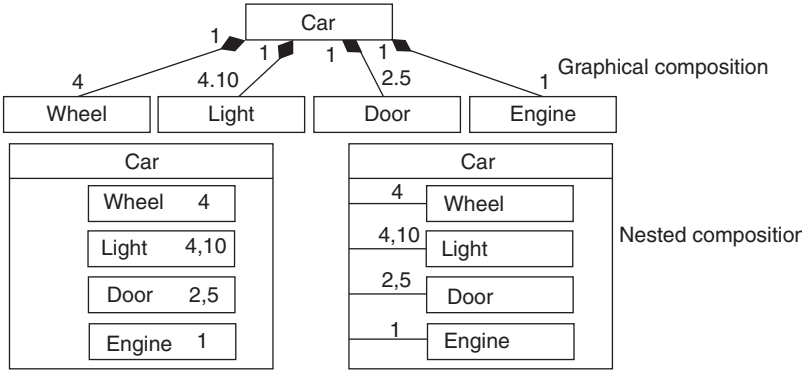
   The domain model also identifies the relationships among all the entities within the scope of the problem domain, and commonly identifies their attributes. A domain model that encapsulates methods within the entities is more properly associated with object oriented models. The domain model provides a structural view of the domain that can be complemented by other dynamic views, such as use case models.

4. Aggregation is a form of association. A hollow diamond is attached at the end of the path to indicate aggregation.




A-part-of relationship, also called aggregation, represents the situation where a class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently.

Composition, also known as the a-part-of, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a part-whole relationship. The UML notation for composition is a solid diamond at the end of a path. The UML provides a graphically nested form that is more convenient for showing composition. The figure below represents the different ways to show composition




5. Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. It shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence. A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction.

6. The relationships used in class diagrams are:
   - Abstraction relationships
   - Aggregation relationships
   - Association relationships
   - Association classes

- Binding relationships
- Composition association relationships
- Dependency relationships
- Directed association relationships
- Element import relationships
- Generalization relationships
- Interface realization relationships
- Instantiation relationships
- Package import relationship
- Realization relationships
- Usage relationships

7. Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

8. Coupling is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship. Coupling is important when evaluating a design because it helps us focus on an important issue in design. Coupling deals with interactions between objects or software components

9. Operations contracts use a pre- and post- condition form to describe detailed changes to objects in a domain model, as the result of a system operation. Operation contracts may be considered part of the Unified Process Use-Case Model because they provide more analysis detail on the effect of the system operations implied in the use cases. Operation contracts may be defined for system operations.

   An operation contract describes the changes in the state of the system when a system operation is invoked. Contracts describe detailed system behavior in terms of state changes to objects in the Domain Model, after a system operation has executed. A domain model can be used to help generate an operation contract.

10. A state chart diagram shows the sequence of states that an object goes through during its life in response to outside stimuli and messages. The state is the set of values that describes an object at a specific point in time and is represented by state symbols and the transitions are represented by arrows connecting the state symbols.

    In a sequence diagram the events that occur between objects are drawn between the vertical object lines. An event is considered to be an action

that transmits information; therefore these actions are the operations that the objects must perform. The OMT state transition diagram is a network of states and events. Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.

## PART B

11. (a) The unified approach (UA) is a methodology for software development. The UA, based on methodologies by Booch, Rumbaugh, Jacobson, and others, tries to combine the best practices, processes, and guidelines.

UML is a set of notations and conventions used to describe and model an application. The heart of the UA is Jacobson's use case. The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs.

The main advantage of an object-oriented system is that the class tree is dynamic and can grow. The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams**.**

The UA consists of the following concepts:

- Use-case driven development.
- Utilizing the unified modeling language for modeling.
- Object-oriented analysis (utilizing use cases and object modeling).
- Object-oriented design.
- Repositories of reusable classes and maximum reuse.
- The layered approach.
- Incremental development and prototyping.
- Continuous testing.

The UA allows iterative development by allowing you to go back and forth between the design and the modeling or analysis phases. It makes backtracking very easy and departs from the linear waterfall process, which allows no form of back tracking.

**1. Object-Oriented Analysis:**

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. It concentrates on describing what the system does rather than how it does it. Separating

the behavior of a system from the way it is implemented require viewing the system from the user's perspective rather than that of the machine. OOA process consists of the following steps:

- Identify the Actors.
- Develop a simple business process model using UML Activity diagram.
- Develop the Use Case.
- Develop interaction diagrams.
- Identify classes.

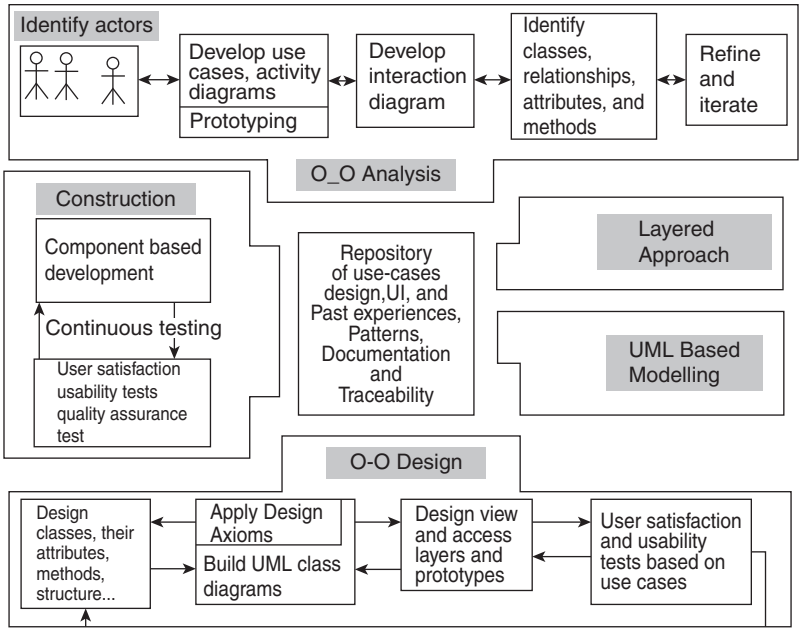
**2. Object-Oriented Design:**

Booch, p rovides the most comprehensive object-oriented design method. Ironically, Rumbaugh and Jacobson high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson analysis and interaction diagrams, Booch's object diagrams, and Rumbaugh domain models. Jacobson life cycle model, can produce designs that are traceable across requirements, analysis, design, coding, and testing. OOD Process consists of:

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms.
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterated and refine the design

**3. Iterative Development and Continuous Testing:**

You must iterate and reiterate until, you are satisfied with the system. Since testing often uncovers design weaknesses, repeat the entire process, taking what you have learned and reworking your design or moving on the prototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results. During this iterative process, your prototypes will be incrementally transformed into the actual application. The UA encourages the integration of testing plans from day 1 of the project. Usage scenarios can become test scenarios; therefore, use case will drive the usability testing. Usability testing is the process in which the functionality of software is measured.



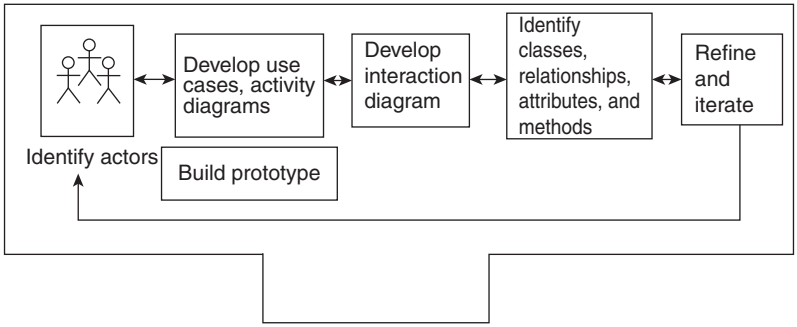

11. (b) The object-oriented analysis (OOA) phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The actors are external factors that interact with the system; use cases are scenarios that describe how actors use the system. The use cases identified here will be involved throughout the development process.

The OOA process consists of the following steps:

1. Identify the actors:
   - Who is using the system?
   - Or, in the case of a new system, who will be using the system?
2. Develop a simple business process model using UML activity diagram.
3. Develop the use case:
   - What are the users doing with the system?
   - Or, in case of the new system, what will users be doing with the system?
   - Use cases provide us with comprehensive documentation of the system under study.
4. Prepare interaction diagrams:
   - Determine the sequence.
   - Develop collaboration diagrams.

5. Classification-develop a static UML class diagram:
   - Identify classes.
   - Identify relationships.
   - Identify attributes.
   - Identify methods.
6. Iterate and refine: If needed, repeat the preceding steps.

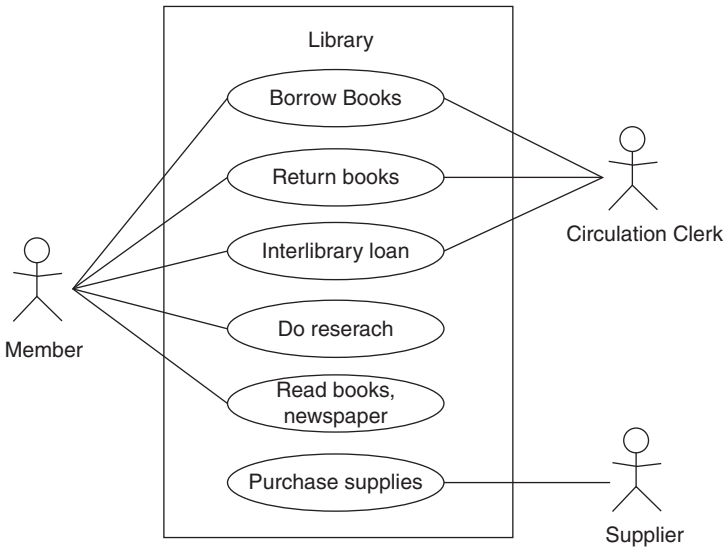
The object-oriented analysis process in the Unified Approach (UA)

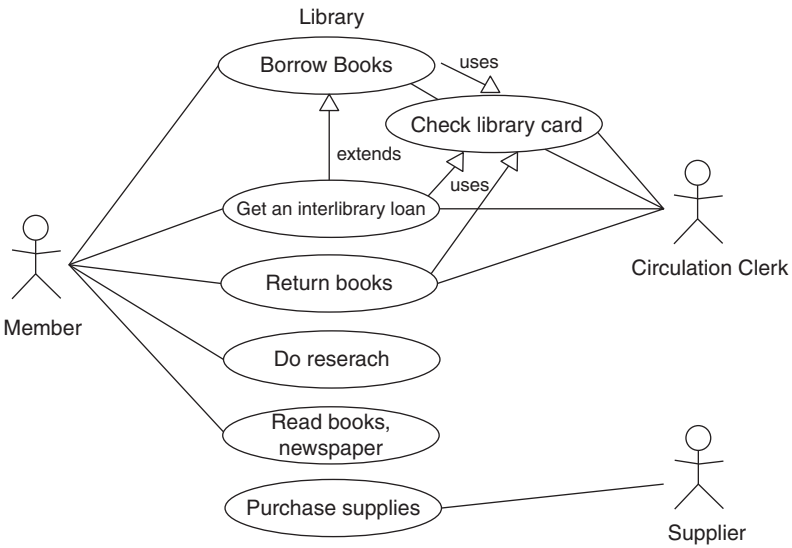



Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems requirements. A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users.

A use-case model can be developed by talking to typical users and discussing the various things they might want to do with the application being prepared. Each use or scenario represents what the user wants to do. Each use case must have a name and short textual description.. Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers. The use-case model expresses what the business or application will do and not how.

A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system. An actor is a user playing a role with respect to the system. When dealing with actors, it is important to think about roles rather than just people and their job titles. Actors carry out the use cases. A single actor may perform many use cases. A use case must help the actor to perform a task that has some identifiable value.



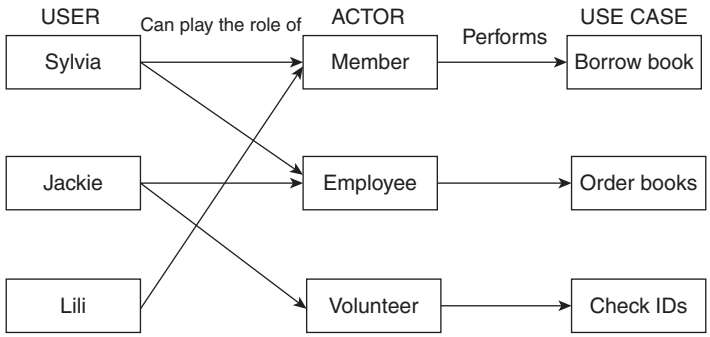

The use-case diagram depicts the extends and uses relationships. A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system.

Identifying the actors is as important as identifying classes, structures, associations, attributes, and behavior. Actor represents the role a user plays with respect to the system. When dealing with actors, it

is important to think about roles rather than people or job titles. A user may play more than one role.

For example, a member of a public library also may play the role of volunteer at the help desk in the library.

The difference between users and actors.




12. (a) A domain model is a representation of real-world conceptual classes, not of software components. Domain modeling is a technique used to understand the project problem description and to translate the requirements of that project into software components of a solution. The software components are commonly implemented in an object oriented programming language.

A domain model contains conceptual classes, associations between conceptual classes, and attributes of a conceptual class. A conceptual class is an idea, thing, or object. Domain models are the initial artifacts created in Object-Oriented-Analysis. An object has identity, state and behavior. An object can be related to other objects and complex. A class is a description of a set of objects that share the same attributes, operations/responsibilities, relationships and semantics. Identifying objects in the problem domain is used to identify conceptual classes in the problem domain.

**Identifying Conceptual Classes**

1. Modify or reuse an existing model.
2. Use a conceptual class category list.
3. Identify noun phrases.

Some categories are:

- physical or tangible objects
- specification, designs, or description of things
- transaction
- roles of people
- container of things

- events
- rules and policy
- records

**Using noun phrases**

Conceptual class can be identified by studying the use case looking for relevant noun phrases. Some heuristics for identification are:

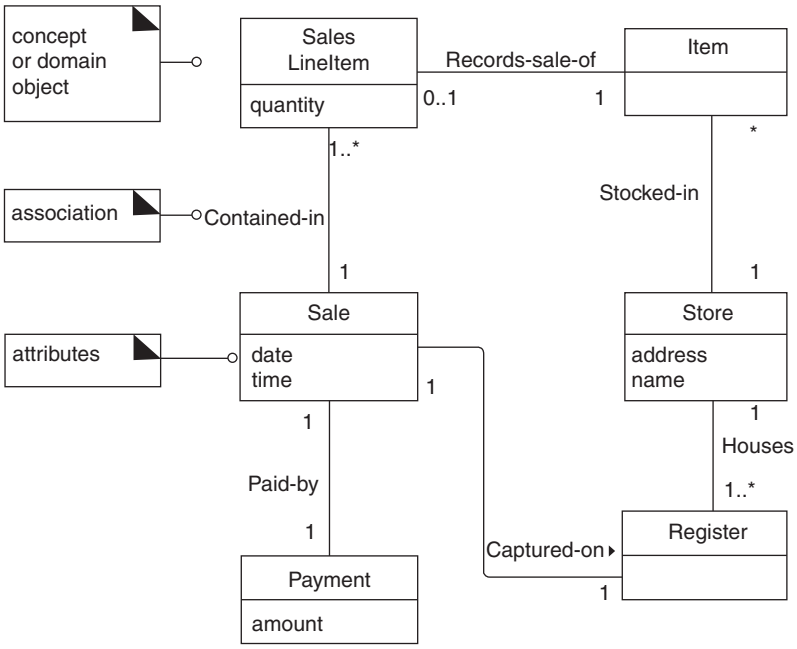
- terms that developers or users need to clarify in order to understand the use case
- recurring nouns
- real world entities that the system needs to track
- real world activities
- data sources or sinks

Using UML notation, a domain model is illustrated with a set of class diagrams in which no operations are defined. It may show:

- domain objects or conceptual classes
- associations between conceptual classes
- attributes of conceptual classes

e.g. a partial domain model

- conceptual class of Payment and Sale are significant in this domain,
- Payment is related to a Sale in a way that is meaningful to note
- Sale has a date and time.



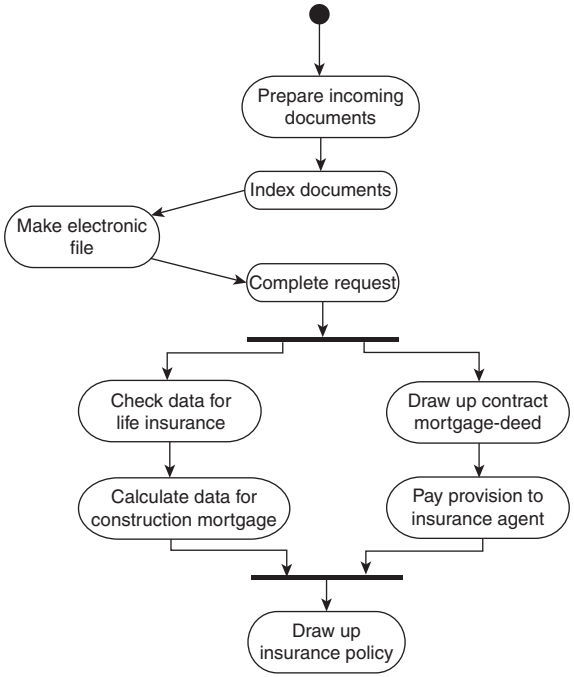
12. (b) An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. An activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. It can also be used to represent a class's method implementation.

An activity model is similar to a state chart diagram, where a token, shown by a black dot, represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation.

An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "wait state," a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result of an activity by another object. The two states are wait state and activity state. More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path.

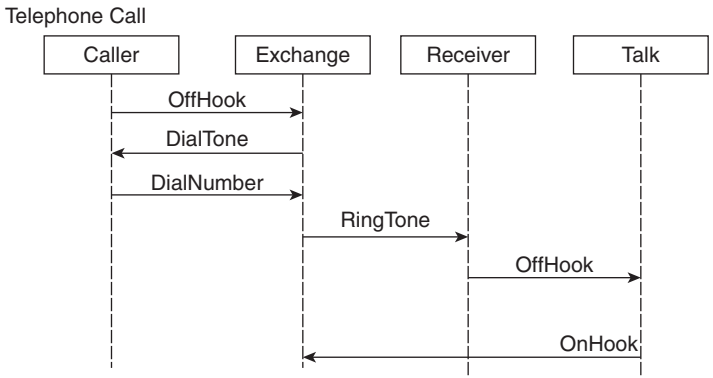
**Example:** An activity diagram for processing mortgage requests




13. (a) UML Sequence Diagram Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. It shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence. A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction.

An object is shown as a box at the top of a dashed vertical line. A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles. A sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

An example of a sequence diagram.

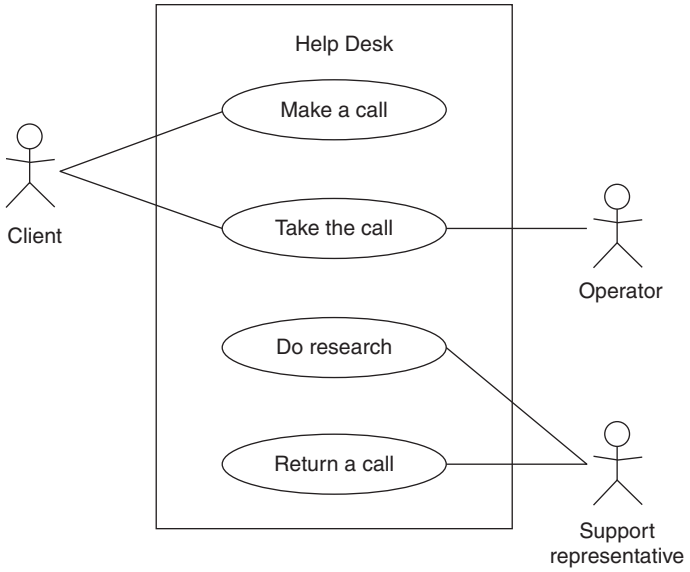



Each message is represented by an arrow between the lifelines of two objects. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary.

Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems requirements. A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users.

A use-case model can be developed by talking to typical users and discussing the various things they might want to do with the application being prepared. Each use or scenario represents what the user wants to do. Each use case must have a name and short textual description.. Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers. The use-case model expresses what the business or application will do and not how.

A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system. An actor is a user playing a role with respect to the system. When dealing with actors, it is important to think about roles rather than just people and their job titles. Actors carry out the use cases. A single actor may perform many use cases. A use case must help the actor to perform a task that has some identifiable value.



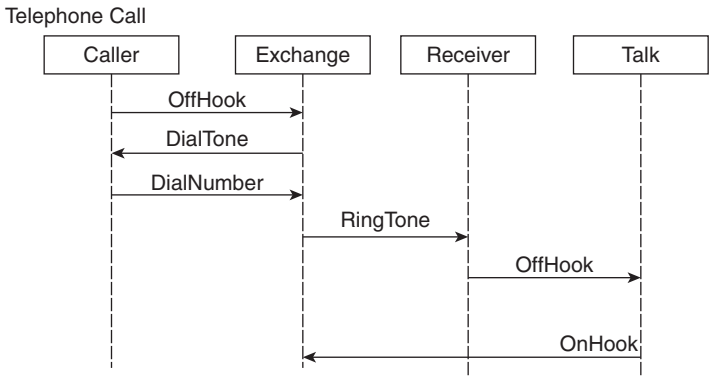

13. (b) Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done. Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. There are two kinds of interaction models are:
- Sequence diagrams
- Collaboration diagrams.

**Sequence diagrams:**

UML Sequence Diagram Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. It shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence. A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction.

An object is shown as a box at the top of a dashed vertical line. A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles. A sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

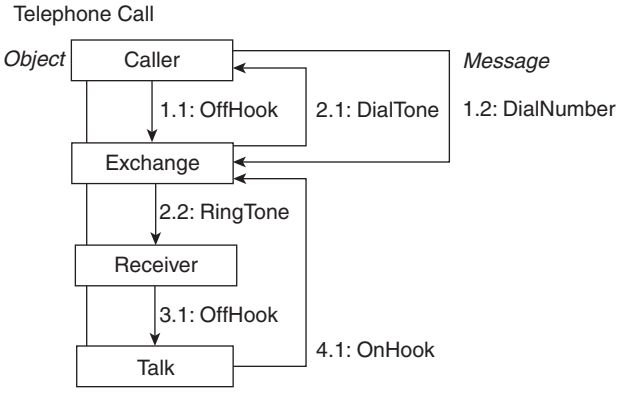
An example of a sequence diagram.




Each message is represented by an arrow between the lifelines of two objects. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary.

**Collaboration diagrams:**

A collaboration diagram represents collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figures. Arrows indicate the message sent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages. A collaboration diagram provides several numbering schemes.



14. (a) General Responsibility Assignment Software Patterns (GRASP), consists of guidelines for assigning responsibility to classes and objects in object-oriented design. GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.The different patterns and principles used in GRASP are:
- Controller
- Creator
- Indirection
- Information Expert
- High Cohesion
- Low Coupling
- Polymorphism
- Protected Variations
- Pure Fabrication

Larman states that, the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology. Thus, GRASP is really a mental toolset, a learning aid to help in the design of object-oriented software.

Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications. It provides a scheme for refining the subsystems or components of a software system or the relationships among them. Design patterns are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context. The main idea is to provide documentation to help categorize and communicate about solutions to recurring problems.

Design patterns are more abstract than frameworks. Design patterns are smaller architectural elements than frameworks. Design patterns are less specialized than frameworks.

**Controller:** The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event. A

use case controller should be used to deal with all system events of a use case, and may be used for more than one use case.

**Creator:** Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes. Creator pattern is responsible for creating an object of class.

**High Cohesion:** It is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change.

**Indirection:** The Indirection pattern supports low coupling between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.

**Information Expert:** Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on. Using the principle of Information Expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. It will lead to placing the responsibility on the class with the most information required to fulfill it.

**Low Coupling:** Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

- low dependency between classes,
- low impact in a class of changes in other classes,
- high reuse potential.

**Polymorphism:** According to Polymorphism, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

**Protected Variations:** The Protected Variations pattern protects elements from the variations on other elements by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

**Pure Fabrication:** A Pure Fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived.

14. (b) Design patterns are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context. A design pattern provides a scheme for refining the subsystems or components of a software system or the relationships among them. Design patterns are documented by writing essays in a fairly well-defined form.

Design pattern identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. It identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

The GoF patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral.

- Creational Patterns: Abstract object instantiation
- Structural Patterns: Compose and organize objects into larger structures
- Behavioral Patterns: Algorithms, interactions and control flow between objects.

**Creational Patterns:**

- **Abstract Factory**: Creates an instance of several families of classes. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Factory Method**: Creates an instance of several derived classes. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Singleton:** A class of which only a single instance can exist. Ensure a class only has one instance, and provide a global point of access to it.

**Structural Patterns:**

- **Adapter**: Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Behavioral Patterns:**

- **Observer**: A way of notifying change to a number of classes. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Adapter:**

Convert the original interface of a component into another interface through an intermediate adapter object. Adapters use interfaces and polymorphism to add a level of indirections to varying APIs in other components. Using adapter

- Protected Variations, Low Coupling, Polymorphism, Indirection helps us to view through the myriad details and see the essential alphabet of design techniques being applied.
- Naming Convention has the advantage of easily communicating to others reading the code or diagrams what design patterns are being used.

**Factory:**

Create a Pure Fabrication object called a Factory that handles the creation. Showing a pseudo code allows one to include dynamic algorithm details on a static class diagram such that it may lessen the need for interaction diagrams. The advantages of factory objects are:

- Separate the responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic
- Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

**Singleton:**

Define a static method of the class that returns the singleton. With this approach, a developer has global visibility to this single instance, via the static getInstance method of the class.

15. (a) Each class in design is implemented by coding it in a programming language or by using a pre-existing component. Exactly what a class in design corresponds to depends on the programming language. The design model can be more or less close to the implementation model depending on how you map its classes to classes or similar constructs in the implementation language.

Design must define enough of the system so that it can be implemented unambiguously. The degree of specification varies with the expertise of the implementer, the complexity of the design, and the risk that the design might be misconstrued. The design is elaborated to the point that the design can be transformed automatically into code. This typically involves extensions to standard UML to represent language or environment specific semantics.

The design may also be hierarchical, such as the following:

- a high level design model which sketches an overview of the overall system
- a subsystem specification model which precisely specifies the required interfaces and behavior of major subsystems within the system
- a detailed design model for the internals of subsystems

**Sketch and Code**

One common approach to design is to sketch out the design at a fairly abstract level, and then move directly to code. Maintenance of the design model is manual. In this approach, we let a design class be an abstraction of several code-level classes. You map each design class to one "head" class that, in turn, can use several "helper" classes to perform its behavior. You can use "helper" classes to implement a complex attribute or to build a data structure that you need for the implementation of an operation. In design, you don't model the "helper" classes and you only model the key attributes, relationships, and operations defined by the head class. The purpose of such a model is to abstract away details that can be completed by the implementer.

**Round-Trip Engineering**

In round-trip engineering environments, the design model evolves to a level of detail where it becomes a visual representation of the code. The code and its visual representation are synchronized.

**High Level Design Model and Detailed Design Model**

In this approach, there are two levels of design model maintained. Each high level design element is an abstraction of one or more detailed elements in the round-tripped model. Traceability from the

high level design model elements to round-trip model elements can help maintain consistency between the two models.

**Single Evolving Design Model**

In this approach, there is a single Design Model. Initial sketches of design elements evolve to the point where they can be synchronized with code. Diagrams, such as those used to describe design use-case realizations, initially reference sketched design classes, but eventually reference language-specific classes. High level descriptions of the design are maintained as needed, such as:

- diagrams of the logical structure of the system
- subsystem/component specifications
- design patterns / mechanisms

Such a model is easier to maintain consistent with the implementation.

**Specification and Realization Models**

A related approach is to define the design in terms of specifications for major subsystems, detailed to the point where client implementations can compile against them. The detailed design of the subsystem realization can be modeled and maintained separately from this specification model. Models can be detailed and used to generate an implementation. Both structure (i.e.) class and package diagrams and behavior diagrams (i.e.) collaboration, state, and activity diagrams can be used to generate executable code.

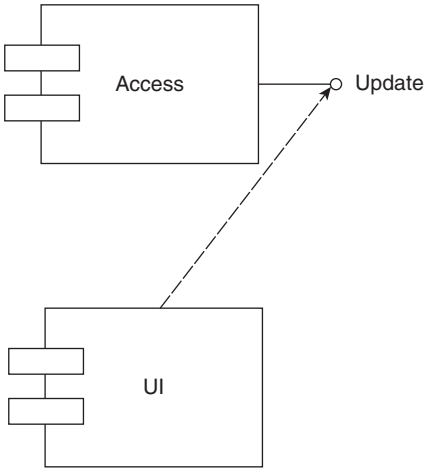
15. (b) Implementation diagrams show the implementation phase of systems development, such as the source code structure and the run-time implementation structure. There are two types of implementation diagrams:

- Component diagrams show the structure of the code itself
- Deployment diagrams show the structure of the runtime system.

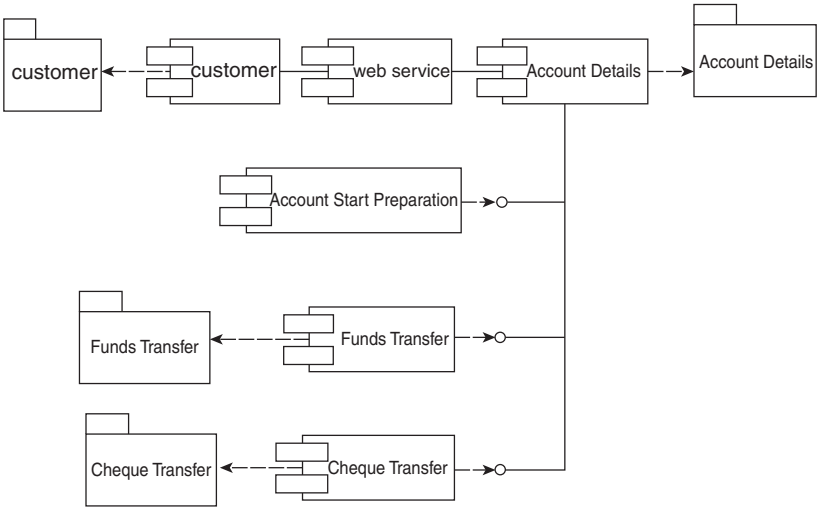
**Component Diagram:**

Component diagrams model the physical components in a design. These high-level physical components mayor may not be equivalent to the many smaller components you use in the creation of your application. Another way of looking at components is the concept of packages. A package is used to show how you can group together classes, which in essence are smaller scale components. A package usually will be used to group logical components of the application, such as classes, and not necessarily physical components.

The package will become a component. A component diagram is a graph of the design's components connected by dependency relationships. A component is represented by the boxed figure. Dependency is shown as a dashed arrow.



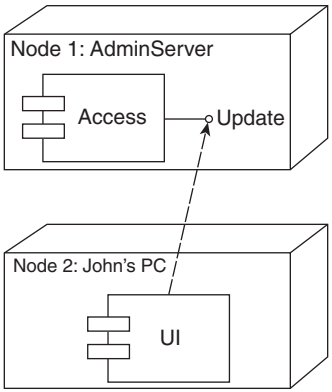

**Example: Banking application**



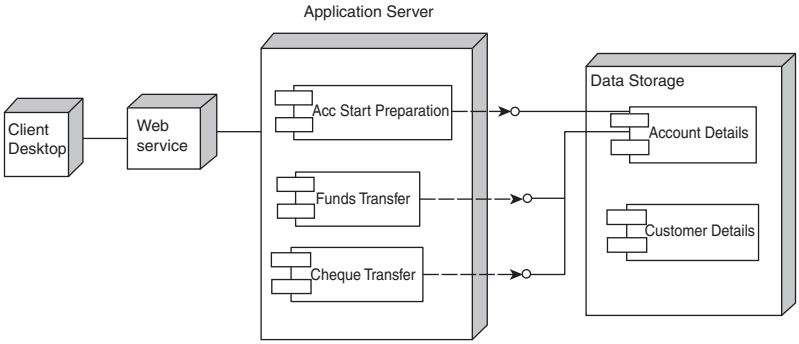

**Deployment Diagram:**

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live in them. Software component instances represent run-time manifestations of code units. Deployment diagrams are used to show how physical modules of code are distributed on various hardware platforms. In many cases, component and deployment diagrams can be combined.

A deployment diagram is a graph of nodes connected by communication association. Nodes may contain a component instance, which means that the component lives or runs at that node. Components may contain objects; this indicates that the object is part of the component. Components are connected to other components by dashed arrow dependencies, usually through interfaces, which indicate one component uses the services of another. Each node or processing element in the system is represented by a three-dimensional box. Connections between the nodes themselves are shown by solid lines.
The basic UML notation for a deployment diagram.




**Example: Banking application**



# B.E/B.Tech. DEGREE EXAMINATION, NOV/DEC 2010

## Sixth Semester

## Computer Science and Engineering

## CS1402-OBJECT ORIENTED ANALYSIS AND DESIGN

## (Common to Information Technology)

### PART A – (10 × 2 =20 marks)

1. Difference between Unified Approach and Unified Modeling Language.
2. What is Dynamic Inheritance?
3. What is an Objectory?
4. What is a Dynamic model?
5. What is Classification?
6. Name some sources of difficulties for collecting requirements.
7. What is Encapsulation Leakage?
8. Define Persistence.
9. What is Quality?
10. Why do you need to prototype user interface?

### PART-B

11. Describe the Object Oriented Systems life cycle?

Or

12. (a) Object Identity
    (b) Object Persistence
    (c) Dynamic Binding
    (d) meta classes

13. Explain the three different object oriented methodologies

Or

14. Explain Set of diagrams that explain collaboration of objects

15. Explain the noun phrase approach for identifying candidate class of a system.

Or

16. Explain the steps involved in identifying associations between the candidate classes

17. Explain the different design corollaries derived from design axioms.

Or

18. Explain the Client-Server Computing

19. Explain the different testing strategies

Or

20. Different processes in view layer classes.

# Solutions

## PART A

1.

| Unified Approach | Unified Modeling Language |
|---|---|
| • The unified approach (UA) is a methodology and framework for software development.<br>• The UA, based on methodologies by Booch, Rumbaugh, Jacobson.<br>• It tries to combine the best practices, processes, and guidelines. | • It is a set of notations and conventions used to describe and model an application of software.<br>• Expressed in English using language called Object Constrain Language.<br>• It is a language for Specifying, Visualizing, Constructing, Documenting the software system and its components. |

2. Dynamic Inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. It has ability to add, delete, or change parents from objects (or classes) at runtime.

3. Object oriented software engineering (OOSE), also called Objectory, and is a method of object oriented development with the specific aim to fit the development of large, real-time systems. The development process, also called as use case driven process. It stresses that use cases are involved in analysis, design, validation and testing.

4. Dynamic model is viewed as a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. It shows how the business objects interact to perform tasks. It is most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of UML dynamic model.

5. Classification, the process of checking to see if an object belongs to category or a class, is regarded as a basic attribute of human nature. Classification guides us in making decisions about modularization. Classes are an important mechanism for classifying objects. The chief role of class is to define the attributes, methods, and applicability of its instances.

6. Normans explains the most common sources of requirement Difficulties:
   - Fuzzy descriptions
   - Incomplete requirements
   - Unnecessary features
7. Lack of a well-designed protocol lead to Encapsulation Leakage. It is defined as the details about a class's internal implementation are disclosed through the interface. When internal details become visible, the flexibility is retained for future changes.
8. Persistence refers ability of some objects to outlive the program that created them. It is also defined as An object can persist beyond application session boundaries, during which the object is stored in a file or a database, in some file or database form. Life time is short for local objects and long the objects that is stored in database.
9. Quality is that the designed software or the software that is to be delivered to the customer must be free of errors or bugs that cause unexpected result. This is because the computers are infamous for doing what we tell them to do, not necessarily what we want them to do.
10. - To better understand the system requirements. This is done using CASE tools, operational software using prototyping or normal development tools.
    - It provide effective tool for communicating the design.
    - Help to understand the task flow and better visualize the design.
    - Low cost for getting user input.

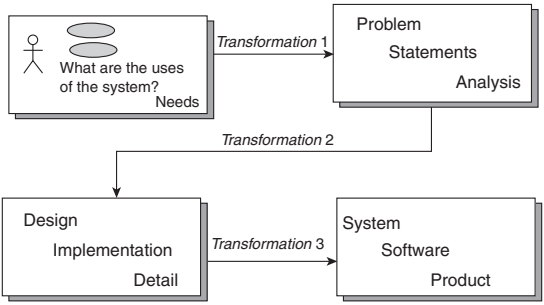
## PART B

**11. Object Oriented Systems Development Life Cycle**

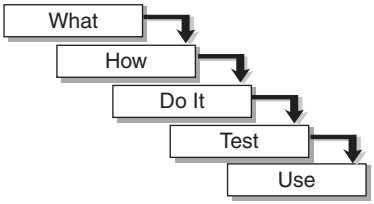
Goals

- The software development process
- Building high-quality software
- Object-oriented systems development
- Use-case driven systems development
- Prototyping
- Rapid application development
- Component-based development
- Continuous testing and reusability

Software Process The essence of the software process is the transformation of Users 'needs to the application domain into a software solution.




**Traditional Waterfall Approach to Systems Development**



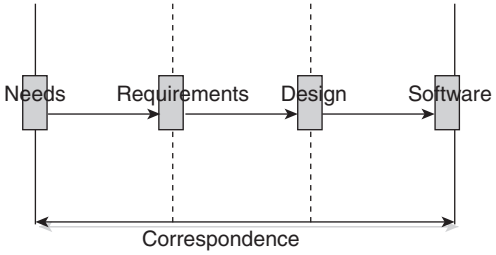

**Software Quality**

- There are two basic approaches to systems testing.
- We can test a system according to how it has been built.
- Alternatively, we can test the system with respect to what it should do.

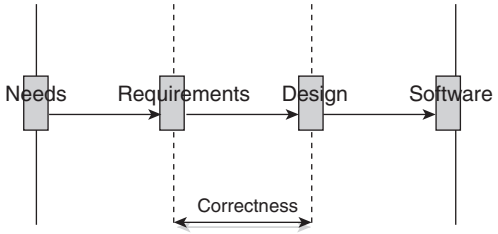
**Quality Measures**

- Systems can be evaluated in terms of four quality measures:
  - Correspondence
  - Correctness
  - Verification
  - Validation
- Correspondence measures how well the delivered system corresponds to the needs of the operational environment.

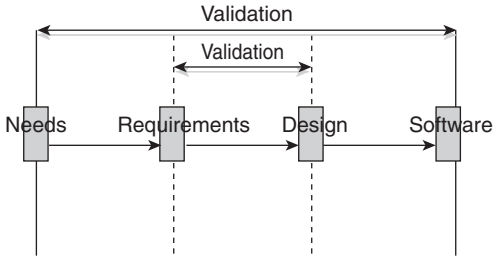
It cannot be determined until the system is in place.



Correctness measures the consistency of the product requirements with respect to the design specification.

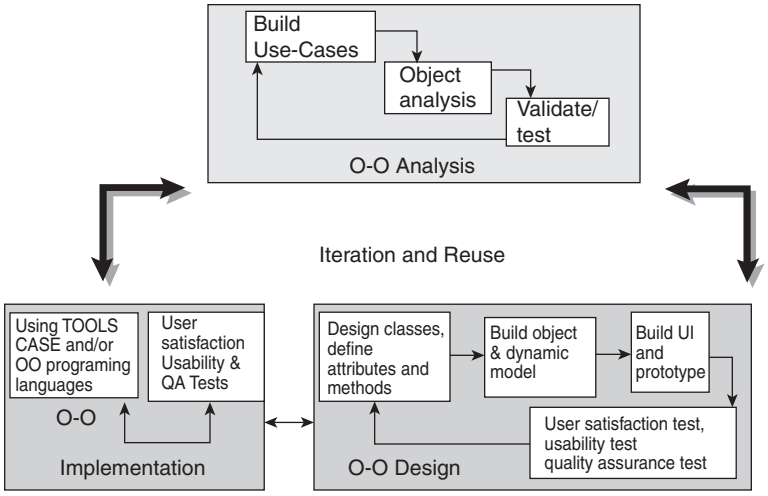



- Verification – "Am I building the product right?"
- Validation – "Am I building the right product?"




- Verification is to predict the correctness.
- Validation is to predict the correspondence.

Object Oriented System Development Approach

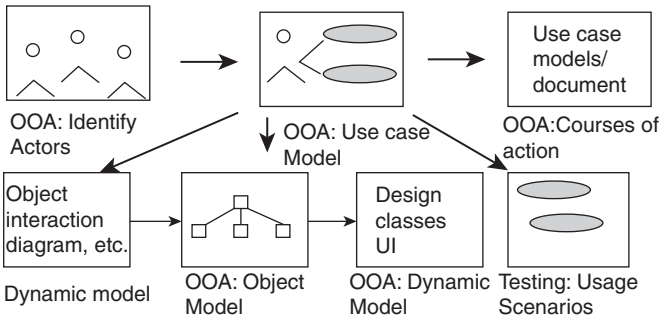


Object-Oriented Systems Development activities

- Object-oriented analysis.
- Object-oriented design.
- Prototyping.
- Component-based development.
- Incremental testing.

Use-case driven systems development

Use Case, is a name for a scenario to describe the user–computer system interaction.




Object-Oriented Analysis OO analysis concerns with determining the system requirements and identifying classes and their relationships that makes up an application.

Object-Oriented Design

- The goal of object-oriented design (OOD) is to design
- The classes identified during the analysis phase,
- The user interface and
- Data access.

OOD activities include:

- Design and refine classes.
- Design and refine attributes.
- Design and refine methods.
- Design and refine structures.
- Design and refine associations.
- Design User Interface or View layer classes.
- Design data Access Layer classes.

**Prototyping**

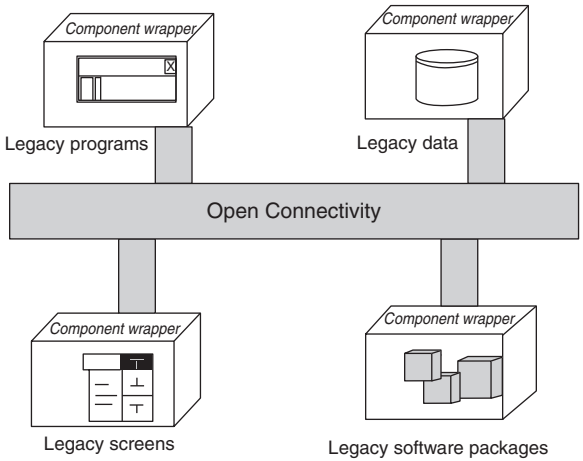
- A Prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system.
- It can also give users a chance to comment on the usability and usefulness of the design.

**Types of Prototypes**

- A horizontal prototype is a simulation of the interface.
- A vertical prototype is a subset of the system features with complete functionality.
- An analysis prototype is an aid for exploring the problem domain.
- A domain prototype is an aid for the incremental development of the ultimate software solution.

**Component-based development (CBD)**

- CBD is an industrialized approach to the software development process.



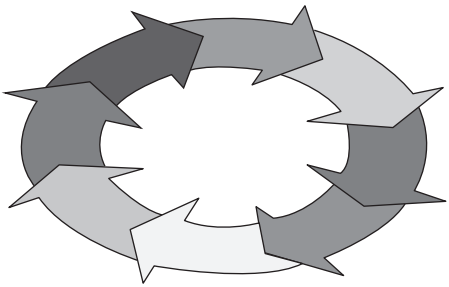

- Application development moves from custom development to assembly of pre-built, pre-tested, reusable software components that operate with each other.

**Rapid Application Development (RAD)**

- RAD is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods.
- RAD does not replace SDLC but complements it, since it focuses more on process description and can be combined perfectly with the object-oriented approach.

**Incremental Testing**

- Software development and all of its activities including testing are an iterative process.



- If you wait until after development to test an application for bugs and performance, you could be wasting thousands of dollars and hours of time.

Reusability -A major benefit of object-oriented systems development is reusability, and this is the most difficult promise to deliver on. Reuse strategy

- Information hiding (encapsulation).
- Conformance to naming standards.
- Creation and administration of an object repository.
- Encouragement by strategic management of reuse as opposed to constant redevelopment.
- Establishing targets for a percentage of the objects in the project to be reused (i.e., 50 percent reuse of objects).

The essence of the software process is the transformation of users'needs into a software solution. The O-O SDLC is an iterative process and is divided into analysis, design, prototyping/ implementation, and testing.

**12. (a) Object Identity**

Object identity: An object retains its identity even if some or all of the values of variables or definitions of methods change over time. Object identity is a stronger notion of identity than typically found in programming languages or in data models not based on object orientation.

**Several forms of identity:**

- Value: A data value is used for identity (e.g., the primary key of a tuple in a relational database).
- Name: A user-supplied name is used for identity (e.g., file name in a file system).
- Built-in: A notion of identity is built-into the data model or programming languages, and no user-supplied identifier is required (e.g., in OO systems).

Object identity is typically implemented via a unique, system-generated OID. The value of the OID is not visible to the external user, but is used internally by the system to identify each object uniquely and to create and manage inter-object references.

**(b) Object Persistence**

Objects have life time. They are created and can exist for a period of time.

- A file or a database can provide support for objects having a longer life timelines than the duration of the process for which they were created. Polymorphism
- Polymorphism means that the same operation may behave differently on different classes.
  Ex. Move operation.

**(c) Dynamic Binding**

The process of determining (dynamically) at run time which functions to invoke is termed dynamic binding. Dynamic Binding: the decision is made at run-time based upon the type of the actual object. Used when the derived classes may be able to provide a different (e.g., more functional, more efficient) implementation that should be selected at run-time. They occur when polymorphic calls are issued. Dynamic binding allows some method invocation decisions to be deferred until the information is known.

Used to build dynamic type hierarchies & to form "abstract data types"

**(d) Meta-Classes**

- Everything is an object.
- How about a class?
- Is a class an object?
- Yes, a class is an object! So, if it is an object, it must belong to a class.
- Indeed, class belongs to a class called a Meta-Class or a class' class.
- Meta-class used by the compiler. For example, the meta-classes handle messages to classes, such as constructors and "new." Rather than treat data and procedures separately, object-oriented programming packages them into "objects." O-O system provides you with the set of objects that closely reflects the underlying application. Advantages of object-oriented programming are:

The ability to reuse code, develop more maintainable systems in a shorter amount of time.

- More resilient to change, and
- More reliable, since they are built from completely tested and debugged classes.

## 13. Three different object oriented methodologies

### Basic Definition

A methodology is explained as the science of methods.

- A method is a set of procedures in which a specific goal is approached step by step.
- Many methodologies are available to choose from for system development.
- Here, we look at the methodologies developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML) and the bases of the UA

### Strength of the Methods

- Rumbaug: Describing Object Model or the static structure of the system
- Jacobson: good for producing user-driven analysis models
- Booch: Detailed object-oriented design models

Rumbaugh Methodologies OMT (Object Modeling Technique) describes a method for the analysis, design, and implementation of a system using an object-oriented technique. Class attributes, method, inheritance, and association also can be expressed easily Phases of OMT

- Analysis
- System Design
- Object Design
- Implementation

OMT consists of four phases, which can be performed iteratively:

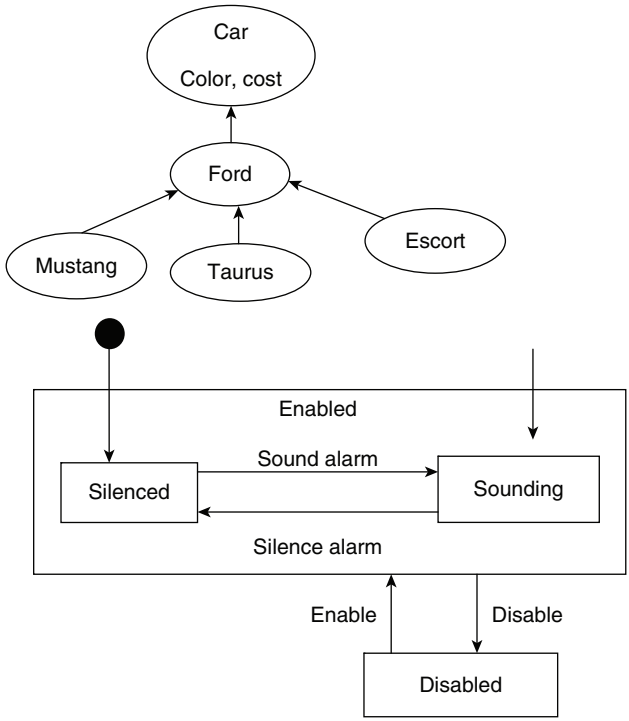
- Analysis. The results are objects and dynamic and functional models.
- System design. The result is a structure of the basic architecture of the system.
- Object design. This phase produces a design document, consisting of detailed objects and dynamic and functional models.
- Implementation. This activity produces reusable, extendible, and robust code.

OMT Modeling OMT separates modeling into three different parts:

- An object model, presented by the object model and the data dictionary.
- A dynamic model, presented by the state diagrams and event flow diagrams.
- A functional model, presented by data flow and constraints.

**Booch Methodology**

- The Booch methodology covers the analysis and design phases of systems development.
- Booch sometimes is criticized for his large set of symbols.
- The Booch method consists of the following diagrams:
  - Class diagrams
  - Object diagrams
  - State transition diagrams
  - Module diagrams
  - Process diagrams
  - Interaction diagrams




- The Booch methodology prescribes

A macro development process

**Micro development process**

The Macro Development Process -It servers as a controlling framework for the micro process. The primary concern is Technical Management of

the System. The macro development process consists of the following steps:

1. Conceptualization
2. Analysis and development of the model.
3. Design or create the system architecture.
4. Evolution or implementation.
5. Maintenance.

**Conceptualization:**

- Establish the core requirements of the system
- Establish a set of goals and develop prototype to prove the concept

**Analysis and development of the modal**

- Using the class diagram to describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system
- Using the object diagram to describe the desired behavior of the system in terms of scenarios or, alternatively
- Using the interaction diagram to describe behavior of the system in terms of scenarios

**Design or create the system architecture**

- Using the class diagram to decide what mechanisms are used to regulate how objects collaborate
- Using the module diagram to map out were each class and object should be declared
- Using the process diagram to determine to which processor to allocate a process. Also, determine the schedules for multiple processes on each relevant processor

**Evolution or implementation**

- Successively refine the system through much iteration
- Produce a stream of software implementations, each of which is refinement of the prior one Maintenance
- Make localized changes to the system to add new requirements and eliminate bugs

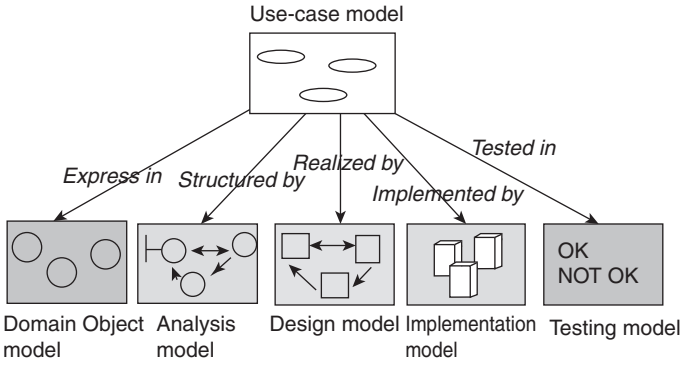
**The Micro Development Process**

- The micro development process consists of the following steps:
- Identify classes and objects.
- Identify class and object semantics.
- Identify class and object relationships.
- Identify class and object interfaces and implementation.

**Jacobson Methodologies**

The Jacobson et al. methodologies (e.g., OOBE, OOSE, and Objectory) cover the entire life cycle and stress traceability between the different phases. Object-Oriented Software Engineering: Objectory

- Object-oriented software engineering (OOSE), also called Objectory, is a method of object-oriented development with the specific aim to fit the development of large, real-time systems.
- Objectory is built around several different models:
- Use case model.
- Domain object model.
- Analysis object model. Implementation model
- OOBE consists of:
  - Analysis phase
  - Design
  - Implementation phases and
  - Testing phase.




14. Collaboration Diagrams UML provides two sorts of interaction diagram,
    - Sequence and
    - Collaboration diagrams.

    Collaboration diagrams illustrate the interaction between the objects, using static spatial structure.

    Unlike sequence diagram the time is not explicitly represented in these diagrams In collaboration diagram the sequence of messages is indicated by numbering the messages. The UML uses the decimal numbering scheme. In these diagrams, an actor can be displayed in order to represent the triggering of interaction by an element external to the system.

    This helps in representing the interaction, without going into the details of user interface.

**Components of collaboration diagram**

**Named objects**

- Links: Links are represented by a continuous line between objects, and indicates the exchange of messages.

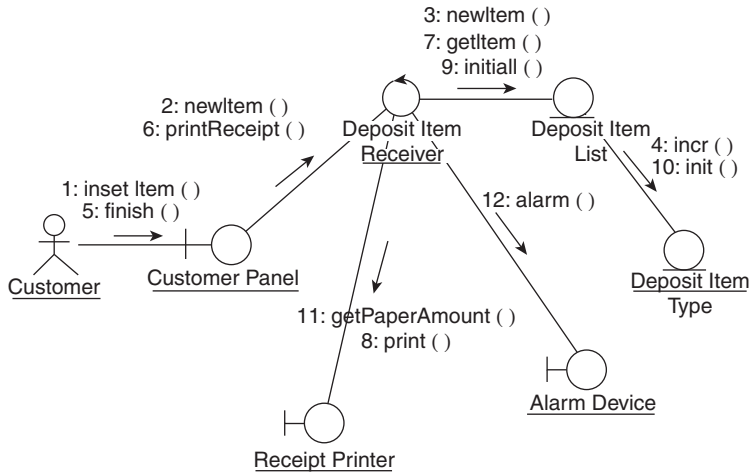
**Messages have following attributes**

- Synchronization --thread name, step within thread.
- Sequence number
- Message labels: The name of the message often corresponds to an operation defined in the class of the object that is the destination of the message. Message names may have the arguments and return values.
- It uses decimal notation.
- Class diagram

Message direction-A class diagram shows the existence of classes and their relationships in the logical view of a system

UML modeling elements in class diagrams are:

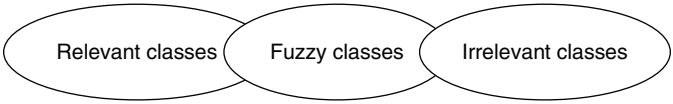
- Classes, their structure and behavior.
- Relationships components among the classes like association, aggregation, composition, dependency and inheritance
- Multiplicity and navigation indicators
- Role names or labels.




- A collaboration diagram represents a set of objects related in a particular context, and the exchange of their messages to achieve a desired outcome.
- Used for design of: components, object, subsystems
- Diagrams show entity event responses

- Event
- receiving a message

15. It examine Use cases, conduct interviews, and read requirements specification carefully, dividing noun phrases into three categories.




Selecting classes from the Relevant and Fuzzy Categories
- Redundant classes
- Adjective classes
- Attribute classes
- Irrelevant classes

The classes are identified from the NOUN PHRASES that exist in the requirements/ use case. The steps involved are

1. Examining the use case/ requirements.
2. Nouns in textual form are selected and considered to be the classes.
3. Plural classes are converted into singular classes.
4. Identified classes are grouped into 3 categories
   - Irrelevant Classes - They are the unnecessary classes
   - Relevant Classes - They are the necessary classes
   - Fuzzy Classes - They are the classes where exist some uncertainty in their existence.
5. Identify candidate classes from above set of classes.

Initial list of noun phrases: candidate classes
Account
Bank
Reviewing the redundant classes and Building a Common Vocabulary
Reviewing the classes containing Adjectives
Reviewing the possible Attributes
Reviewing the Classes Purpose

**Guidelines for selecting candidate classes from Relevant, Irrelevant and Fuzzy set of classes.**

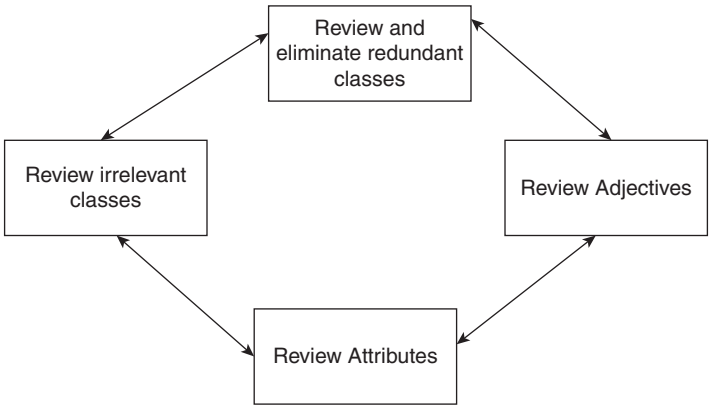
1. **Redundant Classes** – Never keep two classes that represent similar information and behavior. If there exist more than one name for a similar class select the more relevant name. Try to use relevant names that are use by the user. (E.g. Class Account is better than Class Moneysaving)
2. **Adjective Classes** – An adjective qualifies a noun (Class). Specification of adjective may make the object to behave in a different way or may be totally irrelevant. Naming a new class can be decided how far

the adjective changes the behavior of the class. (E.g. The behavior of Current Account Holder

Differs from the behavior of Savings Account Holder. Hence they should be named as two different classes. In the other case the toper adjective doesn't make much change in the behavior of the student object. Hence this can be added as a state in the student class and no toper class is named)

3. **Attribute Class** – Some classes just represent a particular property of some objects. They should not be made as a class instead they can be added as a property in the class. (E.g. No Minimum Balance, Credit limit are not advised to named as a class instead they should be included as an attribute in Account class).
4. **Irrelevant Class** – These classes can be identified easily. When a class is identified and named the purpose and a description of the class is stated and those classes with no purpose are identified as irrelevant classes. (E.g .Class Fan identified in the domain of attendance management system is irrelevant when u model the system to be implemented.

The iterative Process can be represented as below

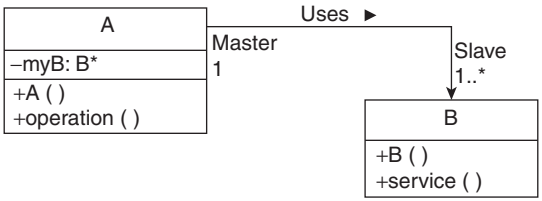



These types of classes can be scraped out.)The above steps are iterative and the guidelines can be used at any level of iteration. The cycles of iteration is continues until the identified classes are satisfied by the analyst/ designer.

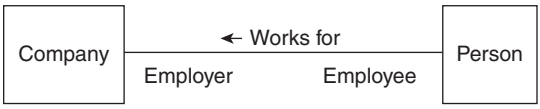
16. In Object-oriented design, association is a relation between two objects, for example, Car & Wheel have an association (that is, car contains wheels), Toolbar & Button have an association (that is, toolbar is composed of buttons), a Student is associated to Course (that is, a student can study a course). It can be strong association or weak association.

**Association Details**

- Name gives details of association
- Name can be viewed as verb of a sentence



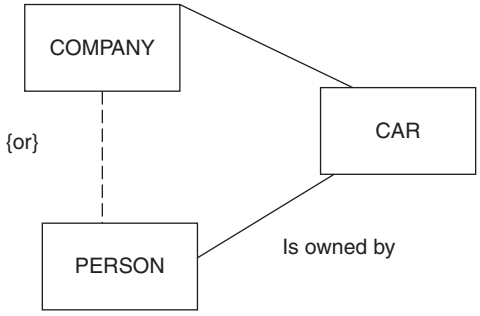

If A is strongly associated to B then A cannot exist without B. In implementation terms, object A would be composed of object B & without creating B, creation of A is not possible. For example, a Toolbar cannot be created without creating a Button. If A is weakly associated to B then A can exists without B. For example, a Student can exist without the existence of Course. It represents the association between two classes represented by a straight line connecting 2 classes. Association has got a name written on the line and association role.




Association Role: it's related to association. Each class that is a member of an association plays a role in the association called association role.
E.g. the person plays the role of EMPLOYEE in the WORKS FOR association where a company plays the role of EMPLOYER.

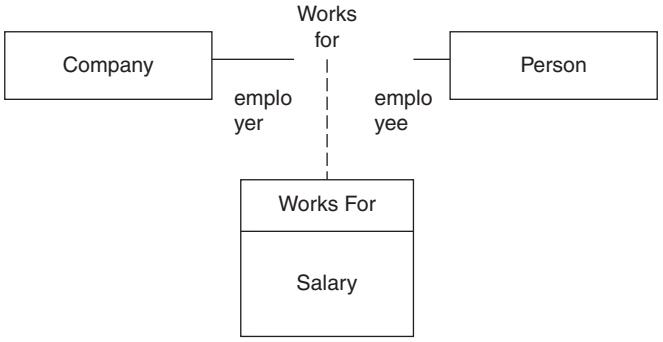
**OR – Association:**

It's a relation in which a class is associated to more than one class and only one association is instantiated at any instance of time for an object. It is represented by a dashed line connecting two associations. A constraint string can be used to label the OR association line.



**Association Class:**

It's an association that has class properties. The association class is attached to an association with a dashed line. Here the works for relation has got one attribute salary. Hence an association class is maintained.




- Notes at association ends explain "roles" of classes (objects).
- Multiplicities show number of objects which participate in the association

  N – Ary Association:

  It's an association where more classes participate. They are connected by a big diamond and the name of the association is named near the line.
- Can express different kinds of associations between classes/objects with UML
  - Association, aggregation, composition, inheritance
  - Friendship, parametric association
- Can go from simple sketches to more detailed design by adding adornments
  - Name, roles, multiplicities
  - lifetime control

17. An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.

    **Axiom 1**-it states that, during the design process, as we go from requirement and use-case to a system component, each component must satisfy that requirement, without affecting other requirements.

    **E.g**.Requirement1: Node1 should send multimedia files requested by the Node2.

    Requirement2: Node1 one should take minimum time for sending due to heavy traffic. Consider the component C1 responsible for sending Multimedia files.

**Choice 1:** C1 reads the files and send the file header first and then the content in a byte stream. Here the component satisfies the first requirement where it fails to satisfy the second requirement.
**Choice 2:** C1 reads the files and compress the content and file header contains the file and compression information. Since the file size transferred is reduced this choice satisfies both requirements.
**Axiom 2**-it is concerned with simplicity. Rely on a general rule known as Occam's razor.

It deals with the simplicity and less information content. The fact is less number of information makes a simple design, hence less complex. Minimizing complexity makes the design more enhanced. The best way to reduce the information content is usage of inheritance in design. Hence more information can be reused from existing classes/ components.

**Chioce 1: (with out inheritance)**

| Vehicle Name |
|---|
| Name<br>Brand<br>Owner |
| Stop ()<br>Start () |

| Car |
|---|
| Name<br>Brand<br>Owner<br>Color<br>Engine No |
| Stop() |

Corollaries -May be called Design rules, and all are derived from the two basic axioms.
They are

1. Corollary 1 _ Uncoupled Design with less information content (from Axiom1 and 2)
2. Corollary 2 _ Single purpose classes (from Axiom1 and 2)
3. Corollary 3 _ large number of simple classes (from Axiom1 and 2)
4. Corollary 4 _ Strong Mapping (from Axiom 1)
5. Corollary 5 _ Standardization (from Axiom 2)
6. Corollary 6 _ Design with inheritance (from Axiom 2)

- Corollaries 1, 2 and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 & 6 are from axiom 2.

**Corollary 1:** Uncoupled design with less information content. highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects

Main goal –To maximize objects cohesiveness among objects & software components * to improve coupling

Strong coupling among objects complicates a system, since the class is harder to understand or highly interrelated with other classes.

Degree or strength of coupling between two components is measured by the amount & complexity of information transmitted between them design has 2 types of coupling: Interaction coupling and Inheritance coupling

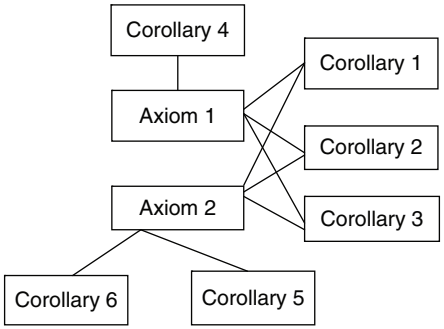



**Fig:** Origin of corollaries

Interaction coupling -The amount & complexity of messages between components.

- Desirable to have a little interaction.
- Minimize the number of messages sent & received by an object

**Corollary 2: Single purpose.**

- Each class must have a purpose & clearly defined.
- Each method must provide only one service

**Corollary 3: Large number of simple classes.**

- Keeping the classes simple allows reusability.
- A class that easily can be understood and reused (or inherited) contributes to the overall system.
- Complex & poorly designed class usually cannot be reused

**Guideline**

- The smaller are your classes, the better are your chances of reusing them in other projects. Large & complex classes are too specialized to be reused
- The emphasis OOD places on encapsulation, modularization, and polymorphism suggests reuse rather than building anew
- Primary benefit of software reusability
- Higher productivity

**Corollary 4: Strong mapping.**

There must be a strong association between the analysis's object and design's object

- OOA and OOD are based on the same model
- As the model progresses from analysis to implementation, more detailed is added

**Corollary 5: Standardization.**

- Promote standardization by designing interchangeable components and reusing existing classes or components
- The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

**Corollary 6: Design with inheritance.**

- Common behavior (methods) must be moved to super classes.
- The super class-subclass structure must make logical sense

18. It is the logical extension of modular programming. In modular programming we separate a large piece of software into its constituent parts (modules). This makes development easier and gives better maintainability. In client-server computing all those modules are not executed within the same memory space or even on the same machine. Here the calling method becomes "client" and the called module becomes the "server".

    The important component of client-server computing is connectivity, which allows applications to communicate transparently with other programs or processes, regardless of their locations. The key element of connectivity is the network operating system (NOS), known as middleware. The NOS provides services such as routing, distribution, messages, filing, printing and network management.

    Client programs manage user interface portion of the application, validate data entered by the user, dispatch requests to server program and executes business logic. The business layer contains all the objects that represent the business. The client-based process is the front-end of the application, which the user sees and interacts with. It manages the local resource with which the user interacts, such as the monitor, keyboard, workstation, CPU and peripherals. A key component of a client workstation is the graphical user interface (GUI).It is responsible for detecting user actions, managing the Windows on the display and displaying the data in the Windows. The server process performs back-end tasks.

**Transaction**

- A transaction is a unit of change, in which either all changes to objects within a transaction will be applied or not at all.

A transaction is said to commit if all changes can be successfully made to the database and to abort if all changes cannot be successfully made to the database

**Two-Tier Architecture**

- Two-Tier Architecture two-tier architecture is one where a client talks directly to a server, with no intervening server.
- This type of architecture is typically used in small environments with less than 50 users

**Three-Tier Architecture**

- Three-tier architecture introduces another server (or an "agent") between the client and the server.
- The role of the agent is many fold.
- It can provide translation services as in adapting a legacy application on a mainframe to a client/server environment.

**Components of client server Architecture**

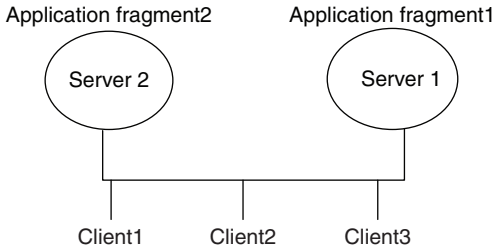
User Interface
Business Processing
Database Processing

**Distributed and Cooperative processing**

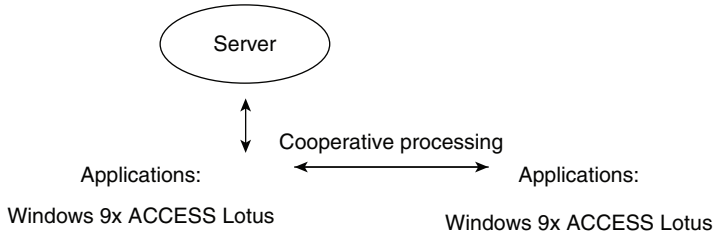
It means distribution of applications and business logic across multiple processing platforms.

Processing will occur in more than one processor.

The processing is distributed across two or more machines where perform part of application in a sequence.



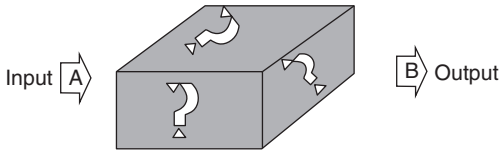
Cooperative processing is computing that require two or more distinct processors to complete a single transactions. It is related to both distributed and cooperative.




19. There are four types of testing strategies are available. They are
    - Black Box Testing
    - White Box Testing
    - Top-down Testing
    - Bottom-up Testing

**Black Box Testing**

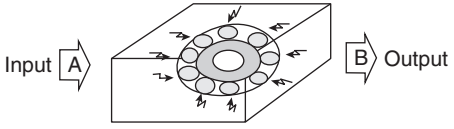
- In a black box, the test item is treated as "black" whose logic is unknown.
- All that's known is what goes in and what comes out, the input and output
- Black box test works very nicely in testing objects in an O-O environment.




- Once you have created fully tested and debugged classes of objects you will put them into library for use or reuse.

**White Box Testing**

- White box testing assumes that specific logic is important, and must be tested to guarantee system's proper functioning.
- One form of white box testing is called path testing.



- It makes certain that each path in a program is executed at least once during testing.

**Two types of path testing are:**

- Statement testing coverage-To test every statement in the objects method by executing it at least once.
- Branch testing coverage-To test every branch alternative has been executed at least once.

**Top-down Testing**

It assumes that the main logic of the application needs more testing than supporting logic.

**Bottom-up Approach**

- It takes an opposite approach.
- It assumes that individual programs and modules are fully developed as standalone processes.
- These modules are tested individually, and then combined for integration testing.

**Impact of OO on Testing**

**Impact of Inheritance**

- It is not necessary to test Inherited Methods because its already been verified in the Base class.
- But if the inherited method is over ridded then the behavior may change and it is needed to be tested.

**Reusability of Test Cases**

- Test Cases can be reused based up on the level of reusage.
- Overridden methods show a different behavior.
- If the similarities in behavior exist test cases can be reused.
- In some cases inherited method accept same parameter as Base but different behavior.
- In the above case a test case can be reused such a way that the expected o/p of the test case is changed and used.

20. View layer objects are more responsible for user interaction and these view layer objects have more relation with the user where business layer objects have less interaction with users. Another feature of view layer objects are they deal less with the logic. They help the users to complete their task in an easy manner.

**The Major responsibilities of view layer objects are**

- Input - View Layer objects have to respond for user interaction. The user interface is designed to translate an action by the user (Eg. Clicking the button) in to a corresponding message.
- Output - Displaying or printing information after processing.

**View Layer Design Process:**

1. Macro Level UI Design Process

a. Identify classes that interact with human actors

b. A sequence/ collaboration diagram can be used to represent a clear picture of actor system interaction.

c. For every class identified determine if the class interacts with the human actor. If so

i. Identify the view layer object for that class.

ii. Define the relationship among view layer objects.

The Macro Development Process -It servers as a controlling framework for the micro process. The primary concern is Technical Management of the System. The macro development process consists of the following steps:

1. Conceptualization
2. Analysis and development of the model.
3. Design or create the system architecture.
4. Evolution or implementation.
5. Maintenance.

**Conceptualization:**

- Establish the core requirements of the system
- Establish a set of goals and develop prototype to prove the concept

Analysis and development of the modal

- Using the class diagram to describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system
- Using the object diagram to describe the desired behavior of the system in terms of scenarios or, alternatively
- Using the interaction diagram to describe behavior of the system in terms of scenarios

**Design or create the system architecture**

- Using the class diagram to decide what mechanisms are used to regulate how objects collaborate
- Using the module diagram to map out were each class and object should be declared

- Using the process diagram to determine to which processor to allocate a process. Also, determine the schedules for multiple processes on each relevant processor

**Evolution or implementation**

- Successively refine the system through much iteration
- Produce a stream of software implementations, each of which is refinement of the prior one

**Maintenance**

- Make localized changes to the system to add new requirements and eliminate bugs

**2. Micro Level UI Design Process**

a. Design of view layer objects by applying Design Axioms and Corollaries.
b. Create prototype of the view layer interface.

3. Testing the usability and user satisfaction testing.
4. Iterate and refine the above steps.

**User Interface Design Rules:**

**UI Design Rule 1: Making the interface simple**

For complex application if the user interface is simple it is easy for the users to learn new applications. Each User Interface class should have a well define single purpose. If a user cannot sit before a screen and find out what to do next without asking multiple questions, then it says your interface is not simple.

**UI Design Rule 2: Making the Interface Transparent and Natural.**

The user interface should be natural that users can anticipate what to do next by applying previous knowledge of doing things without a computer. This rule says there should be a strong mapping and users view of doing things.

**UI Design Rule 3: Allowing users to be in control of the Software.**

The UI should make the users feel they are in control of the software and not the software controls the user. The user should play an active role and not a reactive role in the sense user should initiate the action and not the software.

Some ways to make put users in control are

1. Make the interface forgiving.
2. Make the interface visual.
3. Provide immediate feedback.
4. Avoid Modes.

5. Make the interface consistent.

Purpose of View Layer Interface - Guidelines

Input-responding to user interaction

Output-displaying or printing business objects

The Micro Development Process

- The micro development process consists of the following steps:
- Identify classes and objects.
- Identify class and object semantics.
- Identify class and object relationships.
- Identify class and object interfaces and implementation.

**User Interface Design Rules:**

**UI Design Rule 1:** Making the interface simple for complex application if the user interface is simple it is easy for the users to learn new applications. Each User Interface class should have a well define single purpose. If a user cannot sit before a screen and find out what to do next without asking multiple questions, then it says your interface is not simple.

**UI Design Rule 2:** Making the Interface Transparent and Natural. The user interface should be natural that users can anticipate what to do next by applying previous knowledge of doing things without a computer. This rule says there should be a strong mapping and users view of doing things.

**UI Design Rule 3:** Allowing users to be in control of the Software. The UI should make the users feel they are in control of the software and not the software controls the user. The user should play an active role and not a reactive role in the sense user should initiate the action and not the software. Some ways to make put users in control are

1. Make the interface forgiving.
2. Make the interface visual.
3. Provide immediate feedback.
4. Avoid Modes.
5. Make the interface consistent.

Purpose of View Layer Interface - Guidelines

# B.E./B.Tech. DEGREE EXAMINATION
# APRIL/MAY 2010

## Sixth Semester

## Computer Science and Engineering

## CS 1402 - OBJECT ORIENTED ANALYSIS AND DESIGN

## (Common to Information Technology)

**Time: Three hours** **Maximum: 100 Marks**

**Answer ALL questions**

**PART A (10 · 2 = 20 Marks)**

1. Define Object. What is its significance?
2. What is the purpose of object persistence and object ID?
3. What is the purpose of OMT functional diagram?
4. What are the uses of UML component diagram?
5. What are the various types of association?
6. What do you mean by an actor in a use case?
7. What is the purpose of axiom?
8. What are the various attributes types?
9. What is scenario based testing?
10. What are the principal objectives of user satisfaction?

## PART B (5 · 16 = 80 Marks)

11. (a) (i) What is meant by Inheritance? Explain with a suitable example.
(ii) Explain polymorphism and its advantage with an example.

Or

(b) (i) How is software development viewed? What are the various phases of OOSD life cycle? Explain in detail.
(ii) Advantages and disadvantages of Waterfall Approach

12. (a) What are the components of Booch methodology? Explain with examples.

Or

(b) Compare and contrast between sequence and collaboration diagrams with the help of example diagrams.

13. (a) What is object-oriented analysis? Explain the various steps involved.

Or

(b) (i) What is a use case? Explain its uses in analysis.
(ii) Write down guidelines for finding use cases.

14. (a) Discuss different types of corollary of object oriented design axioms.

Or

(b) What is the need for multi database system? Explain object relational system.

15. (a) Explain the issues and objectives for SQA in detail.

Or

(b) Write about system usability and measuring user satisfaction.

# Solutions

## PART A

1. Object is a real-world entity, identifiably separate from its surroundings, has a well defined set of attributes and a well-defined set of procedures or methods. Properties (or attributes) describe the state (data) of an object. Methods (procedures) define its behavior. Object means a combination of data and logic that represents some real-world entity.
   (E.g.) Car is an object
   Color, manufacturer, cost, owner etc. are attributes.
   Drive it, lock it, tow it, carry passengers in it are all methods.

2. A file or a database can provide support for objects having a longer life-line, longer than the duration of the process for which they were created. This is called object persistence. An object can persist beyond application session boundaries, during which the object is stored in a file or a database. The object can be retrieved in an other application session and will have same state and relationship.

3. ***Functional model*****:** The functional model handles the process perspective of the model, corresponding roughly to data flow diagrams. Main concepts are process, data store, data flow, and actors.

   ***Functional Model in OMT*****:** In brief, a functional model in OMT defines the function of the whole internal processes in a model with the help of "Data Flow Diagrams" (DFD's). It details how processes are performed independently.

4. Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.
   So from that point component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.
   Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.
   A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

5. Association represents a physical or conceptual connection between two or more objects. Binary associations are shown as lines connecting two class symbols. Ternary and higher-order associations are shown as diamonds.

6. An actor is someone or something outside the system that either acts on the system - a primary actor - or is acted on by the system - a secondary actor. An actor may be a person, a device, another system or sub-system, or time. Actors represent the different roles that something outside has in its relationship with the system whose functional requirements are being specified. An individual in the real world can be represented by several actors if they have several different roles and goals in regards to a system. These interact with system and do some action on that.

7. Refer Apr/May 2011 – 14(a).

8. The three basic types of attributes are
   – Single-value attributes.
     The single-valued attribute has only one value or state.
   – Multiplicity or multi value attributes.
     The multiplicity or multi valued attribute can have a collection of many values at any point in time.
   – Reference to another object, or instance connection.
     These attributes are required to provide the mapping needed by an object to fulfill its responsibilities, in other words, instance connection model association.

9. It concentrates on what the user does, not what the product does. This means capturing use cases and the tasks users perform, then performing them and their variants as tests. They often are more complex and realistic than error-based tests.
   Scenario-based tests tend to exercise multiple subsystems in a single test, because that is what users do.

10. It is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use.
    – Create a user satisfaction test for your own project
    – Conduct test regularly and frequently
    – Read the comments very carefully, especially if they express a strong feeling.

– Use the information from user satisfaction test, usability test, reactions to prototypes, interviews recorded, and other comments to improve the product. Important benefit of user satisfaction testing is you can continue using it even after the product is delivered.

## PART B

11. (a) (i) ***Inheritance*****:** Inheritance is a process by which one object acquires the properties of another object. Here, the classes are arranged in a hierarchical manner, where the top class is called as the super - class and at the next level the subclasses reside. Subclass inherits all of the properties and methods that are defined in its super class.

Inheritance allows to express similarity.

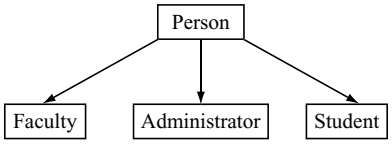



**Fig. 1**

Also, inheritance provides a natural classification for kinds of objects and allows for a commonality of objects to be explicitly taken the advantage in modeling and constructing object system.

Here, the parent class can be called as the super class or base class from which the subclass or derived class or child inherits all or few properties.

***Abstract Class*****:** A class that provides no objects are called as abstract classes.

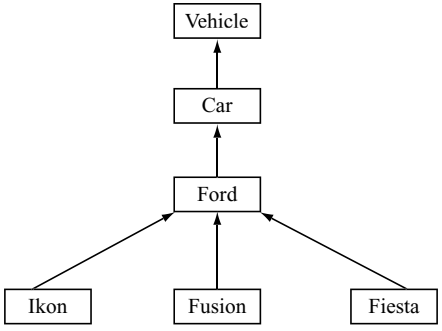



**Fig. 2**

***Example*:** An account class is an abstract class which defines the common behaviours that can be inherited by more specific classes such as checking account and savings account.

***Dynamic inheritance*:** This is the method in which the objects are allowed to change and evolve over time. It refers to the ability to add, delete or change parents from objects at run time.

***Multiple inheritance*:** A class in created by inheriting the properties and methods from more than one super or base class is known as multiple inheritance.

***Generalization*:** Generalization is the relationship between a more general class and a more specific class.

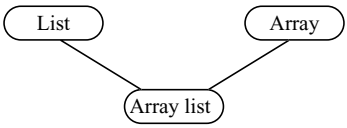



**Fig. 3**

11. (a) (ii) ***Polymorphism*:** Polymorphism is a behaviour that varies depending on the class in which the behaviour is invoked (i.e.) two or more classes can react differently to the same message.

It allows a single operation to have different behaviour in different objects. In other words, polymorphism allows to send identical messages to objects of different classes, each of which responds to the message in a different way.

"Polymorphism is the ability to take more than one form".

***Example*: (1) Driving an automobile**

- With manual transmission
- With automatic transmission

**(2)**

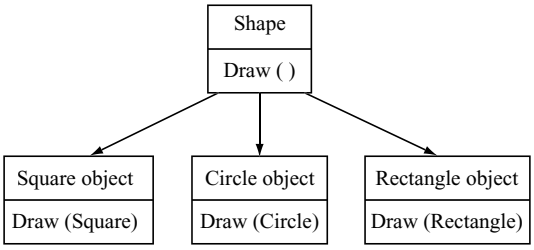



**Fig. 4**

**(3) Open ( )**

- Open a door
- Open a document
- Open a bank account
- Open a book

In each case we are performing different operations. Here, each class knows how that operation takes place. This is polymorphism. Two types of binding are: static binding and dynamic binding.

***State/Early Binding*:** The process or method to be invoked is decided before the run-time is known as static binding. Example: To book a ticket for a journey on 19th February.

***Dynamic/Late binding*:** The method to be invoked for a message is not known until the time of call or run-time is known as dynamic binding. It occurs during the polymorphic calls.

***Example:***

To book a ticket for a journey is the month of February.
To book a textbook in the library.

11. (b) (i)
- Set of activities that transforms user's needs into a software solution that satisfies the needs.
- A process can be divided into a number of sub-processes.

The software development process consists of

- Analysis
- Implementation
- Refinement
- Design
- Testing

The essence of software development process is to transform users' needs into a software solution that satisfies those needs. The main point is to build high quality software.

- Three types of transformation
  Transformation 1 - Analysis
  Transformation 2 - Design
  Transformation 3 - Implementation
  Waterfall Approach
- What • How • Do it • Test • Use

OOSD Life Cycle: Refer May/June 2009 11(b).

11. (b) (ii) **Advantages of Waterfall Approach**

Object-oriented systems development consists of

- Object-oriented analysis
- Object-oriented information modeling
- Object-oriented design
- Prototyping and implementation
- Testing, iteration and documentation

Object-oriented software development encourages the user to view the problem as a system of cooperative objects. Object-oriented analysis shares certain aspects with the structured approach, such as determining the system requirements. One major difference is that the user do not think of data and procedures separately, because objects incorporate both. When developing an object oriented application, 2 basic questions always arise:

- What objects does the application need?
- What functionality should those objects have?

Each object is entirely responsible for itself. For example, an employee object is responsible for things like payroll, printing paychecks and strong data about itself, such as its name, address and SSN.

The first task in Object-oriented analysis is to find the class of objects that will compose the system. At the first level of analysis, the user look at the physical entities in the system. That is, who are the players and how do they cooperate to do the work of the system? These entities could be individuals, organizations, machines, units of information, molecules, pictures of whatever else makes sense in the content of the real-world system. This usually is a very good starting point for deciding what *classes* to design. At this level, you must look at the physical entities in the system. In the process of developing the model, the objects that emerge can help us establish a workable system. Coad and Yourdon have listed the following clues for finding the candidate classes and objects:

- Persons - what role does a person play in the system? For example: customers, employees of which the system needs to keep trace.
- Places - There are physical locations, buildings, stores, sites or offices about which the system keeps information.

- Things or events - These are events, points in time that must be recorded. For example, the system might need to remember when a customer makes an order; therefore, an order is an object. Associated with things remembered are attributes such as who, what, when, where, how or why. For example, some of the data or attributes of *order object* are customer-ID (who), date-of-order when Soup-ID (what), and so on.

We need to identify the hierarchical between superclasses and subclasses. Another task in Object-oriented is to identify the attributes (properties) of objects, such as color, cost and manufacturer. Identifying behavior (methods) is next. What services must a class provide? It allows us to identify the methods a class must contain. Once you identify the overall system's responsibilities and the information it needs is user (can assign each responsibility to the class to which it logically belongs.

Need to model the system's objects and their relationships. For example, the objects in payroll system. Employer, Manager, production worker and temporary office worker. Other objects are part of the system such as the paycheck or production being made and the process being used to make the product. The pay roll program would have different classes of employees corresponding to the different types of employees in the company. Every employee object would know how to calculate its payroll according to its own requirements.

The goal of object-oriented analysis is to identify objects and classes that support the problem domain and the system's requirements. Object-oriented design identifies and defines additional objects and classes that support an implementation of the requirements. For example, during design the user might need to add objects for the user interface for the systems; that is, the data entry windows, browse windows and the like.

**Disadvantages of Waterfall Approach**

The question that must be bothering you now is that with so many advantages at hand, what could be the possible disadvantages of the waterfall model. Well, there are some disadvantages of this widely accepted model too. Let us look at a few of them.

- Ironically, the biggest disadvantage of the waterfall model is one of its greatest advantage. You cannot go back, if the design phase has gone wrong, things can get very complicated in the implementation phase.
- Many times, it happens that the client is not very clear of what he exactly wants from the software. Any changes that he mentions in between may cause a lot of confusion.
- Small changes or errors that arise in the completed software may cause a lot of problem.
- The greatest disadvantage of the waterfall model is that until the final stage of the development cycle is complete, a working model of the software does not lie in the hands of the client.

The waterfall model, as already mentioned, is of course the most widely used model. There are various versions of the same, which allow some waterfall model phases to overlap or feedback to be taken after each phase, which make designing the software a lot simpler. So this, in short, was all about waterfall model advantages and disadvantages. In spite of the disadvantages, the many advantages of this model ensure that it remains one of the most popular models used in the field of software development.

12. (a) Refer Nov/Dec 2009 - 12(b).

12. (b)

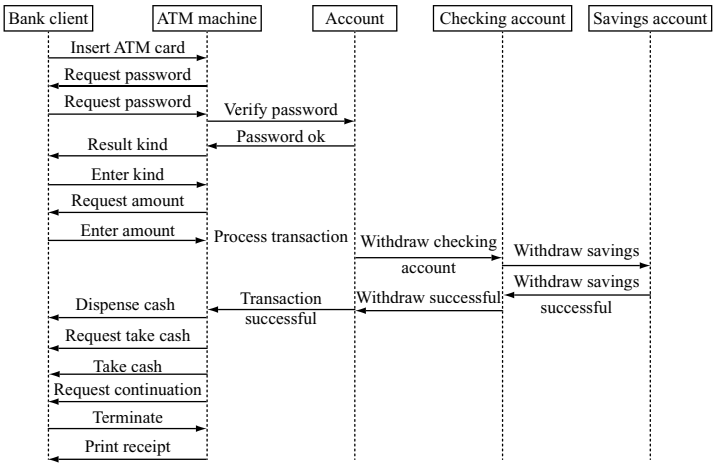



**Fig. 5** Sequence diagram for the withdraw more from checking usecase

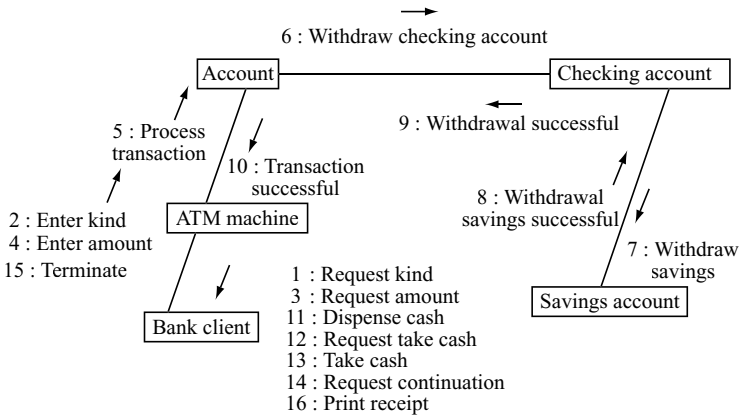



**Fig. 6** The collaboration diagram for the withdraw more from checking usecase

13. (a) Business object analysis is a process of understanding the systems requirements and establishing the goals of an application. The main intent of this activity is to understand users' requirements. The outline of the business object analysis is to identify classes that makeup the business layer and the relationship that play a role in achieving system goals. To understand the user's requirement, we need to find how they "use" the system. This can be accomplish by developing use cases. Usecases are scenarios for understanding system requirements. Domain users or experts are the best authorities. Try to understand the expected inputs and derived responses. Defer unimportant detail until later. State *what* must be done, not *how* it should be done. This, of course, is easier said done. Yet another tool that can be very useful for understanding user's requirement is preparing a prototype of the user interface. Preparation of a prototype work can help you better understand how the system will be used and therefore it is a valuable tool during business object analysis.

The unified approach (UA) steps can overlap each other. The process is iterative, and you may have to backtrack to previously completed steps for another try. Separating the *what* from the *how* simple process. Fully understanding a problem and defining how implement it may require several tries or iterations.

***Usecase drive object-oriented analysis:***

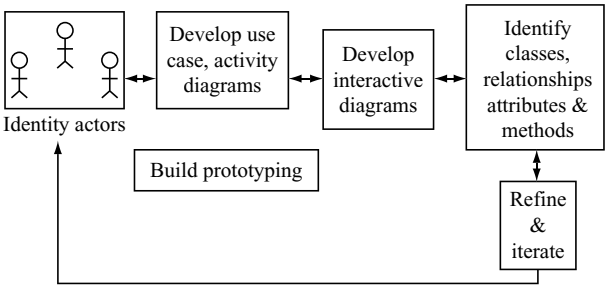



**Fig. 7** The object-oriented analysis in the unified approach(id)

The OOA phase of the unified approach uses actors and usecases to describe the system from the users' perspective. The actor are external factors that interact with the system, *usecases* are scenarios that describe how actors use the system. The usecase identified here will be involved throughout the development process.

The OOA process consists of the following steps (fig. 13):

1. Identify the actors:
   - Who is using the system?
   - or, in the case of a new system, who will be using the system?
2. Develop a single business process model using UML activity diagram.
3. Develop the usecase:
   - What are the users doing with the system?
   - or, in case of the new system, what will users be doing with the system?
   - Usecases provide us with comprehensive documentation of the system under study.
4. Prepare interaction diagrams:
   - Determine the sequence
   - Develop collaboration diagrams
5. Classification-develop a static UML class diagram.
   - Identify classes
   - Identify attributes
   - Identify relationships
   - Identify methods
6. Iterate and define: If needed, repeat the preceding steps.

13. (b) (i) ***Usecase model*:** Usecases are scenarios for understanding system requirements. A use case model can be instrumental in project development, planning and documentation of systems requirement. A usecase is an interaction between users and a system, it captures the goal of the users and the responsibilities of the system to its users. The usecase model describes the uses of the system and shows the courses of events that can be performed. It defines what happens in the system when the usecase is performed. The usecase model tries to systematically identify users of the system and therefore the system's responsibilities. It can discover classes and the relationships among sub systems of the systems. Each use or scenario represents what the uses wants. Each use case must have a name and short textural description, no more than a few paragraphs.

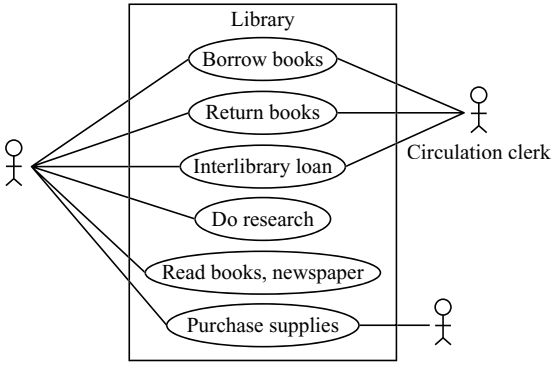



**Fig. 8** Some uses of library (use case model)

Since the usecase model provides an external view of a system or application, it is directly primarily toward the users or the "actors" of the systems, not its supplementers (fig. 14). Jacobson, and Jacobson call the use-case model a "what model".

13. (b) (ii) Guidelines for use cases.

- For each user, find the tasks and functions.
- Name the use cases.
- Describe the use cases briefly.
- Isolate users from actors.
- Isolate actors from other actors.
- Isolate use cases.

14. (a) During the design phase the classes identified in OO Analysis must be revisited with a shift in focus to their implementation. New class or attributes and methods must be added for implementation purpose and user interfaces.
The OO Design process consists of the activities includes:

1. Apply design axioms to design classes, their attributes, methods, associations, structures and protocols.
   1.1 Refine and complete the static UML class diagram by adding details to the UML class diagram. This step consists of the following activities.
      1.1.1 Refine attributes
      1.1.2 Design methods and protocols by utilizing a UML activity diagram to represent the method's algorithm.
      1.1.3 Refine associations between classes (if required)
      1.1.4 Refine class hierarchy and design with inheritance (if required)
   1.2 Iterate and refine again

***Object-oriented design axioms*:** An axiom is a fundamental truth that always is observed to be valid and for which there is no counter example or exception. Axioms may be hypotherized from a large number of observations by noting the common phenomena shared by all cases; they cannot be proven or derived, but they can be invalidated by counter examples or exceptions. A theorem is a proposition that may not be self evident but can be proven from accepted axioms. It is equivalent to a law or principle. A theorem is valid if its reference axioms and deductive steps are valid. Sub's design axioms applied to object-oriented design. Axiom 1 deals with relationships between system components (such as classes, requirements and software components) and Axiom 2 deals with the complexity of design.

- ***Axiom 1*:** The independence axiom. Maintain the independence of components.
- ***Axiom 2*:** The information axiom, minimize the information context of the design.

Axiom 1 states that, during the design process, as we go from requirement and use case to a system component, each component must satisfy that requirement without affecting other requirement.

Let's take a look an example offered by such. You been asked to design a refrigerator door and there are 2 requirements:

The door should provide access to food, and the energy lost should be minimal when the door is opened and closed. Opening the door should be independent of losing energy. Is the vertically hung door a good design? We see that vertically hung door violates axiom 1, because the 2 specific requirements are coupled and are not independent in the proposed design. When, for example, the door is opened to take out milk, cold air in the refrigerator escapes and warm air from the outside enters. What is an uncoupled design that somehow doesn't combine these 2 requirements? Once such uncoupled design of the refrigerator door is a horizontally hinged door, such as used in chest-type freezer when the door is opened to take out milk, the cold air will sit at the bottom and not escape. Therefore, opening the door provides access to the food and is independent of energy loss. This type of design satisfies the first axiom.

Axiom 2 is concerned milk simplicity. Scientific theoreticians often rely on a general rule known as *Ocean's razor*, after idleness of ocean, a 14th century scholastic philosopher. Ocean's razor says that "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straight forwardness".

Ocean's razor has a very useful implication in approaching the design of an object-oriented application. Ocean's razor rule of simplicity in object-oriented terms;

"The besign design usually involve the least complete code but not necessarily the fewest number of classes or methods minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application. In an object-oriented system, the best way to minimize complements is to use inheritance and the systems built-in classes and to add as little as possible to what already is there".

***Corollaries*:** From the 2 design axioms, many corollaries may be derived as a direct consequence of the axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called *design rules*, and all are derived from the 2 basic axioms (fig. 21).

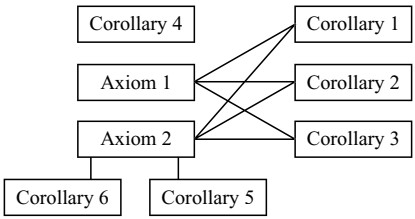



**Fig. 9** The origin of corollaries. corollaries 1, 2, & 3 are from both axioms, whereas corollary 4 if from axiom 1 & corollaries 5 and 6 are from axiom 2

***Corollary 1: Uncoupled design with less information content*:** Highly cohesive objects can improve coupling because only a minimal amount of essential information need by passed between objects.

***Corollary 2: Single purpose*:** Each class must have a single, clearly defined purpose. When you document, you should be able to easily describe the purpose of a class in a few sentences.

***Corollary 3: Large number of simple classes*:** Keeping the classes simple allows reusability.

***Corollary 4: Strong mapping*:** There must be a strong association before the physical system (analysis's object) and logical design (designs object).

***Corollary 5: Standardization*:** Promote standardization by designing interchangeable components and reusing existing classes or components.

***Corollary 6: Design with inheritance*:** Common behavior (methods) must be moved to superclasses. The superclass-subclass structure must make logical sense.

***Corollary 1*:** Uncoupled design with less information context. The main goal here is to maximize object cohesiveness among objects and software components in order to improve coupling because only a minimal amount of essential information need be passed between components.

***Coupling*** is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship. A is coupled with B. Coupling is important when evaluating a design because it helps us focus on an important issue in design. For example, a change to one component of a system should have a minimal impact of other components.

Strong coupling among objects complicates a system, since the class is harder to understand or highly complicates a system, since the class is harder to understand or highly interrelated with other classes. The degree of coupling is a function of

1. How complicated the correction is.
2. Whether the connection refers to the object itself or something inside it.
3. What is being sent or received? OO Design has 2 types of coupling.

***Interaction coupling and inheritance coupling*:** Interaction coupling involves the amount and complexity of messages between components. It also reduces the number of messages sent and received by an object. Inheritance is a form of coupling between super and superclasses. A subclass is coupled to its superclass interms of attributes and methods. High inheritance coupling is desirable.

***Cohesion*:** Coupling deals with interactions between objects or software components. Cohesion reflects the *"single-purposeness"* of an object. Method cohesion like function. Cohesion means that a method should carry only one function. Class cohesion means that all the class's methods and attributes must be highly cohesive, meaning to be used by internal methods or derived classes methods.

***Corollary 2: Single purpose*:** Each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals. When your document a class, you should be able to easily explain its purpose in a sentence or two. If you cannot, then rethink the class and try to subdivide it into more independent pieces.

Keep it simple; to be more precise, each method must provide only one service. Each method should be of moderate size, no more than a page; half a page is better.

***Corollary 3: Large number of simpler classes, reusability*:** A great benefit results from having a large number of simpler classes. You cannot possibly foresee all the future scenarios in which the classes you create will be reused. The less specialized the classes are, the more likely future problems can be solved by a recombination of existing classes, adding a minimal number of subclasses. A class that easily can be understood and reused (or inherited) contributes to the overall system, while a complex, poorly designed class

is just so much dead weight and usually cannot be reused. Keep the following guideline.

'The smaller are your classes, the better are your chance of reusing them in other projects. Large and complex classes are too specialized to be reused'.

Object-oriented design offers a path for producing litraries reusable parts. The emphasis object-oriented design places on encapsulation, modularization and polymorphism suggests reuse rather than building a new.

Coad and Yourdon argue that software reusability rarely is practiced effectively. The primary benefit of software reusability is higher productivity. The software development team that achieves 80% reusability is 4 times as productive as the team achieves 20% reusability. Another form of reusability is using a design pattern.

***Corollary 4: Strong mapping*:** During the design phase, we need to design class-design its methods, its association with other objects, and its new and access classes. A strong mapping links classes identified enduring analysis and classes designed during the design phase. Martin and Odell describe this important issues very elegantly. With OO techniques, the same paradigm is used for analyzer design and implementation. The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects.

***Corollary 5: Standardization*:** To reuse classes, you must have a good understanding of the classes in the OOP environment you are using. The knowledge of existing classes will help you determine what new classes are needed to accomplish the tasks and where you might inherit useful behavior rather than reinvent the wheel.

***Corollary 6: Designing with inheritance*:** When you implement a class, you have to determine its ancestor, what attributes it will have, and what messages will it understand. Then you have to construct its methods and problems. You will choose inheritance to minimize the amount of program instructions. Satisfying these constraints sometimes means that acts inherits from a superclass that may not be obvious at first glance.

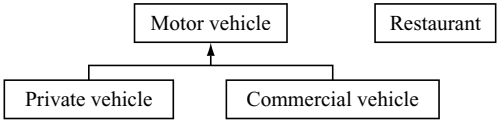


**Fig. 10** The initial single inheritance design

Subclasses of motor vehicle class are private vehicle and commercial vehicle. These are further sub divided into whatever level of specificity seems subclasses of restaurant are designed to reflect their own licensing procedures. This is a simple, easy to understand design, although somewhat limited in the reusability the classes. For example, if in another project you must build a system that models a vehicle assembly plant, the classes from the licensing application are inappropriate since these classes have instructions and data that deals with the legal requirements of motor vehicle license acquisition and renewal. In any case, the design is approved, implementation is accomplished and the system goes into production.

14. (b) A multi database system (MDBS) is database systems that resides unobtrusively on top of existing relational and object databases, and file systems and presents a single database illusion to its users. An MDBS maintains a single global database schema. The local database systems actually maintain all user data. MDBS actually controls multiple gateways (or drivers). This way user can have the benefits of a database with a schema to access data stored in different databases and cross database functionality.

- Primary concept of this model is the relation, which can be thought of as a table. The columns are attributes and the rows are tuples representing individual data objects being stored.
- A relational table should have a primary key.
- A primary key is a combination of one or more attributes whose value clearly locates each row in the table.
- A foreign key is a primary key of one table that is embedded in another table to link the tables.

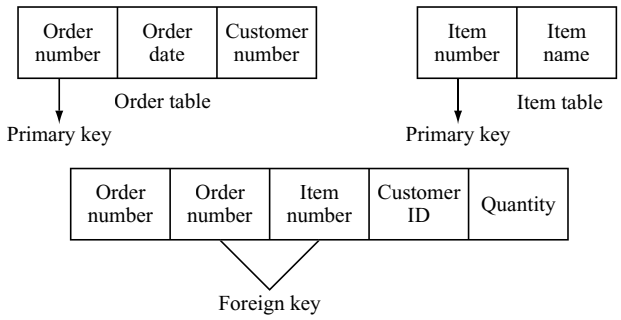



**Fig. 11** The primary and foreign keys in a relational database

***Database Interface:***

- The interface on a database must include a data definition language (DDL), a query, and DML.

***Database Schema and DDL:***

- DDL is the language used to described the structure and relationships between objects stored in a database. This structure of information is termed the database schema.
- SQL commands used to create logical structure are
  - Create schema authorization (creator)
  - Create database (database name)

***Data Manipulation Language and Query Capabilities:***

- A DML is the language that allows users to access and manipulate data organization.
- SQL is widely used for its query capabilities. The query usually specifies.
  - The area of the conversation over which the query is asked.
  - The elements of general interest.
  - The conditions or constraints that apply.
  - The ordering, sorting, or grouping of elements and the constraints that apply to the ordering or grouping.

***Shareability and Transactions of Logical and Physical database*:** Logical database is an abstract view of database structure. Data and objects in the database often need to be accessed and shared by different applications. A transaction is a unit of change in which many individual modifications are aggregated into a single modification that occurs in its entirety or not at all. A transaction is said to commit if all changes can be made successfully to the database and to abort if cancelled because all changes to the database cannot be made successfully. Here atomicity is ensured.

***Concurrency Policy:***

- Every transaction provides each user with a consistent view of the database i.e., the transactions must occur in serial order.
- Concurrency is retained by applying locks.
- Traditional or pessimistic policy provides exclusive access to the object, despite what is done to it.
- Write lock, Read lock, Exclusive lock.

15. (a) **Identifying SQA Issues**

Software Quality Assurance is a good practice that every large scale business should employ. IT related businesses have never hesitated to use SQA to ensure that the application they will release for their users or sell to their customers will live up to their expectations.

On the other hand, there are companies that has opted to forgo with the usage of SQA and relied on application testers just to make sure the application will never have internal errors. The SDLC on the other hand, will ensure the development of the application will work as planned.

Like other development plans, there are issues why developers and companies do not use SQA.

**SQA Cost**

This is the main reason why small scale companies hesitated to use SQA as part of their development plan. SQA is a different entity that works separately from the development team. Hiring another set of developers will mean another set of people to pay for. Developing software takes months to take and if you are a small-scaled company you will definitely have to cut costs.

This problem cannot just be answered with more money and resources. Unfortunately, the developers have to face the fact that SQA cannot just be implemented when everyone has to cut costs. Only the SDLC and the testing team could compensate what the SQA is lacking.

**SQA Complexity**

Years ago, an application could be easily built by a single developer and let the whole world enjoy them. Today, a highly efficient application would take years for a single developer to develop. By the time it could be integrated and implemented, it is already outdated.

The point is, software today is very complicated that team after team will be working on developing a single application and will take them months to develop. SQA has already been here for years and there are models that are not as good as it was.

To answer this concern the SQA team should carefully use CASE tools. These tools could easily summarize the functions. They ensure consistency in the evaluation of the application. That includes documentation of every stage of development.

**SQA Integration**

This is one of the many concerns of the SQA models today. Because application development has been very rapid, the standardization of the application might not be the same as it was five years ago. The result for this is that although it has been deemed as an application developed with SQA, the type of SQA might not live up according to what is expected.

The standardization models might ensure that the application developed ten years ago is good but it might work today. Fortunately, this issue was answered with the development of new standardizations. This ensured that the application was developed according to the plan and today's need. Today's standards usually come with set of tools with an understanding that the application is more complex than it was.

**SQA Objectives**

Software Quality Assurance was created with the following objectives:

***Small to Zero Defects After Installation*** – One of the biggest goals of SQA is to prevent any possible defects when the output is made. Developers and engineers have to use universally approved steps to ensure that the program was built up to expectations but also to prevent errors in the system. Although some standards allow as much as .04 errors in the system, zero-error is still the system's target. When there's zero-error, the program is more likely to have zero crash scenarios. The ability to handle stress of a program is different from the errors it has but crashes usually comes from defects so prevention of defects will most likely yield a continuously working application.

***Customer Satisfaction*** – Everything else will be just nothing if the customers don't like what they see. Part of SQA is to ensure that software development was made according to their needs, wants and exceeding their expectations. Even if the bugs and errors are minimized by the system, customer satisfaction is more important and should be emphasized.

***Well Structured*** – SQA takes care of the stages of application construction. Anyone could be easily build an application and launch it in their environment without any glitches. However, not everyone could easily build an application that could be understood well. SQA ensures that each application are build in an understandable

manner. Their applications could easily be transferred from one developer to another.

15. (b) Usability testing is a black-box testing technique. The aim is to observe people using the product to discover errors and areas of improvement. Usability testing generally involves measuring how well test subjects respond in four areas: efficiency, accuracy, recall, and emotional response. The results of the first test can be treated as a baseline or control measurement; all subsequent tests can then be compared to the baseline to indicate improvement.

- *Performance* – How much time, and how many steps, are required for people to complete basic tasks? (For example, find something to buy, create a new account, and order the item.)
- *Accuracy* – How many mistakes did people make? (And were they fatal or recoverable with the right information?)
- *Recall* – How much does the person remember afterwards or after periods of non-use?
- *Stickiness* – How much time he spends
- *Emotional response* – How does the person feel about the tasks completed? Is the person confident, stressed? Would the user recommend this system to a friend?

**Methods**

Setting up a usability test involves carefully creating a *scenario*, or realistic situation, wherein the person performs a list of tasks using the product being tested while observers watch and take notes. Several other test instruments such as scripted instructions, *paper prototypes*, and pre- and post-test questionnaires are also used to gather feedback on the product being tested. For example, to test the attachment function of an *e-mail* program, a scenario would describe a situation where a person needs to send an e-mail attachment, and ask him or her to undertake this task. The aim is to observe how people function in a realistic manner, so that developers can see problem areas, and what people like. Techniques popularly used to gather data during a usability test include *think aloud protocol*, *Co-discovery Learning* and *eye tracking*.

**Hallway testing**

**Hallway testing** (or **Hall Intercept Testing**) is a general *methodology* of usability testing. Rather than using an in-house, trained group of testers, just five to six *random* people are brought in to test the product, or service. The name of the technique refers to the fact that

the testers should be random people who pass by in the hallway.

Hallway testing is particularly effective in the early stages of a new design when the designers are looking for "brick walls," problems so serious that users simply cannot advance. Anyone of normal intelligence other than designers and engineers can be used at this point. (Both designers and engineers immediately turn from being test subjects into being "expert reviewers." They are often too close to the project, so they already know how to accomplish the task, thereby missing ambiguities and false paths.)

**Remote Usability Testing**

In a scenario where usability evaluators, developers and prospective users are located in different countries and time zones, conducting a traditional lab usability evaluation creates challenges both from the cost and logistical perspectives. These concerns led to research on remote usability evaluation, with the user and the evaluators separated over space and time. Remote testing, which facilitates evaluations being done in the context of the user's other tasks and technology can be either synchronous or asynchronous. Synchronous usability testing methodologies involve video conferencing or employ remote application sharing tools such as WebEx. The former involves real time one-on-one communication between the evaluator and the user, while the latter involves the evaluator and user working separately.

Asynchronous methodologies include automatic collection of user's click streams, user logs of critical incidents that occur while interacting with the application and subjective feedback on the interface by users. Similar to an in-lab study, an asynchronous remote usability test is task-based and the platforms allow you to capture clicks and task times. Hence, for many large companies this allows you to understand the WHY behind the visitors' intents when visiting a website or mobile site. Additionally, this style of user testing also provides an opportunity to segment feedback by demographic, attitudinal and behavioural type. The tests are carried out in the user's own environment (rather than labs) helping further simulate real-life scenario testing. This approach also provides a vehicle to easily solicit feedback from users in remote areas quickly and with lower organisational overheads.

Numerous tools are available to address the needs of both these approaches. WebEx and Go-to-meeting are the most commonly used technologies to conduct a synchronous remote usability test.

However, synchronous remote testing may lack the immediacy and sense of "presence" desired to support a collaborative testing process. Moreover, managing inter-personal dynamics across cultural and linguistic barriers may require approaches sensitive to the cultures involved. Other disadvantages include having reduced control over the testing environment and the distractions and interruptions experienced by the participants' in their native environment. One of the newer methods developed for conducting a synchronous remote usability test is by using virtual worlds.

**User satisfaction testing**

"User satisfaction testing is the process of quantifying the usability test with some measurable attribute of the test, such as functionality, cost or easy of use."

Objective of a user satisfaction test:

As a communication vehicle between designers, as well as between users and designers.

To detect and evaluate changes during the design process.

To provide periodic indication of divergence of option about the current design.

To enable pinpointing specific areas if dissatisfaction for remedy.

To provide a clear understanding of just how the completed design is to be evaluated.

**Guidelines for developing a user satisfaction test:**

Apart from using use-cases to provide you with information to develop a user satisfaction test, users or clients can help with providing attributes to be included in the test.

The user or client can be asked to select a limited number (5 - 10) of attributes by which the final product can be evaluated. For example ease of use, accuracy, functionality, intuitiveness of user interfaces and reliability. Once the attributes have been identifies they can play a crucial role in the evaluation of the final product. The user must use his of hers judgement to rate each attribute selecting a number between 1 and 10, 10 as the most favourable and 1 as the least. Comment often are the most significant part of the test.

The test plans need not be very large. In fact, devoting much time to the plans can be counterproductive. Develop a usability

test plan for the Via Net ATM kiosk by going through the following steps:

***1. Develop test objectives*:** The first step is to develop objectives for the test plan. Generally, test objectives are based on the requirements, usecases, or current or desired system usage. In this case, ease of use is the most important requirement, since the Via Net bank customers should be able to perform their tasks with basically no training and are not expected to read a user manual before withdrawing money from their checking accounts.

The objectives to test the usability of the Via Net Bank ATM kiosk and its user interface:

- 95% of users should be ab6
- le to find out how to withdraw money from the ATM machine without error or any formal training.
- 90% of consumers should be able to operate the ATM within 90 seconds.

***Develop test cases*:** Test cases for usability testing are slightly different from text cases for quality assurance. We are not testing the input and expected output but how users interact with the system. The usecases created during analysis can be used to develop scenarios for the usability test. The usability test scenarios are based on the following usecases.

Deposit Checking
Withdraw Checking
Deposit Savings
Withdraw Savings
Savings "Transaction History
Checking Transaction History

Next we need to select a small number of test participants (6 to 10) who have never before used the kiosk and ask them to perform the following scenarios based on the usecase:

1. Deposit $1056.65 to your checking account
2. Withdraw $40 from your checking account
3. Deposit $200 to your savings account
4. Withdraw $55 from savings account
5. Get your savings account transaction history
6. Get you checking account transaction history

Store by explaining the testing process and equipment to the participants to ease the pressure. To make participants feel

comfortable by emphasizing that you are testing the software, not them. If they become confused or frustrated, it is not reflection on them but the poor usability of the system.

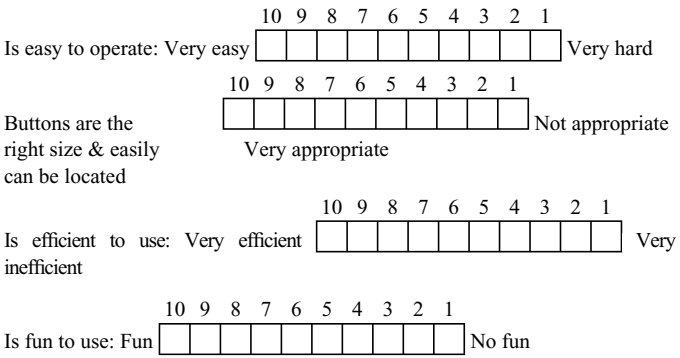
As participants work, record the time they take to perform a task as well as any problems they encounter. In this case, we used the kiosk video camera to record the test results along with a tape recorder. This allowed the design team to reviewed evaluate how the participants interacted with the user interface. For example, look for things such as whether they are finding the appropriate buttons easily and the buttons easily and the buttons are the right size.

Once the test subjects complete their tasks, conduct a user satisfaction test to measure their level of satisfaction with the kiosk. The format of the user satisfaction test is basically the same, but its context is different for the Via Net Bank. The suers, usecases and test objects should provide the attributes to be included in the test. The following attributes have been selected, since the ease of use to the main issue of the user interface:

- Is easy to operate
- Buttons are the right size and easily located
- Is efficient to use
- Is fun to use
- Is visually pleasing
- Provides easy recovery from errors

Based on these attributes, the test shown in fig. can be performed. These attributes can play a crucial role in the evaluation of the final product.

How do you rate the Via Net Bank ATM kiosk interface?



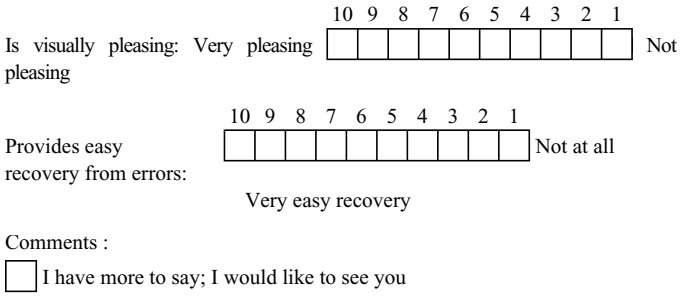



**Fig. 12** A form for the bank system user satisfaction test

***Analyze the tests*:** The final step is to analyze the tests and document the task results. We need to answer questions such as these: What percentage were able to operate the ATM within 90 seconds or without error? Were the participants able to find out how to withdraw money from the ATM m/c with no help? The results of the analysis must be examined.

We need to analyze the results of the user satisfactory tests. The USTS tool similar to it can be used to record and graph the results of user satisfaction tests. A shift in user satisfaction pattern indicates that something in happening and a follow-up interview is needed to find out the reasons for the changes. The UST can be used as a tool for finding out what attributes are important or unimportant. For example, based on the UST, we might find that the users do not agree that the system "is efficient to use", and it got a low score. Alter the follow-up interviews, it became apparent that participant wanted, in addition to entering the amount for withdrawal, to be able to select from a list with predefined values (say $20, $40, $60, $80, $100 and $200). This would speedup the process at the ATM kiosk. Based on the result of the test, the UI was modified to reflect the wishes of the users. You need also to pay close attention to comments, especially if they express a strong feeling. Feelings are facts, the most important facts you have about the users of the system.

# B.E/B.Tech. DEGREE EXAMINATION, NOV/DEC 2009

## Sixth Semester

## Computer Science and Engineering

## CS1402-OBJECT ORIENTED ANALYSIS AND DESIGN

## (Common to Information Technology)

### PART A – (10 × 2 =20 marks)

1. Give the characteristics of object oriented system.
2. What is an object? Give an example.
3. Give a note on patterns and its necessity.
4. Mention the models in Object Modelling Techniques in Rambaugh methodology and its role for describing the system.
5. List out the steps for finding the attributes of a class?
6. Give the hint to identify the attributes of a class?
7. Define axiom along with its types.
8. For the schema employee (emp-id, emp-name, street, city) give the class representation along with the attribute types.
9. Mention the purpose of view layer interface.
10. What are client/server computing? Give two applications which work on this basis?

### PART B – (5 × 16 = 80 marks)

11. (a) Explain and develop the payroll system using the steps of Object oriented approach. (16)

Or

(b) Explain the following
(i) Object Modeling Technique (8)
(ii) Compare Aggregation and Composition with a suitable example. (8)

12. (a) Explain the relationships that are possible among the classes in the UML representation with your example.

Or

(b) What are the various diagrams that are used in analysis and design steps of Booch Methodology? Explain with your own example. (16)

13. (a) Explain the method of identifying the classes using the common class approach with an example. (16)

Or

(b) Consider the Hospital Management System application with the Following requirements
- System should handle the in-patient, out-patient information through receptionist.
- Doctors are allowed to view the patient history and give their prescription.
- There should be a information system to provide the required information.

Give the use case, class and object diagrams. (4 + 8 + 4)

14. (a) With a suitable example explain how to design a class. Give all possible representation in a class (name, attribute, visibility, methods, responsibilities) (16)

Or

(b) Design the access layer for the Students information management which includes personal, fees and mark details. (16)

15. (a) (i) Explain the various testing strategies. (12)

(ii) Give the use cases that can be used to generate the test cases for the Bank ATM application. (4)

Or

(b) (i) How will you measure the user satisfaction? Describe. (6)

(ii) Perform the satisfaction test for any client/server application. (10)

# Solutions

## PART A

1. Object oriented technology is based on a few simple concepts that, when combined, produce significant improvements in software construction. Unfortunately, the basic concepts of the technology often get lost in the excitement of advanced features and advantageous features.
   - Identity
   - Classification
   - Polymorphism
   - Inheritance

2. An object is an instance or specific example of a class. The attributes of the class have specific values within an object of that class; and the operations of a class operate on the attributes of individual objects.

3. A pattern is instructive information that captures the essential structure and insight of a successful family of proven solutions to recurring problem that arises within a certain context and system of forces. Pattern solves a problem, is a proven concept, describes relationships, and has significant human component. The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems.

4. OMT separates modeling into three different parts:
   - An object model, presented by the object model and the data dictionary
   - A dynamic model, presented by the state diagrams and event flow diagrams
   - A functional model, presented by data flow and constraints

5. Attributes are things an object must remember such as color, cost and manufacturer. Identifying attributes of a system's classes starts with understanding the system's responsibilities. Answering the questions like, what information about an object should we keep track of? And what services must a class provide? , will help us to identify attributes.

6. - Attributes usually correspond to nouns followed by preposition phrases. Attributes also may correspond to adjectives or adverbs.
   - Keep the class simple; state only enough attributes to define the object state.
   - Attributes are less likely to be fully described in the problem statement.

- Omit derived attributes. They should be expressed as a method.
- Do not carry excess identification.

7. An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. The axioms cannot be proven or derived but they cannot be invalidated by counter examples or exceptions. There are two design axioms applied to object-oriented design. Axiom 1 deals with relationships between system components and Axiom 2 deals with the complexity of design.
   - Axiom 1: The independence axiom. Maintain the independence of components.
   - Axiom 2: The information axiom. Minimize the information content of the design.

8. There are three types of attributes, namely, single-value attributes, multiplicity or multivalue attributes and reference to another object, or instance connection. Attributes like emp-id, emp-name, street, city are single-value type. If we want to keep track of names of people who have called a customer support line for help, use the multivalue attributes. A person might have one or more bank account. Hence a account has zero to many instance connections.

9. The user interface employs one or more windows. Each window should server clear, specific purpose. Windows are used for the following purposes
   - Forms and data entry windows
   - Dialog boxes and error messages
   - Command button layout
   - Application windows
   - Using colors and fonts

10. In client-server computing the calling module becomes the "client" and the called module becomes the "server". The connectivity allows applications to communicate transparently with other programs (process) regardless of their locations.

    The client is a process (program) that sends a message to a server process requesting the server to perform a task (service). Client programs usually manage the user interface portion of the application; validate data entered by the user and dispatch requests to server programs.

    A server process (program) fulfills the client request by performing the task requested. Server programs generally perform database retrieval and updates, manage data integrity and dispatch responses to client requests.

    The Bank Processing System and Employee Information System works on the client/server computing.

**PART B (5 × 16 = 80 marks)**

11. (a) Consider a payroll program that processes employee records at a small manufacturing firm. This company has three types of employees:
    1. Managers: receive a regular salary.
    2. Office Workers: receive an hourly wage and are eligible for overtime after 40 hours.
    3. Production Workers: are paid according to a piece rate.

**Structured Approach:**

```
FOR EVERY EMPLOYEE DO
    BEGIN
        IF employee = manager THEN
            CALL computeManagerSalary
        IF employee = office worker THEN
            CALL computeOfficeWorkerSalary
        IF employee = production worker THEN
            CALL computeProductionWorkerSalary
    END
```

If new classes of employees are added, such as temporary office workers ineligible for overtime or junior production workers who receive an hourly wage plus a lower piece rate, the main logic of the application must be modified to accommodate these requirements:

```
FOR EVERY EMPLOYEE DO
    BEGIN
        IF employee = manager THEN
            CALL computeManagerSalary
        IF employee = office worker THEN
            CALL computeOfficeWorker_salary
        IF employee = production worker THEN
            CALL computeProductionWorker_salary
        IF employee = temporary office worker THEN
            CALL computeTemporaryOfficeWorkerSalary
        IF employee = junior production worker THEN
            CALL computeJuniorProductionWorkerSalary
    END
```

**The Object-Oriented Approach:**

Object oriented systems development consists of

- Object –oriented analysis
- Object –oriented information modeling
- Object –oriented design

- Prototyping and implementation
- Testing, iteration and documentation

When developing an object-oriented application, two basic questions always arise:

- What objects does the application need?
- What functionality should those objects have?

The goal of OO analysis is to identify objects and classes that support the problem domain and system's requirements. Some general candidate classes are:

- Persons
- Places
- Things or events

**Class Hierarchy**

- Identify class hierarchy
- Identify commonality among the classes
- Draw the general-specific class hierarchy.

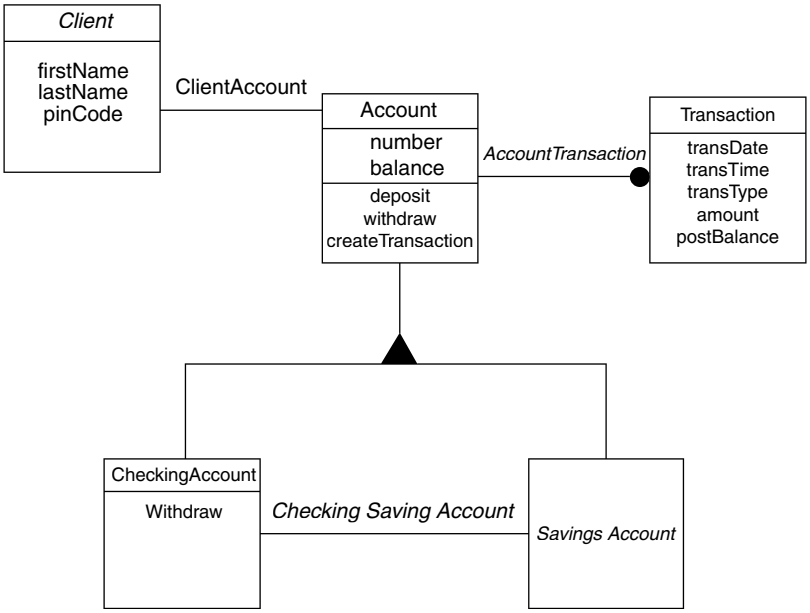
```
FOR EVERY EMPLOYEE DO
        BEGIN
            employee computePayroll
        END
```

11. (b) (i) The object modeling technique (OMT) presented by Jim Rumbaugh and his coworkers describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. Details such as class attributes, method, inheritance, and association also can be expressed easily. Finally, a process description and consumer-producer relationships can be expressed using OMT's functional model. OMT consists of four phases, which can be performed iteratively:
    1. **Analysis:** The results are objects and dynamic and functional models.
    2. **System design:** The results are a structure of the basic architecture of the system along with high-level strategy decisions.
    3. **Object design:** This phase produces a design document, consisting of detailed objects static, dynamic, and functional models
    4. **Implementation:** This activity produces reusable, extendible, and robust code.

OMT separates modeling into three different parts:

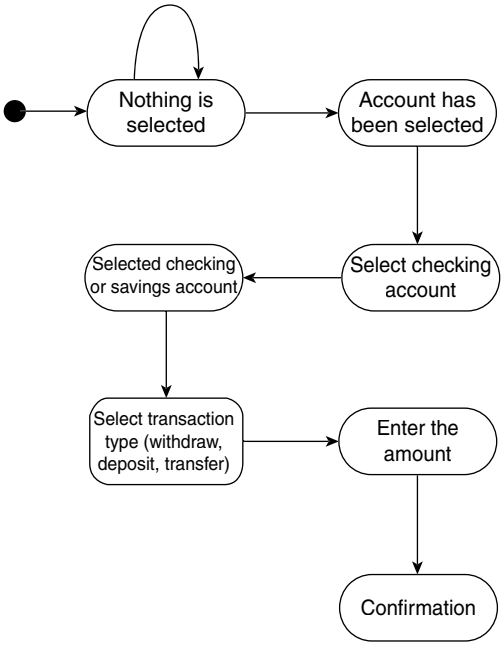
1. An object model, presented by the object model and the data dictionary
2. A dynamic model, presented by the state diagrams and event flow diagrams
3. A functional model, presented by data flow and constraints

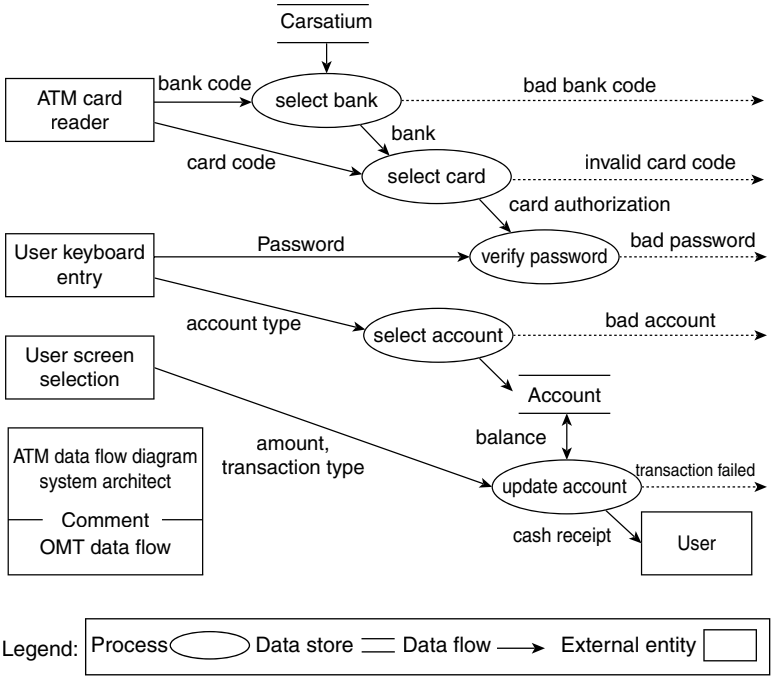
**Object model:** The object model represents the static and most stable phenomena in the modeled domain. Main concepts are classes and associations, with attributes and operations. Aggregation and generalization are predefined relationships. Figure below shows the OMT object model of a bank system



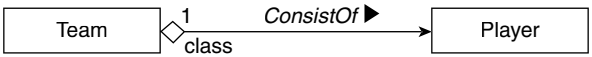

**Dynamic model:** The dynamic model represents a state/transition view on the model. Main concepts are states, transitions between states, and events to trigger transitions. Actions can be modeled as occurring within states. Generalization and aggregation are predefined relationships. Figure below shows the state transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

**Functional model:** The functional model handles the process perspective of the model, corresponding roughly to data flow diagrams. Main concepts are process, data store, data flow, and actors. Figure shows the OMT DFD of the ATM system.


Nothing is selected
Account has been selected
Select checking account
Selected checking or savings account
Select transaction type (withdraw, deposit, transfer)
Enter the amount
Confirmation


Carsatium
ATM card reader
bank code
select bank
bad bank code
bank
card code
select card
invalid card code
card authorization
User keyboard entry
Password
verify password
bad password
account type
select account
bad account
User screen selection
Account
balance
ATM data flow diagram system architect
Comment
OMT data flow
amount, transaction type
update account
transaction failed
cash receipt
User
Legend:
Process
Data store
Data flow
External entity

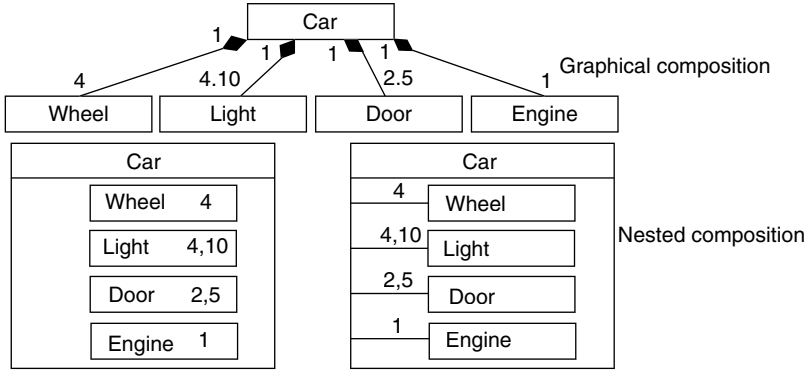
11. (b) (ii) Aggregation is a form of association. A hollow diamond is attached at the end of the path to indicate aggregation.




A-part-of relationship, also called aggregation, represents the situation where a class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently. Two major properties of a-part-of relationship are:

1. **Transitivity:** If A is part of B and B is part of C, then A is part of C.
   Example: a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of car
2. **Antisymmetry:** If A is part of B, then B is not part of A.
   Example: an engine is part of part of a car, but a car is not part of an engine.

Composition, also known as the a-part-of, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a part-whole relationship. The UML notation for composition is a solid diamond at the end of a path. The UML provides a graphically nested form that is more convenient for showing composition. The figure below represents the different ways to show composition




12. (a) In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements. Relationships in class diagrams show the interaction between classes and classifiers. Such relationships indicate the classifiers that

are associated with each other, those that are generalizations and realizations, and those that have dependencies on other classes and classifiers.

The following topics describe the relationships that you can use in class diagrams:

- **Abstraction relationships:** An abstraction relationship is a dependency between model elements that represents the same concept at different levels of abstraction or from different viewpoints. Abstraction relationships to a model in several diagrams, including use-case, class, and component diagrams can be added.
- **Aggregation relationships:** In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier.
- **Association relationships:** In UML models, an association is a relationship between two classifiers, such as classes or use cases, which describes the reasons for the relationship and the rules that govern the relationship.
- **Association classes:** In UML diagrams, an association class is a class that is part of an association relationship between two other classes.
- **Binding relationships:** In UML models, a binding relationship is a relationship that assigns values to template parameters and generates a new model element from the template.
- **Composition association relationships:** A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier.
- **Dependency relationships:** In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier. Dependency relationships in class diagrams, component diagrams, deployment diagrams, and use-case diagrams to indicate that a change to the supplier might require a change to the client can be used.
- **Directed association relationships:** In UML models, directed association relationships are associations that are navigable in only one direction.
- **Element import relationships:** In UML diagrams, an element import relationship identifies a model element in another package, and allows the element in the other package to be referenced by using its name without a qualifier.

- **Generalization relationships:** In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.
- **Interface realization relationships:** In UML diagrams, an interface realization relationship is a specialized type of implementation relationship between a classifier and a provided interface. The interface realization relationship specifies that the realizing classifier must conform to the contract that the provided interface specifies.
- **Instantiation relationships:** In UML diagrams, an instantiation relationship is a type of usage dependency between classifiers that indicates that the operations in one classifier create instances of the other classifier.
- **Package import relationship:** In UML diagrams, a package import relationship allows other namespaces to use unqualified names to refer to package members.
- **Realization relationships:** In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Several clients can realize the behavior of a single supplier. Realization relationships in class diagrams and component diagrams can also be used.
- **Usage relationships:** In UML modeling, a usage relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation.

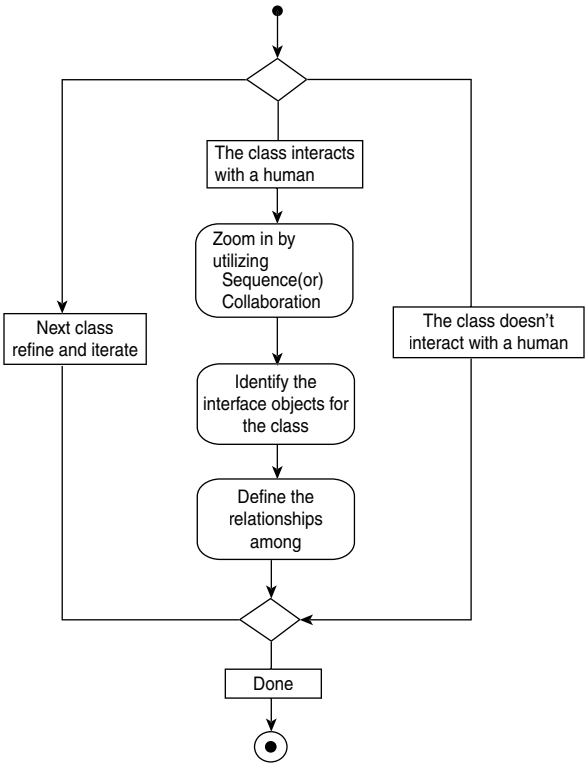
12. (b) The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm. It covers the analysis and design phases of an object-oriented system. Booch sometimes is criticized for his large set of symbols. The Booch method consists of the following diagrams:
    1. Class diagrams
    2. Object diagrams
    3. State transition diagrams
    4. Module diagrams
    5. Process diagrams
    6. Interaction diagrams

    The Booch methodology prescribes two processes

1. Macro development process
2. Micro development process

**1. The Macro Development Process:**

The macro process serves as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system. Such management is interested less in the actual object-oriented design than in how well the project corresponds to the requirements set for it and whether it is produced on time.



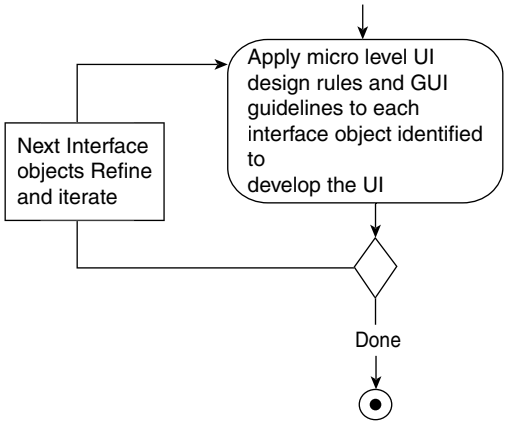

The macro development process consists of the following steps:

1. **Conceptualization:** Establish the core requirements and develop a prototype.
2. **Analysis and development of the model:** Use the class diagram to describe the roles and responsibilities of objects. Use the object diagram to describe the desired behavior of the system.

3. **Design or create the system architecture:** Use the class diagram to decide what classes exist and how they relate to each other, the object diagram to decide what mechanisms are used, the module diagram to map out where each class and object should be declared, and the process diagram to determine to which processor to allocate a process.
4. **Evolution or implementation:** Refine the system through much iteration.
5. **Maintenance:** Make localized changes to the system to add new requirements and eliminate bugs

**2. The Micro Development Process:**

The micro process is a description of the day-to-day activities by a single or small group of software developers.




The micro development process consists of the following steps:

1. Identify class and objects
2. Identify class and objects semantics
3. Identify class and object relationships
4. Identify class and object interfaces and implementation.

13. (a) The common class patterns approach is based on a knowledge base of the common classes that have been proposed researchers. The patterns used for finding the candidate class and object are:
    - Concept class
    - Events class
    - Organization class
    - People class

- Places class
- Tangible things and devices class

**1. Name: Concept class:** A concept is a particular idea or understanding that we have for our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communication. Privately held ideas or notions are called conceptions

E.g.: Performance is an example of concept class object

**2. Name: Event Class:** Events classes are points in time must be recorded. Things happen, usually to something else at a given data and time. Associated attributes are who, what, when, where, why, how etc

E.g.: Landing, interrupt, request and order are possible events

**3. Name: Organization Class:** An organization class is a collection of people, resources, facilities or groups to which the user belongs.

E.g.: An accounting department might be considered a potential class

**4. Name: People class (also known as person, roles and roles played class):** People class represents the different roles users play in interacting with the application. People carry out some function. Person can be divided into two types

1. Those representing users of the system
2. Those representing people who do not use the system

E.g.: Employee, client, teacher

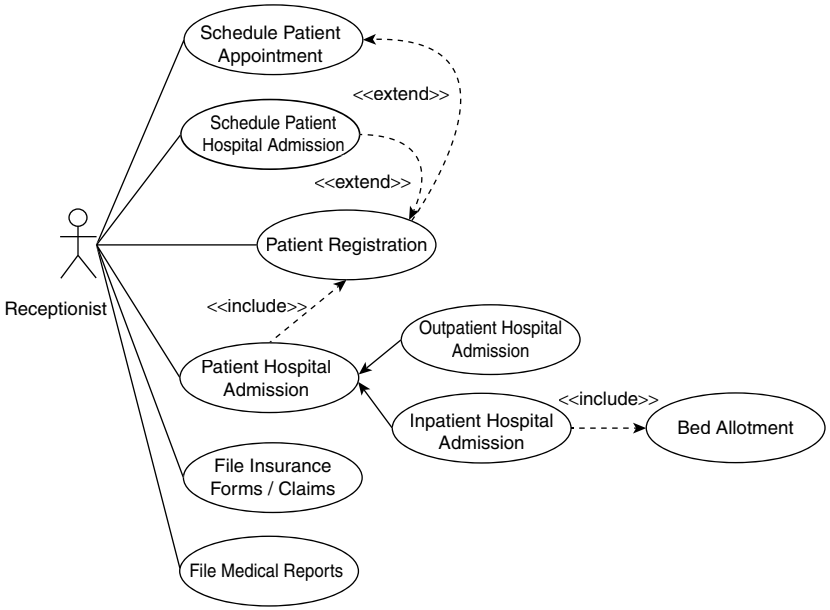
**5. Name: Places Class:** Places are physical location that the system must keep information about.

E.g.: Building, stores, sites, offices

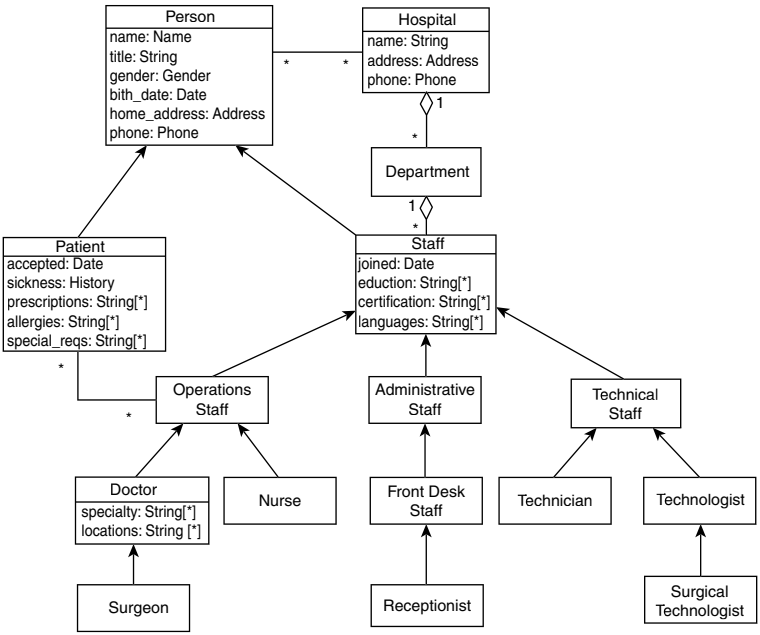
**6. Name: Tangible things and devices class:** This class includes physical objects or groups of object that are tangible and devices with which the application interacts.

Eg. Cars – eg for tangible things & Pressure – eg for an device

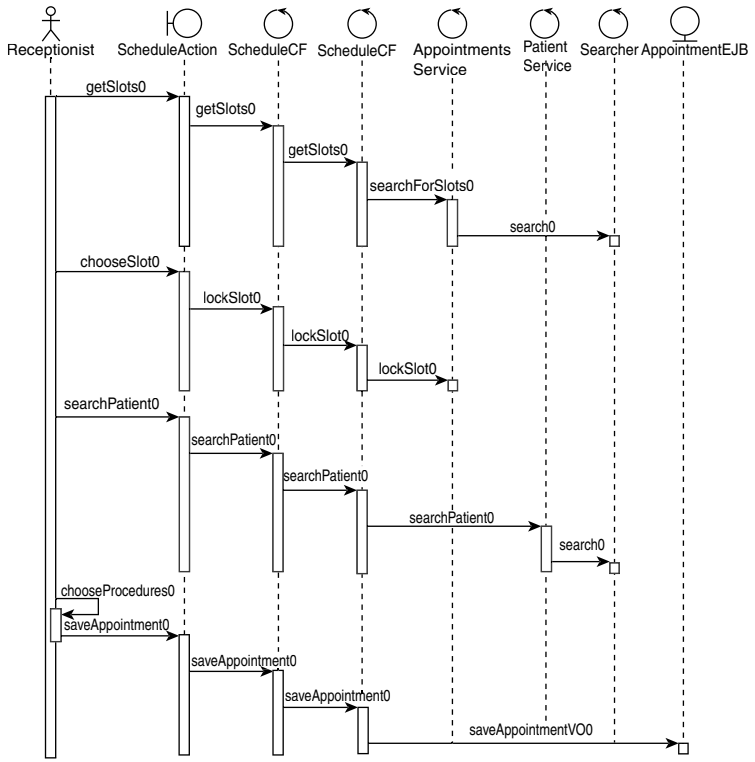
13. (b) Use Case diagram:




Class diagram:



Object diagram:




14. (a) Object-oriented design requires taking the object identified during object-oriented analysis and designing classes to represent them. As a class designer, we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other classes.

**Class visibility: Designing well-defined public, private and protected protocols**

In designing methods or attributes for classes, we are confronted with two problems. One is the protocol or interface to the class operations and its visibility and the other is how it is implemented. The class's protocol or the messages that a class understands, can be hidden from other objects (private protocol) or made available to other objects (public protocol). Public protocols define the functionality and external messages of an object. Private protocols define the implementation of an object.

A class might have a set of methods that it uses only internally, messages to itself. This private protocol of the class, includes messages that normally should not be sent from other objects. Here only the class itself can use the methods.The public protocol defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes. If the methods or attributes can be used by the class itself (or its subclasses) a protected protocol can be used. Here subclasses can used the method in addition to the class itself. The lack of well-designed protocol can manifest itself as encapsulation leakage. It happens when details about a class's internal implementation are disclosed through the interface.

**Designing classes: Refining attributes**

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute is enough. But in the design phase, detailed information must be added to the model. The 3 basic types of attributes are:

- Single-value attributes
- Multiplicity or multivalue attributes
- Reference to another object or instance connection

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes. Single value attribute has only one value or state. E.g.: Name, address, salary. Multiplicity or multivalue attribute can have a collection of many values at any time. E.g.: If we want to keep tact of the names of people who have called a customer support line for help. Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities. E.g.: A person may have one or more bank accounts. A person has zero to many instance connections to Accounts. Similarly, an Account can be assigned to one or more persons(i.e.) joint account. So an Account has zero to many instance connection to Persons.

**Designing methods and protocols:**

A class can provide several types of methods:

- **Constructor:** Method that creates instances (objects) of the class
- **Destructor:** The method that destroys instances
- **Conversion Method:** The method that converts a value from one unit of measure to another.
- **Copy Method:** The method that copies the contents of one instance to another instance

- **Attribute set:** The method that sets the values of one or more attributes
- **Attribute get:** The method that returns the values of one or more attributes
- **I/O methods:** The methods that provide or receive data to or from a device
- **Domain specific:** The method specific to the application.

**Packages and Managing Classes**

A package groups and manages the modeling elements, such as classes, their associations and their structures. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model elements. A package provides a hierarchy of different system components and can reference other packages. Classes can be packaged based on the services they provide or grouped into the business classes, access classes and view classes.

(b) The main idea behind creating an access layer is to create a set of classes that know how to communicate with the places where the data actually reside. Regardless of where the data reside, whether it be a file, relational database, mainframe, Internet, DCOM or via ORB, the access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. These classes also must be able to translate the data retrieved back into the appropriate business objects.

The access layer's main responsibility is to provide a link between business or view objects and data storage. Three-layer architecture is similar to 3-tier architecture. The view layer corresponds to the client tier, the business layer to the application server tier and the access layer performs two major tasks:

**Translate the request:** The access layer must be able to translate any data related requests from the business layer into the appropriate protocol for data access.

**Translate the results:** The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back into the business layer. Here design is tied to any base engine or distributed object technology such as CORBA or DCOM. Here we can switch easily from one database to another with no major changes to the user interface or business layer objects. All we need to change are the access classes methods.

A Date Base Management System (DBMS) is a set of programs that enables the creation and maintenance (access, manipulate,

protect and manage) of a collection of related data. The purpose of DBMS is to provide reliable, persistent data storage and mechanisms for efficient, convenient data access and retrieval.

Persistence refers to the ability of some objects to outlive the programs that created them. Object lifetimes can be short for local objects (called transient objects) or long for objects stored indefinitely in a database (called persistent objects).

Most object-oriented languages do not support serialization or object persistence, which is the process of writing or reading an object to and from a persistence storage medium, such as disk file. Unlike object oriented DBMS systems, the persistent object stores do not support query or interactive user interface facilities. Controlling concurrent access by users, providing ad-hoc query capability and allowing independent control over the physical location of data are not possible with persistent objects.

**Object store and persistence:**

Atkinson describes 6 broad categories for the lifetime of a data.

1. Transient results to the evaluation of expressions
2. Variables involved in procedure activation
3. Global variables and variables that are dynamically allocated
4. Data that exist between the execution of a program
5. Data that exist between the versions of a program
6. Data that outlive a program.

The first 3 are transient data, data that cease to exist beyond the lifetime of the creating process. The other 3 are non-transient, or persistent data. The programming languages provide excellent support for transient data. The non-transient data are well supported by DBMS or a file system.

**Creating an access class for the student information management:**

Apply access layer design to identify the access classes

1. Determine if a class has persistent data
2. Mirror the business class package.
3. Define relationships

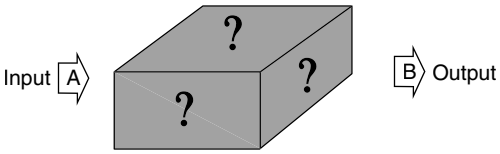
The student class has the following attributes
studentid
firstname
lastname
class
feesdetails
The retrieve method sends a message to the object to get the client information. The update method updates or changes attributes.

The marks class has the following attributes
studentid
classtestmark
internalmark
externalmark
with studentid as reference we enter the classtestmark and update the internalmark. externalmark is entered and the result is calculated for a particular student.

15. (a) (i) First, we must
There are four types of testing strategies. They are:
1. Black Box Testing
2. White Box Testing
3. Top-down Testing
4. Bottom-up Testing

**1. Black Box Testing**

- In a black box, the test item is treated as "black" whose logic is unknown.
- All that's known is what goes in and what comes out, the input and output
- Black box test works very nicely in testing objects in an O-O environment.




**2. White Box Testing**

- White box testing assumes that specific logic is important, and must be tested to guarantee system's proper functioning.
- One form of white box testing is called path testing.
- It makes certain that each path in a program is executed at least once during testing.

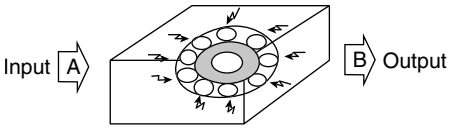
One form of white box testing, called **path testing**, makes certain that each path in a object's method is executed at least once during testing. Two types of path testing are:

(i) Statement testing coverage: test every statement in the object's method by executing it at least once.

(ii) Branch testing coverage: perform enough tests to ensure that every branch alternative has been executed atleast once under some test.

**3. Top-down Testing**

- It assumes that the main logic of the application needs more testing than supporting logic.




**4. Bottom-up Approach**

- It takes an opposite approach.
- It assumes that individual programs and modules are fully developed as standalone processes.
- These modules are tested individually, and then combined for integration testing.

15. (a) (ii) The lists of use cases that can be used to generate the test cases for the bank ATM applications are as follows
    - Bank Transaction
    - Checking Transaction
    - Deposit Checking
    - Deposit Savings
    - Savings Transaction History
    - Withdraw Checking
    - Withdraw Savings
    - Valid/Invalid PIN

    (b) (i) User satisfaction testing is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost or ease of use. Usability can be assessed by defining the measurable goals, such as
    - 95% of users should be able to find how to withdraw money from the ATM machine without error and with no formal training.
    - 70% of all users should experience the new function as "a clear improvement over the previous one"
    - 90% of consumers should be able to operate the VCR within 30 minutes.

    A positive side effect of testing with a prototype is that you can observe how people actually use the software. Another tool that can assist us in developing high-quality software is measuring and monitoring user satisfaction during software development, especially during the design and development of the user interface.

Gause and Weinberg have developed a user satisfaction test that can be used along with usability testing.

The principal objectives of the user satisfaction test are

- To act as a communication vehicle between designers, as well as between users and designers.
- To detect and evaluate changes during the design process.
- To provide a periodic indication of divergence of opinion about the current design.
- To enable pinpointing specific areas of dissatisfaction for remedy.
- To provide a clear understanding of just how the completed design is to be evaluated.

The guidelines for developing a user satisfaction test is that the format of every user satisfaction test is basically the same, but its content is different for each project.

Commercial off-the-shelf (COTS) software tools are already written and few are available for analyzing and conducting user satisfaction tests. The user satisfaction test spreadsheet (USTS) automates many bookkeeping tasks and can assist in analyzing the user satisfaction test for a particular project. One use of a tool like this is that it shows patterns in user satisfaction level.

15. (b) (ii) The usability test plan for the ATM system is an example of an client/srever system which are detailed as follows

**Develop test objectives:** Test objectives are based on the requirements, use cases, or current or desired system usage. In this case, ease of use is the most important requirement. The Objectives are as follows:

- 95% of users should be able to find how to withdraw money from the ATM machine without error and with no formal training.
- 90% of consumers should be able to operate the VCR within 30 minutes.

**Develop test cases:** The usability test scenarios are based on the following use cases

1. Deposit checking
2. Withdraw checking
3. Deposit savings
4. Withdraw saving
5. Saving transaction history
6. Checking transaction history

Next, we need to select a small number of test participants and asked them to perform the following scenarios

1. Deposit Rs 3000 to your checking account
2. Withdraw Rs 2000 from your checking account
3. Deposit Rs 200 to your savings account
4. Withdraw Rs 100 from your savings account
5. Get your savings account transaction history
6. Get your checking account transaction history

As the participants work record the time they take to perform a task as well as any problems they encounter. Use video camera and a tape recorder to record the test results. Once the test subjects complete their task conduct a user satisfaction test to measure their level of satisfaction. The users use cases and test objects should provide the attributes to be included in the test. Here the following attributes have been selected.

- Is Easy to operate
- Buttons are the right size and easily located.
- Is efficient to use
- Is fun to use
- Is visually pleasing
- Provides easy recovery from errors.

Analyze the tests: The final step is to analyze the tests and document the test results. We need answer the questions such as these as follows:

- What percentage was able to operate the ATM within 90 seconds or without error?
- Were the participants able to find out how to withdraw money from the ATM machine with no help?

The results of the analysis must be examined. And then analyze the results of user satisfaction tests. The user satisfaction test can be used as a tool for finding out what attributes are important or unimportant. Also pay close attention to comments, if they express a strong feeling. Feelings are important facts about the users of the system.

The main side effect of developing user satisfaction tests is that you benefit from it even if the test is never administered to anyone; it still provides useful information. Performing the test regularly helps keep the user actively involved in the system development. It also helps us stay focused on the users' wishes.

# B.E./B.Tech. DEGREE EXAMINATION
# MAY/JUNE 2009

**Sixth Semester**

**Computer Science and Engineering**

**CS 1402 - OBJECT ORIENTED ANALYSIS AND DESIGN**

## (Common to Information Technology)

**Time: Three hours** **Maximum: 100 Marks**

**Answer ALL questions**

**PART A (10 · 2 = 20 Marks)**

1. Define abstraction.
2. What is the software development process?
3. What is a design pattern?
4. What are the primary goals in the design of UML?
5. What is the purpose of analysis?
6. Define classification.
7. What is meant by an axiom?
8. Define middleware.
9. Distinguish between white box testing and black box testing.
10. Define usability testing.

## PART B (5 · 16 = 80 Marks)

11. (a) Compare the object oriented approach with the structural approach.

Or

(b) Explain the different phases of OOSD life cycle in detail.

12. (a) (i) Briefly describe the Booch development process.
(ii) What is framework? Describe the difference between patterns and frameworks.

Or

(b) (i) What are phases of OMT? Briefly describe each phase.
(ii) Explain the components of an activity diagram with an example.

13. (a) (i) Describe the guidelines for finding the use case.
(ii) Describe the noun phrase approach for identifying the classes with example.

Or

(b) (i) What are the guidelines for defining attributes? How would you identify a super-sub class structure?
(ii) What is CRC? Explain how it is used to identify classes and their responsibilities.

14. (a) (i) What is corollary? Explain different types of corollaries in details.
(ii) Describe how design axiom help to avoid pitfalls.

Or

(b) (i) Explain the process of creation of access layer with example.
(ii) Describe client server computing.

15. (a) (i) Explain the guidelines for developing quality assurance test cases.
(ii) Explain the activities involved of designing view layer classes.

Or

(b) (i) Describe the guideline for developing test plans.
(ii) What are the objectives of user satisfaction test? Explain the guideline for development the user satisfaction test.

# Solutions

## PART A

1. Abstraction is the principle of ignoring those aspects of a subject that are not relevant to the current purpose. It focuses on the outside view of an object and server to separate an object's essential behaviour from its implementation.

2. A software development process is a series of processes that, if followed, can lead to the development of an application. The software processes describe how the work is to be carried out to achieve the original goal based on the system requirements. Each process consists of a number of steps and rules that should be performed during development.

3. A device that allows systems to share knowledge about their design by describing commonly recurring structures of communicating components that solve a general design problem within a particular context.

4. (1) Provides users with a ready-to-use, expressive visual modeling language.
   (2) Provides extensibility and specialization mechanism to extend core concepts.
   (3) Independent of particular programming language and development processes.
   (4) Supports-higher-level development concepts, such as collaborations, frameworks, patterns and components.
   (5) Integrates best design practices and methodologies.

5. • Understanding the problem domain and the system's responsibilities in the ceser's perspective.
   • Analysis results in a model of system that aims to be correct, complete, consistent and verifiable.
   • Analysis is the process of transforming a problem definition from an unclear set of facts and legend into a logical statement of system's requirements.

6. Classification is the process of checking whether an object belongs to a category (or) classes, if so isolate them under that category.

7. Refer Nov/Dec 2008 - 14(a)(ii).

8. The key element of connectivity is the network operating system (NOS). It provides services such as routing, distribution, messages, filing and printing and network management.

9.

| | Black box | White box |
|---|---|---|
| 1. | Represents a system whose inside workings are not available for inspection. | Logic is important and must be tested to guarantee the system's proper functioning. |
| 2. | Can be used for scenario-based testing. | Used mainly in error-based testing. |
| 3. | Test item is treated as black. *Ex:* Users requirements manual. | Test items are treated as statement and branch. *Ex:* Path testing |

10. Usability testing measure the case of use as well as the comfort and satisfaction users have with the system.

## PART B

11. (a) **Structural approach**

- It requires different styles and methodologies for each step of process.
- Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds.
- It slow down the development process.
- It increases the size of projects.
- Chances for errors introduced in moving from one language to another.

**Object oriented approach**

- It uses the same language to talk about analysis design, programming and data base design.
- It reduces the level of complexity and redundancy and make for clearer more robust system development.

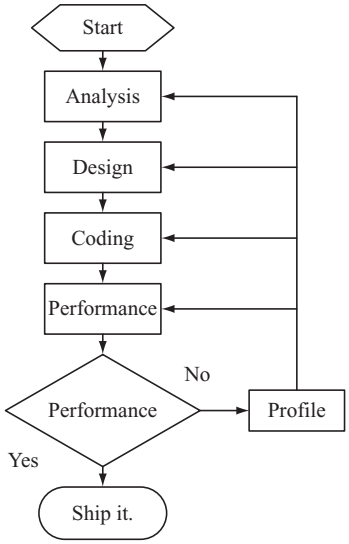
| Structural approach | Object oriented approach |
|---|---|
| Focuses on the functions of the system. | Focuses on the objects (which combines data and functionality). |
| Software is viewed as collection of function and isolated data. | Software is viewed as a collection of objects which encapsulate both methods and data combined together. |

11. (b) The object-oriented software development life cycle consists of four phases.

- Object-oriented Analysis
- Object-oriented design
- Coding
  Testing **Implemention**

**Performance process**

Below flowchart shows the steps in the performance process. These mirror the phases of the traditional OOSD model, with one addition. This new phase is performance profiling-determining the performance characteristics of a software system.

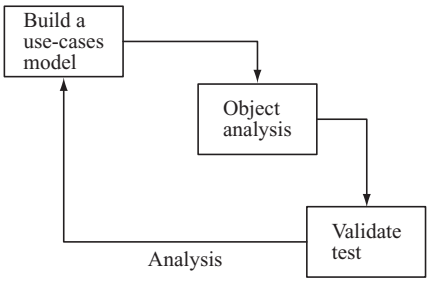


Popular OOSD methodologies define particular tasks to be completed during each development phase. As we discuss each phase, we will introduce some additional performance-related tasks that are critical to producing high-performance software. These complement, rather than replace, those required by the OOSD methodologies.

### Analysis

Analysis is the first stage of the development process. It is also the first step in producing high-performance software. Analysis is central to the performance tuning process.

The analysis part having these below steps –

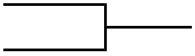



### Object-Oriented Design

Object-oriented design plays a major role in the performance process. While there are many factors that contribute to a good design, and all of them are important, there is one concept that is especially critical to creating high-performance systems. This concept is called encapsulation.

Making encapsulation part of your design from the start enables you to:

- Quickly evaluate different algorithms and data structures to see which is most efficient.
- Easily evolve your design to accommodate new or changed requirements.



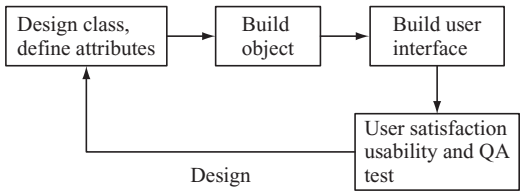
## Implementation

### 1. Coding

Clearly, the way we write our code has an impact on how our system performs.

**2. Testing (Performance)**

Quality assurance is an important part of the software development process. In this context, however, we want to focus on performance testing rather than quality testing. Performance testing is really about bench marking. These bench marks should based on our performance requirements.




12. (a) (i) Booch methodology assists in designing the system using the object paradigm, where large set of notations are used. The diagrams are classified as static and dynamic.

To design the system, the following static diagrams are used.

- Class diagrams
- Module diagrams
- Object diagrams
- Process diagrams

The dynamic diagrams are

- State transition diagrams
- Interaction diagrams

**Diagrams**

***Class diagrams***

Class diagram shows the existence of classes and their relationships. In the analysis phase they indicate the roles and responsibilities of the entities in the system and during the design phase the structure of the classes are shown.

***Notation:***

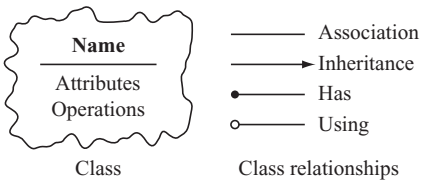



**Fig. 1**

***Object diagrams***

This is used to show the existence of objects and their relationships in the logical design of a system. The two major elements of an object diagram are objects and their relationships.

***Notation:***

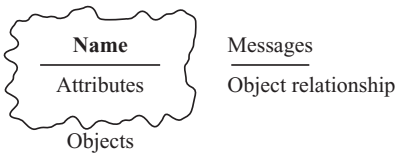



**Fig. 2**

***Example*:** Object diagram for Library Information System.

***Sample classes***

Member, book, fine amount etc.

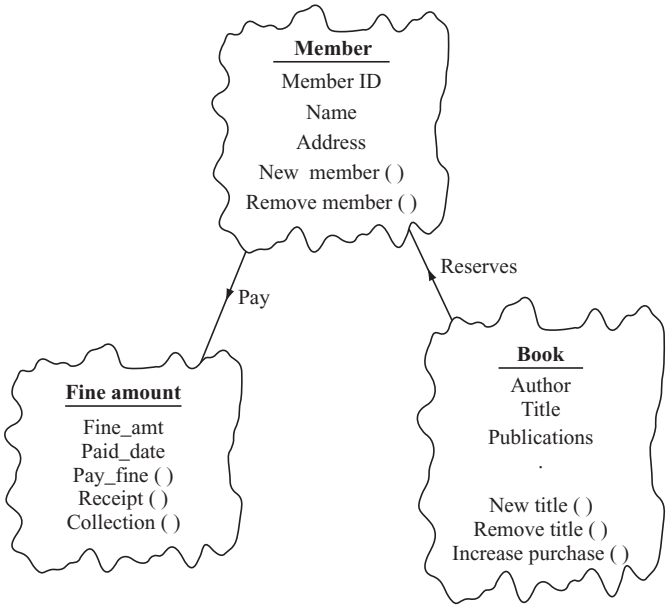



**Fig. 3**

**Modulate diagrams (component)**

A module diagram shows the allocation of classes and objects to modules in the physical design of a system. The two major elements are modules and their dependencies.

**Notation:**

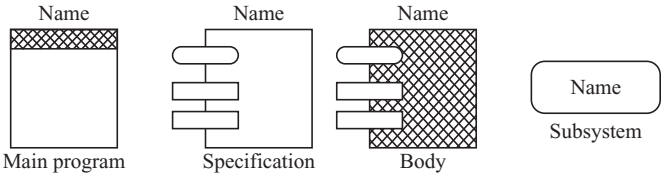



**Fig. 4** Modules and subsystems

***Example***

Library Information System (LIS)

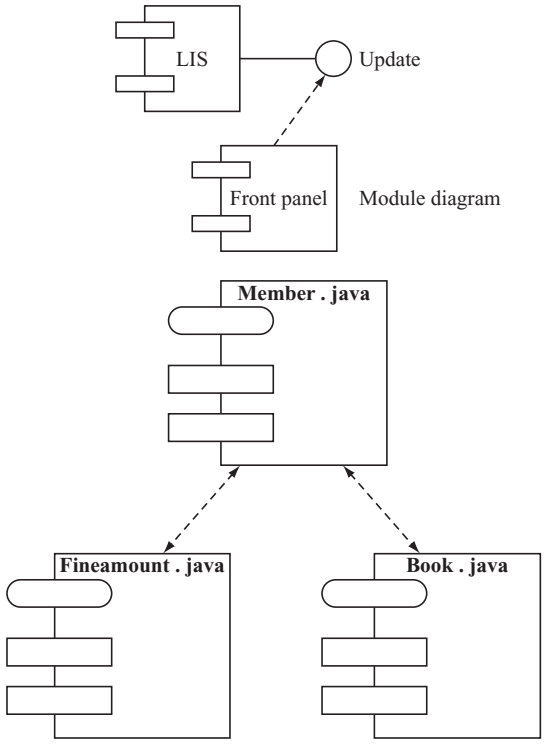



**Fig. 5**

***Process diagrams (Development diagrams)*:** A process diagram is used to show the allocation of processes to processors in the physical design of a system. The major elements of a process diagram are processors, devices and their connections.

***Notation:***

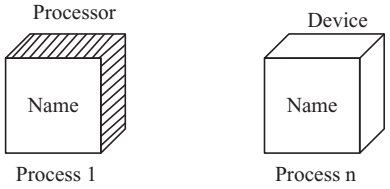



**Fig. 6**

***Example*:** Process diagram of LIS

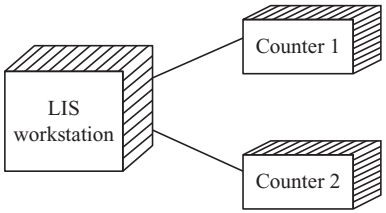



**Fig. 7**

***State transition diagrams*:** This diagram is used to show the state space of a given class, the events that makes a transition from one state to another and the actions that take place for a state change. A single state transition diagram represents a view of the dynamic model of a single class or of the entire system. The major elements of this diagram are the states and state transitions.

***Notation:***

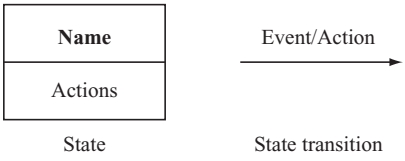



**Fig. 8**

***Example***

State transition diagram for LIS. (To reserve or remove reservation of book).

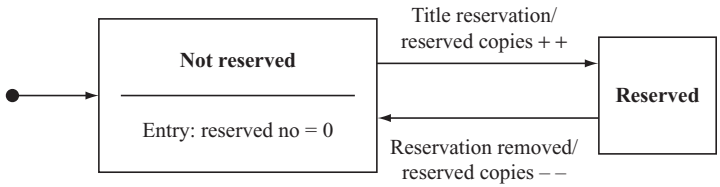



**Fig. 9**

***Interaction diagrams***

An interaction diagram is used to watch the execution of a scenario in the same context as an object diagram. The two major elements are objects and interactions.

***Example*:** Interaction diagram for LIS (Lend book).

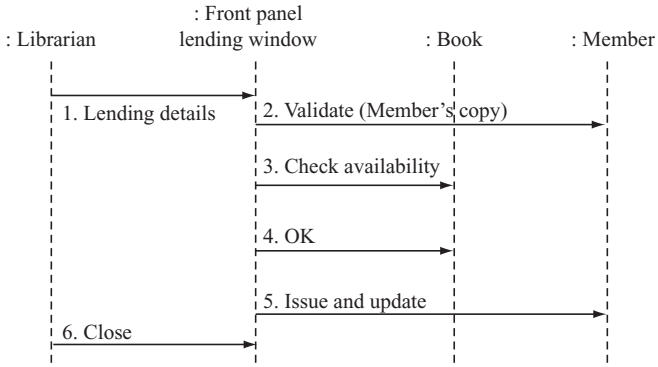



**Fig. 10**

With the help of these diagrams the incremental and iterative approach to project development is achieved. And, the Booch methodology focus on two types of process namely, the macro development and micro development process.

12. (a) (ii) A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development. Framework provides architectural guidance by portioning the design into abstract classes and defining their responsibilities and collaborations.

Major differences between design patterns and frameworks as follows:

i) ***Design patterns are more abstract than frameworks:*** Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied by executed and reused directly. In contrast, design patterns have to be implemented each time than they used. Design patterns also explain the intent, trade offs and consequences of a design.

ii) ***Design patterns are smaller architectural elements than frameworks*****:** A typical frameworks contains several design patterns but the reverse is never true.

iii) ***Design patterns are less specialized than frameworks:*** Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even though would not dictate an application architecture.

12. (b) (i) The main focus of this methodology relies on building a model of an application domain and adding the implementation details during the design of a system. Also, Rambaugh's methodology is termed as object modeling technique (OMT). The various stages of OMT are:

***Analysis*****:** Analysis model does not contain any type of implementation details rather the analysts constructs a model of the real-world along with its known properties by getting the appropriate requirements. The results of this stage are modeling objects, dynamic and functional models.

***System design*****:** The overall architecture of the system is designed by including the various subsystems.

***Object design*****:** The system is designed by including the implementation details.

***Implementation*****:** The classes which are identified in the earlier stages are transformed into a programming language, database etc to a system. During this process the software engineering metrics are followed.

Rambaugh's OMT is generalized into three parts:

***Dynamic model*:** Dynamic model describes the interactions among objects in the system (ie., the aspects which change over time). Rambaugh's dynamic model contains state diagrams. State diagram is a graph whose nodes are states and arcs are transitions between states caused by events.

***Example*:** One-shot state diagram for chess game.

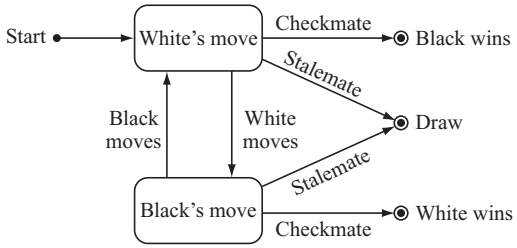



**Fig. 11**

**Object model:** Object model describes the static structure of the objects and their relationships. It also gives an outline of the dynamic and functional models. This model is represented graphically using the object diagram.

***Object diagram*:** It is a graphical notation whose nodes are object classes and arcs are relationships among classes.

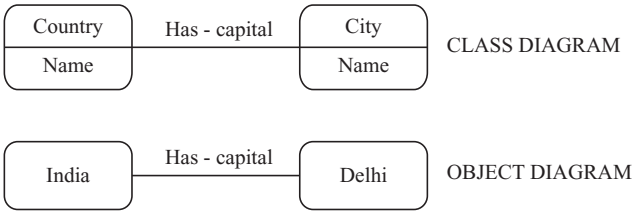



**Fig. 12**

Object diagrams are useful for abstract modeling which helps to design actual programs.

Two types of object diagram are: Class and Instance diagrams.

1) ***Class diagrams*:** It is a schema, pattern or template to describe many possible instances of data ie., the object classes.
2) ***Instance diagrams*:** It describes the relationships between objects ie., the object instances.

- The *relationship is* established through links and associations where, links are a physical or conceptual connection between object instances.

  ***Example*:** John *plays for* Indian football team.
- *Association* is a group of links with common structure and common semantics.

***Example:***

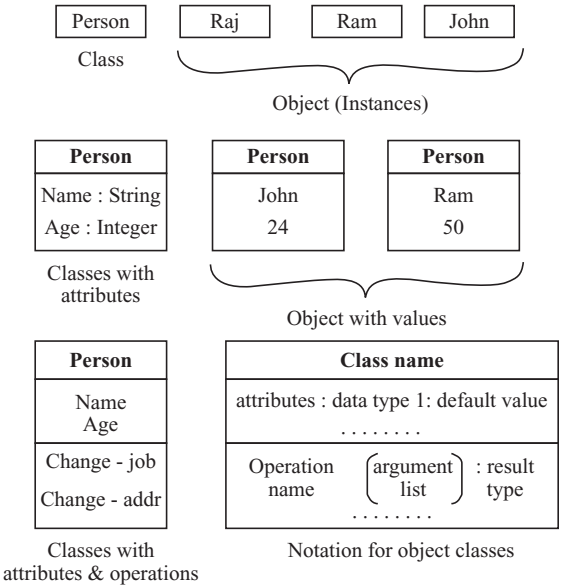



**Fig. 13**

***Example*:** A person plays for a team.

Also, the association describes a set of hidden links in the same way as the class describes a set of potential object.

***Example:***

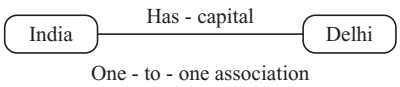



**Fig. 14**

Association is represented using lines between objects.

- Multiplicity specifies the number of instances a class may relate to a single instance of an associated class.

**Functional model:** It describes the transformations of the system ie., the flow of data between processes in a business. The model is explained using data flow diagrams.
DFD uses four primary symbols, they are:

- ***Process*:** Function being performed.
  ***Symbol*:** 
- ***Data flow*:** Shows the direction of data element's movement.
  ***Symbol*:** →
- ***Data store*:** Area of data storage.
  ***Symbol*:**  *or* 
- ***External entity*:** Source or destination of a data element.
  Symbol: 
  ***Example*:** DFD for a withdraw function:

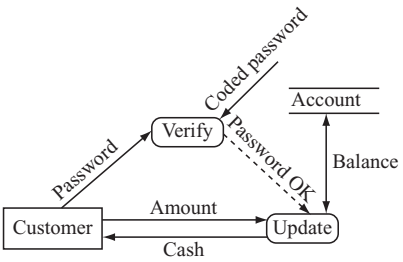



**Fig. 15**

***Process*:** Verify and update.
***Data flow*:** Password, coded password, amount, cash etc.
***Data store*:** Account
***External entity*:** Customer

12. (b) (ii) ***Activity diagrams*:** Represents the behaviour of an operation as a set of actions. It is a flowchart showing the flow of control from activity to activity. An activity diagram is a variant of a state diagram in which most of the states are activity states.
***Terms*:** Uses most of the notations from statechart diagram and few exceptional are discussed below.

1) ***Branch*****:** A branch holds a single incoming transition and several outgoing transitions.
2) ***Merge*****:** A merge has multiple input transitions and a single output.

   Activity diagrams can be used while
   1. Understanding a use case
   2. Understanding the workflow
   3. Describing a complicated algorithm
   4. Dealing with multithreaded applications

   Activity diagrams commonly contain activity states, action states, transitions.
3) ***Swimlanes*****:** A kind of package.

   ***Example*****:** Activity diagrams for the account class - withdrawal method.

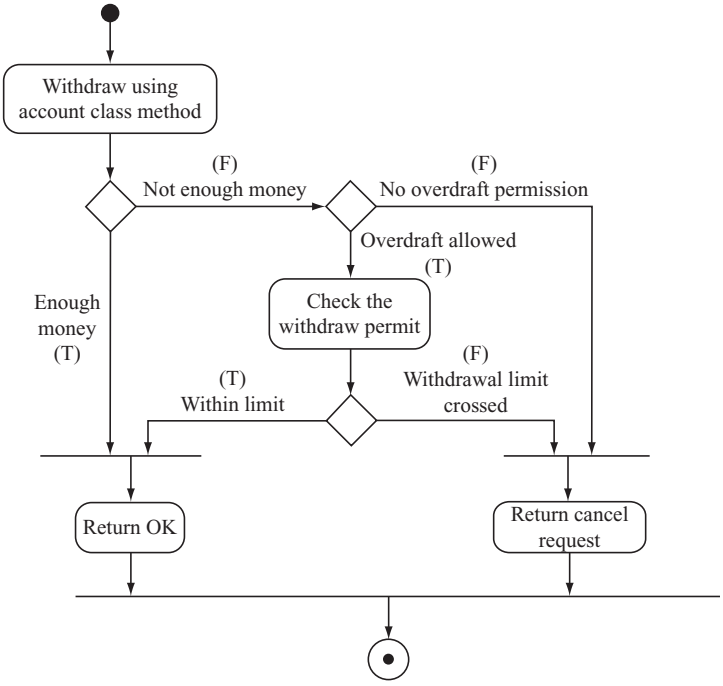



**Fig. 16**

13. (a) (i) ***Definition given by Jacobson*****:** A use case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system.

***Steps for finding use cases***

- Find the responsibilities and functions that the actor should be able to perform or that the system needs the actor to perform.
- Name the use cases.
- Describe the use cases briefly by applying terms with which the user is familiar.

Every use case has one main actor to initiate them. If a use case is not initiated by any of the actors then the use case is called as abstract use case.

***Naming a use case***

- The use case name should be a general description of the use-case function, ie., it should express what happens when instance of the use case is performed.
- It should be descriptive and reliable.

***Example:***

**ATM System:** Use case to describe the depositing money into an ATM machine could be named as receive money or deposit money.

**Soda machine:** Use case to describe the operation of filling the stock in the soda machine can be named as restock.

13. (a) (ii) The main phrase approach was proposed by Rebecca Wifrs Brock, Brian Wilkerson and Lauren Wiener. In this method, you read through the requirements or use 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 3 categories (Fig. 4) relevant classes, fuzzy classes and irrelevant classes.


Fig.4 using the noun phrase strategy, candidate classes can be divided into 3 categories: Relevant classes, Fuzzy area or Fuzzy classes 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 2 categories. 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 needed to be added or removed from the model and remember, flexibility is a virtue. You must able to formulate a statement of purpose for each candidate class; if not, simply eliminte it.

***Identifying tentative classes*:** The following are guidelines for selecting classes in an application.

- Look for nouns and noun phrases in the usecases.
- 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.

Finding classes is not easy. The more practice you have, the better you get at identifying classes. Finding classes is an incremental and interactive process. Booch explains this point elegantly: "Intelligent classification is intellectually hard work, and it best comes about through an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as GUIs, database standards, and even 4GC".

***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.

- ***Redundant classes*:** Don't keep 2 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.
- ***Adjective 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.

- ***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 demographic of client are not classes but attributes of the client class.
- ***Irrelevant classes*:** Each class must have a purpose and every class should be clearly defined and necessary. You must formulated statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

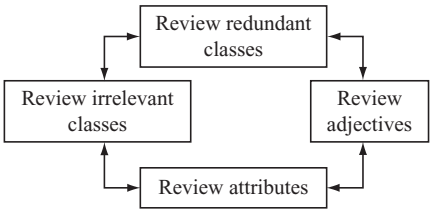



**Fig. 6** The process of eliminating the redundant classes and refining the remining classes is not sequential

This is an incremental process. Some classes will be missing, others will be eliminated or refined later. Unless you are starting with a log of domain knowledge, you probably are missing more classes then you will eliminate. Although some classes ultimately may become superclasses, at this stage simply identify them as individual, specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it. Classification is the essence of good object-oriented analysis and design. This process (of eliminating redundant classes, classes containing adjectives, possible attributes and irrelevant classes) is not sequential. You can move back and forth among these steps as often as you link (Fig. 6).

13. (b) (i) **Defining Attributes**

Attributes make it possible to assign user-defined build criteria to text, topics, documents, and styles — which makes it possible to single source one project several different ways (for example, you could create both an Administrator and Manager version of a manual and/or help system from the same project).

Once you have created an attribute, it can be chosen when you mark text as Conditional Text. Build attributes can also be

assigned to specific Character Styles, Topics, and even entire Documents. Then specify the attributes to include in the build of each Target — and when that Target is built the text, topic, or document flagged with those attributes will appear only in the appropriate output.

Doc-To-Help has two default Build attributes built in: **Internal** and **Release**.

To open the Attributes dialog box

1. Open the **Project** tab.
2. From the **Project** ribbon group, click the **Attributes** button. The **Attributes** dialog box will open.

To create a new attribute

In the **Attributes** dialog box, click the **Add New Attribute** button. An Attribute named **NewAttribute** will appear in the Attributes list. The **NewAttribute** name will initially be editable; if you wish to change it later, select it, then click the **Edit** (pencil) icon.

***Example*:** An example of an attribute name would be "Audience." The "Audience" attribute would then need values.

To add a new value to an attribute

In the **Attributes** dialog box, click the **Add New Value** button. A Value named **Value** will appear in the list. The **Value** name will initially be editable; if you wish to change it later, select it, then click the **Edit** (pencil) icon.

***Example*:** An example of attribute values for the "Audience" attribute would be "Pharmacist" and "Nurse."

These types of relationships allow objects to be built from other objects. It is the relationships between classes, where one class is a parent of another class. Reusability of objects is the major advantage of these relationships.

***Guidelines*:** Top-down-look for noun phrases composed of various adjectives without much refinement.

***Example*:** Phone operator
Father
Clerk
Business man

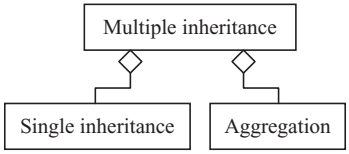



**Fig. 1** Reletionship of inheritance and aggregation

These three objects have similar properties.

- Bottom-up – Look for classes with similar attributes or methods.
- Reusability – Move attributes and methods as high as possible.
- Multiple inheritance – This may mostly lead to complications and hence avoid the excessive use of multiple inheritance.

13. (b) (ii) Cunningham and Beck were concerned about teaching of the basic concepts of object-oriented development and presented the simple technique of Class-Responsibility-Collaboration (CRC) cards. This is a technique used for identifying classes, responsibilities and their attributes and methods. Classes can accomplish a certain responsibility and they are represented on 4" × 6" index cards.

Responsibility is a high-level description of the purpose of the class. One major benefit of CRC cards is that they encourage animated description among the developers. With CRC cards, we can model the interactions by picking up the cards and moving around. CRC cards helps to explore an interaction between classes, typically to show how a scenario is implemented.

| ***Class Name*** | ***Collaboration*** |
|---|---|
| Responsibilities | ... |

(a) CRC index card

| ***Order*** | ***Collaboration*** |
|---|---|
| * Check the items in stack<br>* Identity price | Order line |
| * Check payment | Customer |
| * Deliver item | |

(b) Example-Purchase of Item

**Fig. 17**

***CRC process*:** CRC process consists of three steps:

1) Identify classes' responsibilities (and identify classes).
2) Assign responsibilities.
3) Identify collaborators.

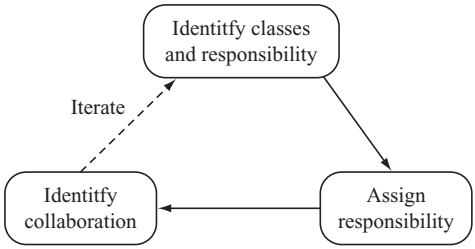



**Fig. 18**

***Naming classes:***

Guidelines

1) The class name should be singular.
2) Use names with which the users or clients are comfortable.
3) The name of a class should reflect its intrinsic nature.
4) Use readable names. Capitalize class names.

14. (a) (i) Refer Apr/May 2010 - 14(a).

14. (a) (ii) Refer Apr/May 2010 - 14(a).

14. (b) (i) Refer Nov/Dec 2009 - 14(b).

14. (b) (ii) Client–server computing is a distributed computing model in which client applications request services from server processes. Clients and servers typically run on different computers interconnected by a computer network. Any use of the Internet (q.v.), such as information retrieval (q.v.) from the World Wide Web (q.v.), is an example of client–server computing. However, the term is generally applied to systems in which an organization runs programs with multiple components distributed among computers in a network. The concept is frequently associated with enterprise computing, which makes the computing resources of an organization available to every part of its operation.

A client application is a process or program that sends messages to a server via the network. Those messages request the server to perform a specific task, such as looking up a customer record in a database or returning a portion of a file on the server's hard disk. The client manages local resources such as a display, keyboard, local disks, and other peripherals.

The server process or program listens for client requests that are transmitted via the network. Servers receive those requests and perform actions such as database queries and reading files. Server processes typically run on powerful PCs, workstations (q.v.), or mainframe (q.v.) computers.

An example of a client–server system is a banking application that allows a clerk to access account information on a central database server. All access is done via a PC client that provides a graphical user interface (GUI). An account number can be entered into the GUI along with how much money is to be withdrawn or deposited, respectively. The PC client validates the data provided by the clerk, transmits the data to the database server, and displays the results that are returned by the server.

The client–server model is an extension of the object based (or modular) programming model, where large pieces of software are structured into smaller components that have well defined interfaces. This decentralized approach helps to make complex programs maintainable and extensible. Components interact by exchanging messages or by Remote Procedure Calling. The calling component becomes the client and the called component the server.

A client–server environment may use a variety of operating systems and hardware from multiple vendors; standard network protocols like TCP/IP provide compatibility. Vendor independence and freedom of choice are further advantages of the model. Inexpensive PC equipment can be interconnected with mainframe servers, for example.

Client–server systems can be scaled up in size more readily than centralized solutions since server functions can be distributed across more and more server computers as the number of clients increases. Server processes can thus run in parallel, each process serving its own set of clients. However, when there are multiple servers that update information, there must be some coordination mechanism to avoid inconsistencies.

The drawbacks of the client–server model are that security is more difficult to ensure in a distributed environment than it is in a centralized one, that the administration of distributed equipment can be much more expensive than the maintenance of a centralized system, that data distributed across servers needs to be kept consistent, and that the failure of one server can render a large client–server system unavailable. If a server fails, none of its clients can make further progress, unless the system is designed to be fault-tolerant.

The computer network can also become a performance or reliability bottleneck: if the network fails, all servers become unreachable. If one client produces high network traffic then all clients may suffer from long response times.

15. (a) (i) A comprehensive definition of Software Quality Assurance SQA is presented here, in the form of a short history of the Software Quality Movement, an example of a theoretical CMMi SQA\SQC implementation and an example of what a typical Software Application QA engineer does day to day.

The definition still refers back to the traditional manufacturing QA world. There are, however, **some notable differences between software and a manufactured product**.

These differences all stem from the fact that the manufactured product is physical and can be seen whereas the software product is not visible. Therefore its function, benefit and costs are not as easily measured.

The following differences highlight some of the **issues in taking the manufacturing QA\QC model and applying it to software development.**

- The manufactured product is a physical realization of the customer requirements.
- The function of the product can be verified against this physical realization.
- The costs of manufacture, including rework, repairs, recalls etc., are readily categorized and visible.
- The benefit of the product to its user\customer are readily categorized and visible.

In order to overcome these types of issues, and reap the benefit of QA\QC applied to software, other terms, models and paradigms needed to be (and were) developed.

In order to identify the Software Costs and Benefits, remembering Fujitsu's term **with cost and performance as prime consideration**, a number of **Software Characteristics** where defined. These characteristics are sometimes referred to as Quality Attributes, Software Metrics or Functional and Non-Functional Requirements.

The intention here is to **breakdown the Software product into attributes that can be measured (in terms of cost benefit).**

Examples of these attributes are Supportability, Adaptability, Usability and Functionality.

There are many definitions of these Software Quality Attributes but a common one is the FURPS + model which was developed by Robert Grady at Hewlett Packard.

Under the FURPS, the following characteristics are identified:-

**Functionality**

The F in the FURPS+ acronym represents all the system-wide functional requirements that we would expect to see described.

These usually represent the main product features that are familiar within the business domain of the solution being developed.

For example, order processing is very natural for someone to describe if you are developing an order processing system.

The functional requirements can also be very technically oriented. Functional requirements that you may consider to be also architecturally significant system-wide functional requirements may include auditing, licensing, localization, mail, online help, printing, reporting, security, system management, or workflow.

Each of these may represent functionality of the system being developed and they are each a system-wide functional requirement.

**Usability**

Usability includes looking at, capturing, and stating requirements based around user interface issues, things such as accessibility, interface aesthetics, and consistency within the user interface.

**Reliability**

Reliability includes aspects such as availability, accuracy, and recoverability, for example, computations, or recoverability of the system from shut-down failure.

**Performance**

Performance involves things such as throughout of information through the system, system response time (which also relates to usability), recovery time, and startup time.

**Supportability**

Finally, we tend to include a section called Supportability, where we specify a number of other requirements such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, localizability, and so on.

15. (a) (ii) The following is the process of designing view (interface) objects:

1. For every interface object identified in the macro UI design process (fig.21), apply *micro-level* UI *design rules and corollaries to develop the* UI. Apply the design rules and GUI guidelines to design the UI for the interface objects identified.
2. Iterate and refine.

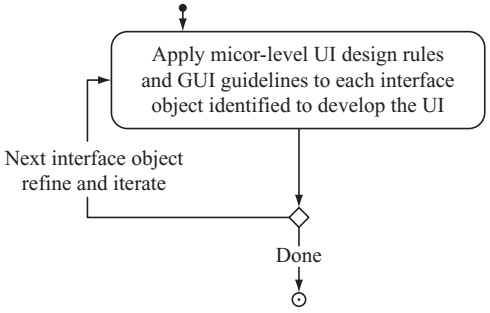



**Fig. 21** The micro-level design process

***UI design rule1*:** Making the Interface simple (Application of corollary 2)

User interface should be so simple that users are unaware of the tools and mechanisms that make the application work.

As applications become more complicated, users must have an even simpler interface, so they can learn new applications more easily. Today's car engines are so complex that they have on board computers and sophisticated electronics the driver interface remains simple. The driver needs only a steering wheel and the gas and brake pedals to operate a car. Drivers don't have to understand what is under the hood or even aware of it to drive a car, because the driver interface remains simple. The UI should provide the same simplicity for users.

This rule is an application of single purpose in UI design. Here it means that each UI class must have a single, clearly defined purpose. When you document, you should be able easily to describe the purpose of the UI class with a few sentences. A number of additional factor may affect the design of your application. For example, deadlines may require you to deliver a product or market with a minimal design process, or comparative evaluations may force you to consider additional features. There is no simple equation to determine. When a design trade-off is appropriate. So, in evaluating the impact, consider the following:

- Every additional feature potentially affects the performance, complaints stabilities, maintenance and support costs of an application.
- It is harder to fix a design problem after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.
- Simplicity is different from being simplistic. Making something simple to use often require a good deal of work and code.
- Features implemented by a small extension in the application code do not necessarily have a proportional effect in a UI. For example, if the primary task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carryout.

***UI design Rule 2*:** Making the Interface Transparent & natural the UI should be so interactive and natural that users can anticipate what to do next by applying their previous knowledge of doing tasks without a computer. An application, should reflect a real-world model, of the users' goals and the tasks necessary to reach those goals.

The 2nd UI rule is an application of strong mapping in UI design. This corollary implies that there should be strong mapping between the user's view of doing things and UI classes. A *metaphor* relates 2 otherwise unrelated things by using one to denote the other. For example, writers use metaphors to help readers understand a conceptual image or model of the subject. This principle also applies to UI design. It is a way to develop the users' conceptual model of an application. Familiar metaphors can assist the users to transfer their previous knowledge from their work environment to the application interface and create a strong mapping between the user's view and the UI objects.

***UI design rule 3 - Allowing users to be in control of the software*:** The third UI design rule states that the users always should feel in control of the software, rather than feeling controlled by the software. This concept has a number of implications. The first implication is the operational assumption that actions are started by the user rather than the computer or software, that the uses plays an active rather than reactive role. The 2nd implication is that users, because of their widely varying skills and preferences, must be able to customize aspects of the interface. The system software provides user access to many of these aspects. The software should reflect user settings for different system properties such as color, fonts or other options. The final implication is that the software should be as interactive and responsive as possible. A mode is a state that excludes general interaction or otherwise limits the user to specific interactions. There are situations in which modes are useful. For example, selecting a file name before opening it. The dialog that gets me the file name must be model. The main idea here is to avoid creating a single UI class for several business objects, since it makes the UI less flexible and forces the user to perform tasks in a monolithic way.

***Make the interface forgiving*:** The users' actions should be easily reversed. Users like to explore an interface and often learn by trial and error. They should be able to back up or undo previous actions. Actions that are destructive and may cause the unexpected loss of data should require a confirmation or better, should be reversible or recoverable. Users feel more comfortable with a system when their mistakes do not cause serious or unreversible results.

***Make the interface visual*:** Design the interface so users can see, rather than recall, how to proceed. Whenever possible,

provide users a list of items from which to choose, instead of making them remember valid choices.

***Provide immediate feedback*:** Users should never press a key or select an active without receiving immediate visual or audible feedback or both. When the cursor is an as choice, for example, the color, emphasis and selection indicators show users they can select that choice. After users select a choice, the color, emphasis and selection indicators change to show users their choice is selected.

***Avoid modes*:** Users are in a mode whenever they must cancel what they are doing before they can do something else or when the same action has different results in different situations. These are some of the modes that can be used in the user interface.

***Make the interface consistent*:** Consistency is one way to develop and reinforce the user's conceptual model of appreciating and give the user the feeling that he/she is in control, since the user can predict the behavior of the system.

***Example:***

***Guidelines for designing forms and data entry windows*:** When designing a data entry window or forms (or webforms) identify the information you want to display or change. Consider the following issues:

- In general, what kind of information will users work with and why? For example, a user might want to change inventory information, entry orders, or maintain prices for stock items.
- Do users need access to all the information in a table or just some information? When working with a portion the information in a table, use a query that selects the rows and columns users want.
- In what order to users want rows to appear? For example, users might want to change inventory information stored alphabetically, chronologically, or by inventory number. You have to provide a mechanism for the user to that the order can be modified.

Identify the tasks that users need to work with data on the form or data entry window. Typical data entry tasks include the following:

- Navigating rows in a table, such as moving forward and backward, and going to the first and lost record.

- Adding and deleting rows.
- Changing data in rows.
- Saving and abandoning changes.
- You can provide menus, push buttons and speed bar buttons that users choose to initiate tasks. You can put controls anywhere on a window. The layout you choose determines how successfully users can enter data using the form. Here are some guidelines to consider.
- If the printed form contains too much information to fit on a screen, consider using a main window with optional smaller windows that users can display on demand or using a window with multiple page (fig.22). Users typically are more productive when a screen is not clustered.

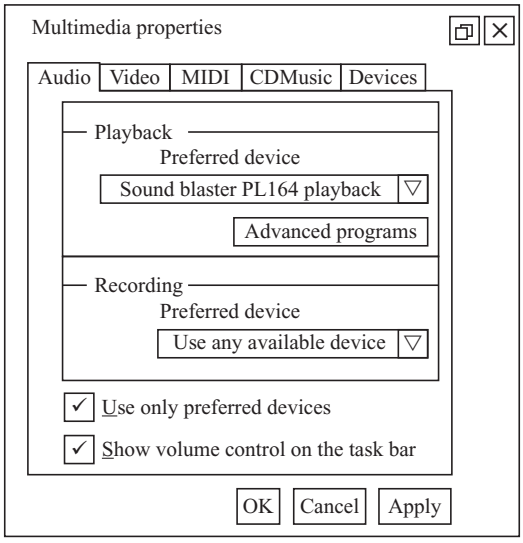



**Fig. 22** An example of a dialog box with multiple pages in the microsoft multimedia setup

- Users scan a screen in the same way they read a page of a book, from left to right and top to bottom. In general, put required or frequently entered information toward the top and left side of the form, entering optional or selector-entered information toward the bottom and right side. For example, on a window for enter inventory data, the inventory number and item name might best be placed in the upper-left corner, while the signature could appear lower and to the right (fig.23).

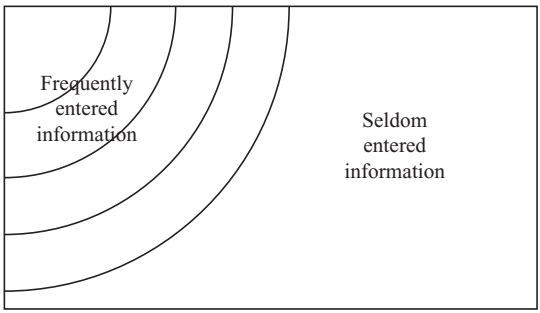



**Fig. 23** Required information should be put forward the top and left side of the form, entering optional or seldom entered information toward the button

- When information is positioned vertically, align fields at their left edges. This usually makes it easier for the user to scan the information text labels usually are left aligned and placed above or to the left of the areas to which they apply when placing text labels to the left of text box controls, align the height of the text with text displayed in the text box (fig.24)

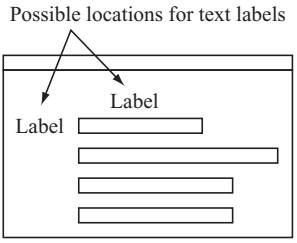



**Fig. 24** Place text labels to the left of the text box controls, align the height of the text with text displayed in the text box

- When entering data, users expect to type information from left to right and top to bottom, as if they were using a type writer (usually the Tab key moves the focus from one control to another). Arrange controls in the sequence users expect to enter data. You may want the users to be able to jump from one group of controls to the beginning of another group, skipping over individual controls. For example, when entering address information, users expect to enter the Address, city, state and Zipcode (fig.).

- Put similar or related information together, and use visual effects to emphasize the grouping. For example, you might want to put a company's billing and shipping address information in separate groups. To emphasize a group, you can enclose its controls in a distinct visual area using a rectangle, lines, alignment or color (fig.). The real-world issues on Agenda "*Future of the GUI Landscape*" examines window presentations for the future.

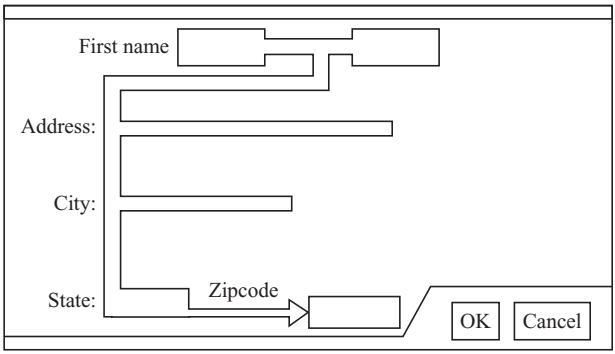



**Fig. 25** Arrange controls left to right and top to bottom

15. (b) (i) ***Test plan*****:** A test plan is developed to detect and identify potential problems before delivering the software to its users. Your users might demand a test plan as one item delivered with the program. In other cases, no test plan is required, but that does not mean you should not have one. A test plan offers a road map for testing activities, whether usability, user satisfaction or QA tests. It should state the test objectives and how to meet them.

The following steps are needed to create a test plan:

1. ***Objective of the test*****:** Create the objectives and describe how to achieve them. For example, if the objective is usability of the system, that must be stated and also how to realize it.
2. ***Development of a test case*****:** Develop test data, both input and expected output based on the domain of the data and the expected behaviors that must be tested.
3. ***Test analysis*****:** This step involves the examination of the test output and the documentation of the test results. If bugs are detected, then this is reported and the activity centres on debugging. After debugging, steps 1 through 3 must be repeated with no bugs can be detected.

All passed tests should be repeated with the revised programs called *regression testing,* which can discover errors introduced during the debugging process. When sufficient testing is behaved to have been conducted, this fact should be reported, and testing for this specific product is complete.

***Guidelines for developing test plans*:** As software gains complexity and interaction among program is more tightly coupled, planning becomes essential. A good test plan not only prevents overlooking a feature, it also helps divide the work load among other people, explains Thomas.

Thomas developed the following guidelines for writing test plans:

- You may have requirements that dictate a specific appearance or format for your test plan. These requirements may be generated by the users. Whatever the appearance of your test plan, try to include as much detail as possible about the tests.
- The test plan should contain a schedule and a list of required resource. List how many people will be needed, when the testing will be done, and what equipment will be required.
- After you are determined what types of testing are necessary, you need to document specifically what you are going to do. Document every type of test you plan to complete. The level of detail in your plan may be driven by several factors, such as the following. How much test time do you have? Will you use the test plan as a training tool for new team members?
- A *configuration control system* provides a way of tracking the changes to the code. At a minimum, every time the code changes, a record should be kept that tracks which module has been changed, who changed it, and when it was altered with a comment about why the change was made. Without configuration control, you may have difficulty keeping the testing in line with the changes, since frequent changes may occur without being communicated to the testers.
- A well-though-out design tends to produce better code and result in more complete testing, so it is a good idea to try to keep the plan up to data. If the test plan is so old that it has become part of the fossil record, it is not terribly useful. AS you approach the end of a project, you will have less time to create plans. If you don't take the time to document the

work that needs to be done, you risk forgetting something in the mad dash to the finish line. Try to develop a habit of routinely bringing the test plan in sync with the product or product specification.

- At the end of each month or as you reach each milestone, take time to complete routine updates. This will help you avoid being overwhelmed by being so out-of-data that you need to rewrite the whole plan. Keep configuration information on your plan too. Notes about who made which updates and when can be very helpful down the road.

15. (b) (ii)
- The ISO defines usability as the effectiveness, efficiency, and satisfaction with which a specified set of users can achieve a specified set of tasks in particular environments.
- The ISO definition requires.
  - Defining tasks
  - Defining users
  - A means for measuring effectiveness.
- Usability testing measures the case of use as well as the degree of comfort and satisfaction users have with the software.
- It is one of the most crucial factors in the design and development of a product, especially the user interface.
- It should begin in the early stages of product development.
- It begins with the identification of use cases that can specify the target audience, tasks, and test goals.
- Use cases and usage scenarios can become the test scenarios.

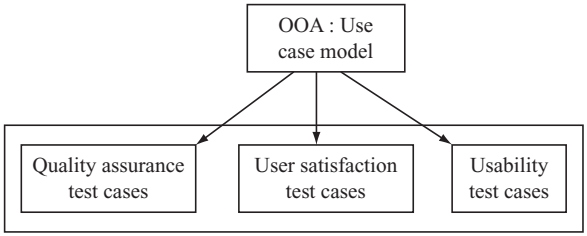



**Fig. 26** The use cases identified using analysis can be used in testing the design

***Guidelines for developing usability testing***

- Many techniques can be used to gather usability information. Focus groups can be helpful for generating initial ideal or trying out new ideas.
- Usability tests can be conducted in an one-on-one fashion.
- The usability testing should include all of software's components.
- Usability testing need not be very expensive or elaborate.
- All tests need not involve many subjects.
- Consider the user's experience as part of your software usability.
- Apply usability testing early and ofter.

***Recording the usability test***

- Provide an environment where the target audience can comfortably work with all the situations and record them.
- Do not intervene the users during testing process and make them to encounter all possible situations.
- As participants work, record the time they take to perform a task as well as any problems they encounter.
- Follow-up the session with the user satisfaction test and a question name that asks the participants to evaluate the product or tasks they performed.
- Record the test results using a portable tape recorder or a video camera.
- Recorded data allows the design team to review and evaluate the results.
- To ensure user satisfaction, measure user satisfaction along the way as the design takes form.

# B.E/B.Tech. DEGREE EXAMINATION, APRIL/MAY 2008

**Sixth Semester**

## Computer Science and Engineering

### CS1402 – OBJECT ORIENTED ANALYSIS AND DESIGN

## (Common to Information Technology)

**Time: Three Hours** **Maximum: 100 Marks**

**Answer ALL questions.**

**PART A – (10 × 2 = 20 marks)**

1. What is the heart of the unified approach?
2. Create the class hierarchy to organize the following vehicle classes land, water, air, car, cycle, rowboat, ship, aeroplane, gas balloon
3. List the OMT models and their purpose.
4. How are use cases used in every phases of software development?
5. What is a meta model? Is it important to understand a meta model?
6. Define the types of associations between use cases.
7. Differentiate between public and private protocols.
8. List the ways to categorize data based on their lifetime.
9. What is user satisfaction testing?
10. What is a test plan? What are its components?

**PART B – (5 × 16 = 80 marks)**

11. (a) Explain in detail the transformations in Object-oriented software development life cycle giving suitable examples. (16)

Or

(b) (i) What are the orthogonal views of software? (4)

(ii) List the advantages offered by object-oriented system development. (4)

(iii) Describe the components of the unified approach with a neat diagram. (8)

12. (a) (i) What is a pattern? What are the characteristics of a good pattern? (4)

(ii) List the components that decide the template of a pattern (4)

(iii) Draw and explain the following diagrams for Automatic Teller Machine Operations

1. Use case diagram. (4)
2. Activity Diagram. (4)

Or

(b) (i) What is a framework? How does it differ from patterns? (4)

(ii) Describe the different types of relationships among classes with examples (6)

(iii) Draw and explain the following diagrams for a cell phone scenario

1. sequence diagram (3)
2. state diagram (3)

13. (a) (i) What is classification? List the approach to identify classes. (6)

(ii) Explain the noun phrase approach to identify classes with an example. (10)

Or

(b) Write the steps to identify the classes and their relationships. Consider the example of an ATM system. (16)

14. (a) Explain in detail the steps involved in the creation of access layer classes with examples. (16)

Or

(b) (i) Explain in detail the object oriented design process with a suitable diagram. (8)

(ii) Explain in detail the different object oriented design axioms and corollaries. (8)

15. (a) (i) Explain the techniques followed in designing interface objects. (8)

(ii) Discuss the major software quality assurance schemes followed in system development. (8)

Or

(b) (i) What are the different testing strategies? Explain. (9)

(ii) What is a test case? Explain it with an example. (7)

**Solution available at www.pearsoned.co.in/questionbanks**

# B.E./B.Tech. DEGREE EXAMINATION
# NOV/DEC 2008

## Sixth Semester

## Computer Science and Engineering

## CS 1402 - OBJECT ORIENTED ANALYSIS AND DESIGN

## (Common to Information Technology)

**Time: Three hours** **Maximum: 100 Marks**

**Answer ALL questions**

**PART A (10 · 2 = 20 Marks)**

1. Define an object.
2. What is coupling?
3. What is the use of UML in object oriented approach?
4. State how a use case is represented.
5. How a class is represented in UML?
6. Define aggregation.
7. State how analysis differs from design?
8. When do you call software as successful software?
9. List any two aspects of software quality.
10. State any two functional requirements.

## PART B (5 · 16 = 80 Marks)

11. (a) (i) List the reasons to explain why objects are so powerful in programming.
(ii) Give a brief overview on development of object oriented systems.

Or

(b) (i) List the characteristic features of object oriented approach.
(ii) Give a brief discussion on object oriented system life cycle.

12. (a) (i) Discuss Booch's Object-oriented Analysis and Design methodology.
(ii) What is unified approach? Explain.

Or

(b) Summarize briefly the different UML diagram types that are focused on a different way to analyze and define the system.

13. (a) (i) Construct and explain object model for an automatic washing machine.
(ii) Describe the essential components of use-cases.

Or

(b) (i) Construct a use-case diagram for hotel reservation system.
(ii) Discuss how objects can be related in other ways than by inheritance and aggregation.

14. (a) Describe the following: (i) Access layer, (ii) Design axioms.

Or

(b) (i) Discuss in detail how object interoperability is achieved.
(ii) Write a note on object storage.

15. (a) (i) Describe the distinct aspects of system usability.
(ii) What are interface objects? Explain how to design them.

Or

(b) (i) Discuss how a software quality is assured to a user.
(ii) Write a brief overview on user satisfaction measurement.

**B.E./B.Tech. DEGREE EXAMINATION, NOV/DEC 2007**

**Sixth Semester**

**Computer Science and Engineering**

**CS 1402 – OBJECT ORIENTED ANALYSIS AND DESIGN**

## (Common to Information Technology)

**Time: Three hours** **Maximum: 100 marks**

**Answer All Questions**

**PART A – (10 × 2 = 20 marks)**

1. What is an instance? Give an example?
2. List the relationships among classes.
3. What are the phases of OMT?
4. Differentiate between Patterns and Frameworks.
5. Who are actors? How do they differ from users?
6. When to the CRC cards?
7. Define object storage.
8. List the object oriented design axioms and corollaries.
9. What are the main tools of Quality Assurance?
10. What is a test case?

**PART B – (5 × 16 = 80 marks)**

11. (a) (i) Discuss the advantages of Object Oriented Approach. (6)
(ii) Briefly explain the elements of Object model. (10)

Or

(b) (i) Briefly explain about Object oriented Systems development life cycle. (10)

(ii) Describe State, behaviour, and identity with respect to an object with relevant example. (6)

12. (a) (i) Compare and contrast the Object Oriented methodology of Booch, Rumbaugh and Jacobson. (6)

(ii) Explain about a Unified approach to software development. (10)

Or

(b) Draw the Class diagram and Usecase diagram, for Railway Reservation system. (8+8)

13. (a) Describe the basic activities in Object oriented analysis and explain how Usecase modeling is useful in analysis. (16)

Or

(b) Discuss the importance of proper classification. Briefly explain the different approaches used for identifying classes and objects. (16)

14. (a) Write short notes on the following:

(i) Object Interoperability (8)

(ii) Access Layer. (8)

Or

(b) (i) Briefly explain, how design axioms help to avoid design pitfalls. (8)

(ii) Explain the principles and metrics of good object oriented design. (8)

15. (a) Explain about software quality and usability. (16)

Or

(b) (i) Describe the different types of testing strategies

(ii) How do you develop a custom form for a user satisfaction test? Explain with an example. (8)