SRINIVASAN COLLEGE OF ARTS & SCIENCE

(Affiliated Bharathidasan University, Tiruchirappalli)

PERAMBALUR – 621 212

Department of Computer Science & Information Technology

COURSE MATERIAL

Subject	**:**	**OOAD & UML**
Subject Code	**:**	**P16CS21**
Class	**:**	**I M.Sc., COMPUTER SCIENCE**
Semester	**:**	**II**

OOAD AND UML

UNIT I
Structured approach to system construction: SSADM/SADT - An overview of object oriented systems development & Life cycle.

UNIT II
Various object oriented methodologies – Introduction to UML.

UNIT III
Object oriented analysis – Use cases- Object classification, relationships, attributes, methods.

UNIT IV
Object oriented design – Design axioms – Designing classes – Layering the Software design: - data access layer, User interface layer, Control/business Logic layer.

UNIT - V
UML - Examples on: Behavioral models – Structural models – Architectural Models from real world problems.

Text Books:

1. Bahrami Ali, Object oriented systems development, Irwin McGraw Hill, 2005 (First 4 units covered here).
2. Booch Grady, Rumbaugh James, Jacobson Ivar, The Unified modeling language – User Guide, Pearson education, 2006 (ISBN 81-7758-372-7) (UNIT -5 covered here).

***

UNIT-I

OOAD & UML

SSADM(Structured System Analysis and Design Method)

- The SSADM methodology is a well-defined (structured) methodology and is quite difficult to use.
- When used skillfully it can produce well-documented, accurate information systems.
- It recognizes the following stages in the systems development lifecycle.
- It concentrates on the analysis and design phase of the Waterfall Model of the Systems

The objectives of SSADM

- Improve the project management and control.
- Make more effective use of experienced and inexperienced development staff.
- Develop better quality system.
- Enable projects to be supported by computer based tools such as computer aided software engineering system.
- Establish a framework for good communication between participents in a project.

Stages in SSADM:

SSADM is composed of seven stages within a project's life cycle, and at the end of each stage the analyst and users can decide whether to move on to the next level, abandon the project, or revise one or more stages.

Stage 0: Feasibility

- The Feasibility stage is a short assessment of a proposed information system to determine if the system can meet the business requirements of

an organization, assuming the business case exists for developing the system.
- The analyst considers possible problems faced by the organization and produces various options to resolve these issues.

Stage 1: Investigation of the Current Environment

- Detailed requirements are collected and business models are built in the Investigation of the Current Environment stage.
- This stage is where you develop a business-activity model, investigate and define requirements, investigate current processing in the data flow model, investigate current data and derive the logical view of current services.

Stage 2: Business System Options

- The Business Systems Options, or BSO, stage allows the analyst and you to choose between a number of business-system options that each describe the scope and functionality provided by a particular development and implementation approach.
- After you present these to management, the management then decides which BSO is the better option.

Stage 3: Definition of Requirements

- This stage specifies the details in the processing and data requirements of the selected BSO option.
- In this stage you define the required system processing, develop the required data model, determine the systems for existing or new functions, develop the user job specifications, enhance the required data model, develop specific prototypes and confirm the system objectives.

Stage 4: Technical Systems Options

- This stage allows you and the analyst to consider the technical options.

- Details such as the terms of cost, performance and impact on the organization is determined.
- We identify, define and select the possible technical system option in this stage.

Stage 5: Logical Design

- This stage involves you specifying the new system through designing the menu structure and dialogues of the required system.
- The steps in this stage include defining the user dialogue, defining update processes and defining the inquiry processes.

Stage 6: Physical Design

- This is the implementation phase of SSADM.
- The Physical Design stage is used to specify the physical data and process design use the language and features of the chosen environment and incorporate installation standards.
- This stage concentrates on the environment in which the new system will be running.

SSADM uses a combination of three techniques:

- **Logical Data Modeling** -- the process of identifying, modeling and documenting the data requirements of the system being designed. The data is separated into entities (things about which a business needs to record information) and relationships (the associations between the entities.

- **Data Flow Modeling** -- the process of identifying, modeling and documenting how data moves around an information system. Data Flow Modeling examines processes (activities that transform data from one form to another),data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system, and data flows (routes by which data can flow).

- **Entity Behavior Modeling** -- the process of identifying, modeling and documenting the events that affect each entity and the sequence in which these events occur. Each of these three system models provides a different viewpoint of the same system, and each viewpoint is required to form a complete model of the system being designed. The three techniques are cross- referenced against each other to ensure the completeness and accuracy of the whole application.

  Software developers and program managers are used the SSADM in software development team.

  A software developer represents the software application using UML notations. Illustrates and interpret software application, relationships, actions and connections.

  A program manager shows high level static software structures in presentations and specification documentation.

Basic Concepts of Object Orientation

- Class
- Object
- Attributes
- Methods
- Inheritance
- Encapsulation and Information Hiding
- Polymorphism
- Message

Class

- The role of a class is to define the attributes and methods (the state and behavior) of its instances. The class car, for example, defines the property color.

- Each individual car (object) will have a value for this property, such as "maroon,""yellow" or "white."

Class Hierarchy

- An object-oriented system organizes classes into subclass-super hierarchy.
- At the top of the hierarchy are the most general classes and at the bottom are the most specific.
- A subclass inherits all of the properties and methods (procedures) defined in its super class.

Object

The term object was first formally utilized in the similar language to simulate some aspect of reality.

An object is an entity.

- It knows things (has attributes)
- It does things (provides services or has methods)
- Attributes or properties describe object‘s state (data) and methods define its behavior.
- In an object-oriented system, everything is an object: numbers, arrays, records, fields, files, forms, an invoice, etc.
- An Object is anything, real or abstract, about which we store data and those methods that manipulate the data.
- Conceptually, each object is responsible for itself.
- A window object is responsible for things like opening, sizing, and closing itself.
- A chart object is responsible for things like maintaining its data and labels, and even for drawing itself.

Attributes

- Attributes represented by data type.
- They describe objects states.
- In the Car example the car‘s attributes are:
  Color, manufacturer, cost, owner, model, etc.,

Methods

=

- Methods define objects behavior and specify the way in which an Object's data are manipulated.
- In the Car example the car's methods are: drive it, lock it, tow it, carry passenger in it.

Inheritance (programming by extension)

- Inheritance is a relationship between classes where one class is the parent class of another (derived) class.
- Inheritance allows classes to share and reuse behaviors and attributes.
- The real advantage of inheritance is that we can build upon what we already have and,
- Reuse what we alreadyhave.

Multiple Inheritances

- OO systems permit a class to inherit from more than one super class.
- This kind of inheritance is referred to as multiple inheritance.

Encapsulation and Information Hiding

- Information hiding is a principle of hiding internal data and procedures of an object
- By providing an interface to each object in such a way as to reveal as little as possible about its innerworkings.
- Encapsulation protects the data from corruption.

Polymorphism

- Polymorphism means that the same operation may behave differently on different classes.
- Example: compute Payroll.

Message

- Objects perform operations in response to messages. For example, you may communicate with your computer by sending it a message from hand-help controller.

The software development process

- It consists of analysis, design, implementation, testing and retirement is to transform users' needs into a software solution that satisfies needs.

- It is tempting to ignore the process and plunge into the implementation and programming phases of software development.
- Programmers have been able to ignore the counsel of systems development is a building a system.
- The development itself in essence is a process of change, refinement, transformation to the existing product.
- The object-oriented approach provides us a set of rules for describing inheritances and specialization in a consisted way when a sub process changes the behavior of its parent process.
- The process can be divided into small, interacting phases sub process.
- Each sub process have following:
  - A description in terms of how it works.
  - Specification of the input required for the process.
  - Specification of the output to be produced.
- The software development process can be viewed in a series of transformations where the output of one transformation becomes the input of the subsequent transformation.

Transformation 1

1. It is analysis translates the users' needs into system requirements and responsibilities.
2. The way they use the system can provide insight into the user's requirements.
3. For example, one use of the system might be analyzing an incentive payroll system which will tell us that this capacity must be included in the system requirements.
4. Software process reflecting transformation from needs to a software product that

Transformation 2

1. It comes and explains about the design part.
2. It begins with a problem statement and ends with a detailed design that can be transformed into an operational system.
3. This transformation includes the bulk of the software development activity.
4. It also includes the design descriptions, the program and the testing materials.

Transformation 3

1. In this discuss about the implementation part.
2. Implementation refines the detailed design into the system deployment that will satisfy the users needs.
3. This takes into account the equipment, procedures, people and the like.
4. It represents embedding environment product with its operational environment.
5. For example, the new compensation method is programmed, new forms are put to use and new reports now can be printed.
6. In the real world, the problems are not always well-defined and that is why the waterfall model has utility.
7. For example, it a company has expenses in building accounting system, then building another such product based on the existing design is best managed with the waterfall model.
8. This model assumes that the requirements are known before the design begins.
9. But one may need experience with the product before the requirements can be fully understood.
10. It also assumes that the requirements will remain static over the development cycle.
11. That a product delivered months after it was specified will meet the delivery time needs.

The Object Model

- Object oriented development offers a different model from the traditional software development approach, which is based on functions and procedures.
- An Object-Oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real world “Objects”.
- Object are defined, it will perform their desired functions and seal them off in our mind like black boxes.

- The object- Oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.
- An objective orientation producers system that are easier evolve, move flexible more robust, and more reusable than a top-down structure approach.
- An object orientation allows working at a higher level of abstraction.
- It provides a seamless transition among different phases of software development.
- It encourages good development practices.
- It promotes reusability.

The unified Approach (UA)

The unified Approach (UA) is the methodology for software development proposed and used.

The following concepts consist of Unified Approach

1. Uses-case driven development.
2. Utilizing the unified modeling language for modeling.
3. Object-Oriented analysis where it utilizing we case and object modeling.
4. Object-Oriented design
5. Responsibilities of reusable classes and maximum reuse.
6. The layered approach.
7. Incremental development and prototyping.
8. Continuous testing.

The Elements of an Object Model

The elements of an object model are classes and objects, attributes, operations and messages.

Class: A class is the definition of the behavior and properties of one or more objects within the system. A class binds the data (attributes) of an object to the behavior (operations) that it can perform.

Objects: 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.

Attributes: An attribute is a data value or state that describes an object and helps you to tell one object from another of the same class.

Operations: An operation is a behavior or function that an object can perform.

1. If the objects are required to implement a solution, then it is part of the solution space.
2. If the object is necessary only to describe a solution, it is part of the problem space.

Objects Can Have the Following Properties

1. External Entities

That produces or consumes information to be used by a computer-based system. Ex. Other systems, devices, people.

2. Things

Those are part of the information domain for the problem. Ex. Reports, Displays, Letters and Signals.

3. Occurrences

It is otherwise known as events that occur within the context of system operation. Ex. A property transfer or the completion of series of robot movement

4. Roles

It played by people who interact with the system.
Ex. Manager, Engineer, Sales Person.

5. Organization units

Those are relevant to an application. Ex. Division, Group, Team.

6. Places

That establishes the context of the problem and the overall function of the system. Ex. Manufacturing floor

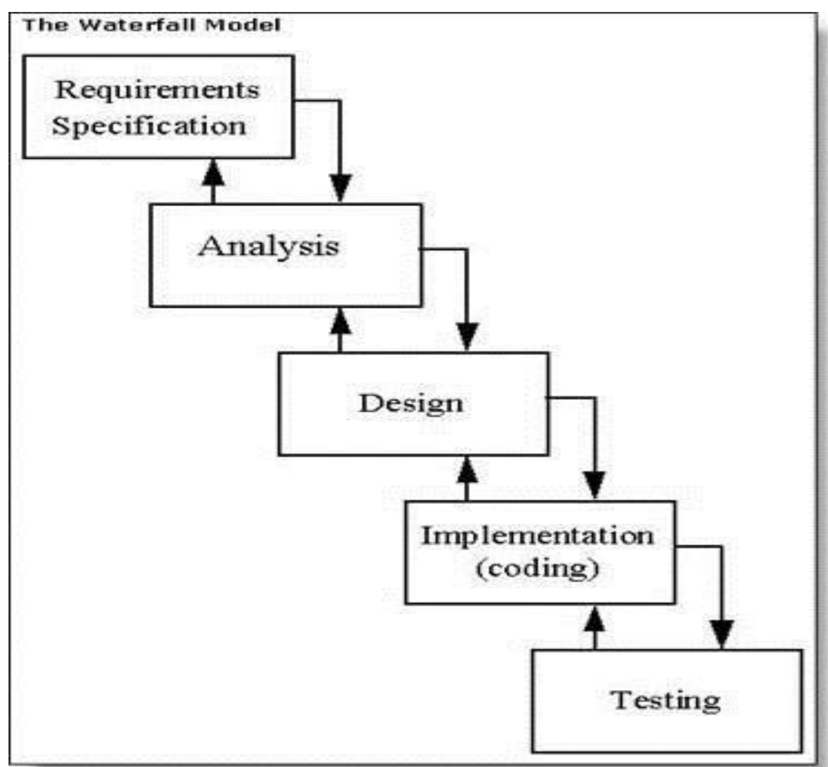
7. Structures

That defines a class of objects (or) in the extreme, related classes of objects.

UNIT –II

The Waterfall Software Development Process

Software development is the translation of a user need or marketing goal into software product. The **waterfall model** is a sequential software development model (a process for the creation of software) in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance.



Building High – Quality Software

1. Once system (Programs) exists, we must test it to see if it is tree of bugs.
2. High quality produces must meet user's needs and expectations.
3. The products should attain this with minimal (or) no defects, the focus being on improving products prior to delivery rather than correcting them after delivery.

4. To achieve high quality in software we need to be able to answer the following questions.
5. How do we determine when the system id reading for delivery?
6. Is it now an operational system that satisfies user's needs?
7. Is it correct and operating as we thought it should?
8. Does it pass an evaluation process?
9. There are two basis approaches to system testing.
10. This means of system evaluation in terms of four quality measures.

They are:

- Correspondence
- Correctness
- Verification
- Validation

Correspondence

It measures how well the delivered system matches the needs of the operational environment, as described in the original requirements statements.

Correctness

It measures the consistency of the product requirements with the respect to the design specification. Correctness determines whether (or) not the system correctly computes the results based on the rules created during the system analysis and design, measuring the consistency of product requirements with respect to the design specification.

Verification:

Verification is the task of determining correctness.
Eg: Am I building the product right?
Validation is the task of predicting correspondence.
Eg: Am I building the right product?

Systems Development Lifecycle

Object – Oriented software development life cycle (SDLC)

It consists of three macro processes.

1. Object – Oriented analysis.
2. Object – Oriented design.
3. Object – Oriented implementation.

Design

1. Object oriented system development includes these activities
2. Object oriented analysis use case driven
3. Object oriented design
4. Prototyping
5. Component – based development
6. Incremental testing

Validation:

- It is the task of predicting correspondence.
- True correspondence cannot be determined until the system is in place.

OBJECT – ORIENTED ANALYSIS USE CASE DRIVEN

1. The object oriented analysis phase of software development is concerned with determining the system requirements and identifying classes and their relationship to other classes in the problem domain.
2. The object-oriented programming community has adopted use cases to a remarkable degree.
3. The intersection among objects roles to achieve a given goal is called collaboration.
4. Expressing these high level processes and interactions with customers in a scenario and analyzing it is referred to use-case modeling.
5. For example, the objects in the incentive payroll system might include the following
examples.

a. The employee, worker, supervisor, office administrator.
b. The paycheck.
c. The process used to make the product.

Design

1. Object-Oriented design requires move rigor up front to do things right.

2. We need to spend more time gathering requirements developing requirements model and an analysis model, and then turning them into the design model.
3. Object-Oriented design centers on establishing design classes and their protocol.
4. Building class diagrams, user interfaces and usability based on usage and use cases.
5. The use-case concept can be employed through most of the activities of software development.

COMPONENT BASED DEVELOPMENT

1. Component based development (CBD) is an industrialized approach to software development.
2. Software components are functional units or building block offering a collection of reusable services.
3. A CBD developer can assemble components to construct a complete software system.
4. Components themselves may be constructed from other components and so on down to the level of prebuilt components (or) written in a language such as C, COBOL.
5. The object-oriented concept addressed analysis, design and programming; Whereas component-based development is concerned with the implementation and system integration aspects. Eg. Software development.

RAPID APPLICATION DEVELOPMENT

Rapid application development (**RAD**) is a term originally used to describe a software development process introduced by James Martin in 1991.
1. The rapid application development (RAD) approach to system development rapidly develops software to quickly.
2. Incrementally implement the design by using tools such as case.

Reusability

1. Reusability is a major benefit of object-oriented system development.
2. It is also the most difficult promise to deliver.
3. To develop reusability in the objects.

Meta Class

A meta-class is a class about a class. They are normally used to provide instance variables and operations.

Meta class in 7 points:

- Every object is an instance of a class
- Every class eventually inherits from Object
- Every class is an instance of a meta class
- The meta class hierarchy parallels the class hierarchy
- Every meta class inherits from Class and Behavior
- Every meta class is an instance of Meta class
- The meta class of Meta class is an instance of Meta class

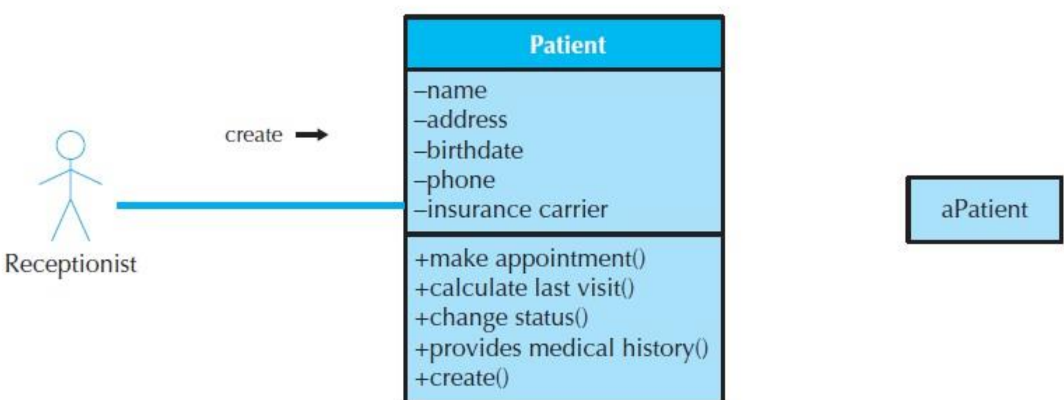
Advantages of object-oriented programming are:

- The ability to reuse code.
- Develop more maintainable systems in a shorter amount of time.
- More resilient to change.
- More reliable, since they are built from completely tested and debugged classes.

Information Hiding

- Information hiding was first promoted in structured systems development.

- The principle of information hiding suggests that only the information required using software module be published to the user of the module.
- Typically, this implies the information required to be passed to the module and the information returned from the module are published.
- Exactly how the module implements the required functionality is not relevant.
- We really do not care how the object performs its functions, as long as the functions occur.
- In object-oriented systems, combining encapsulation with the information-hiding principle suggests that the information-hiding principle be applied to objects instead of merely applying it to functions or processes.
- As such, objects are treated like black boxes.
- The fact that we can use an object by calling methods is the key to reusability because it shields the internal workings of the object from changes in the outside system, and it keeps the system from being affected when changes are made to an object.



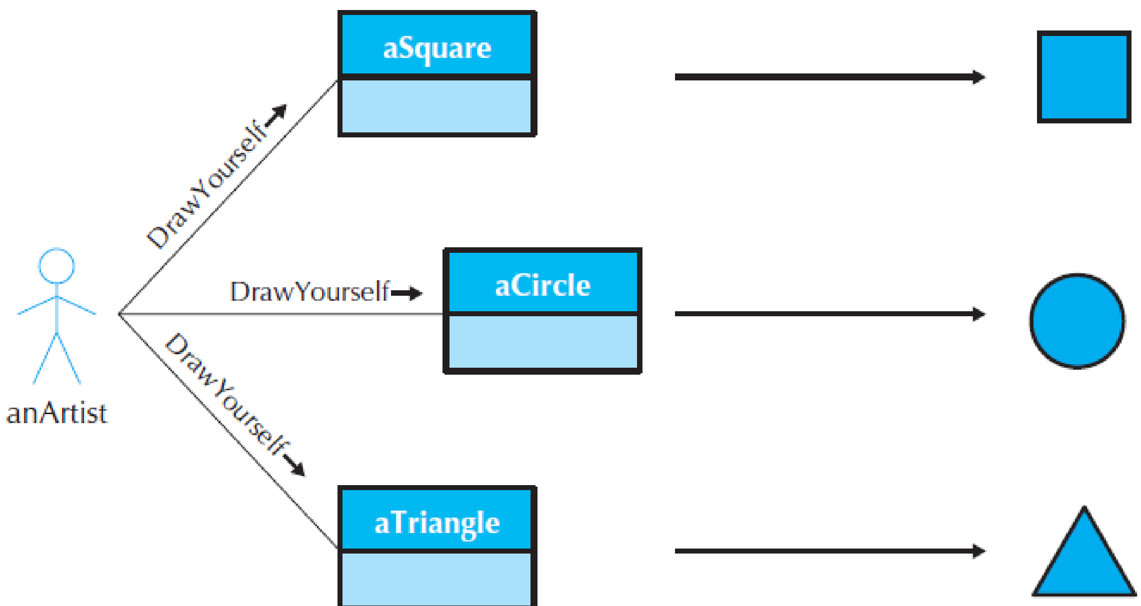
- In above figure, notice how a message (insert new patient) is sent to an object, yet the internal algorithms needed to respond to the message are hidden from other parts of the system.

- The only information that an object needs to know is the set of operations, or methods, that other objects can perform and what messages need to be sent to trigger them

Polymorphism

- Polymorphism means that the same message can be interpreted differently by different classes of objects.
- For example, inserting a patient means something different than inserting an appointment.
- As such, different pieces of information need to be collected and stored.
- We do not have to be concerned with how something is done when using objects.
- We can simply send a message to an object, and that object will be responsible for interpreting the message appropriately.

For example, if an artist sent the message Draw yourself to a square object, a circle object, and a triangle object, the results would be very different, even though the message is the same.



The above figure how each object responds appropriately (and differently) even though the messages are identical.

RELATIOPNSHIPS IN THE UML

There is a Semantic connection among elements. There are four kinds of relationships in the UML.

- Dependency
- Association
- Generalization
- Realization

Dependency

A dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of other thing. Change in structure or behavior of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it happens.

Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. We may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

Example: A Student and a Faculty are having an association.

Generalization

A generalization is a specification / generalization relationship in which objects of the specified element are substitutable of the generalized element. Generalization uses a "is-a" relationship from a specialization to the generalization class. Common structure and behavior are used from the specialization to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a "Is-a" relationship.

Example: Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

Realization

A realization is a semantic relationship between classifiers, where in one classifiers specifies a contract that another classifier guarantees to carry out. Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

Example: A particular model of a car 'GTB Fiorano' that implements the blueprint of a car realizes the abstraction.

We will encounter realization relationship in two places

- Between interfaces.
- The classes or components.

Aggregation

Aggregation is a special case of association. It is a directional association between objects. When an object 'has-a' another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a "Has-a" relationship.

Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

Example: A class contains students. A student cannot exist without a class. There exists composition between class and students.

Class diagram

1. The class diagram shows the building blocks of any object oriented system. This diagram describes a static view of the model to show what attributes and behaviors it has.
2. A class diagram shows a set of classes , interfaces, collaboration and their relationships.
3. Classes are represented by a rectangle which shows the name of the class, name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.
4. If the '+' symbol is used the attribute (or) operation has a public level of visibility.
5. If a '-'symbol is used, the attribute or operation is private.
6. In addition, '#' symbol allows an operation or attribute to be defined as protected.

7. The '~' symbol indicates package visibility. (grouping behavior)

flower
Fragrance – string color -string
Life() Use()

In above structure,

Flower – class name fragrance, color – attributes life (), use () – methods

Contents of class diagram:

It contains the following things.

i) Classes
ii) Interfaces
iii) Collaborations
iv) Relationships

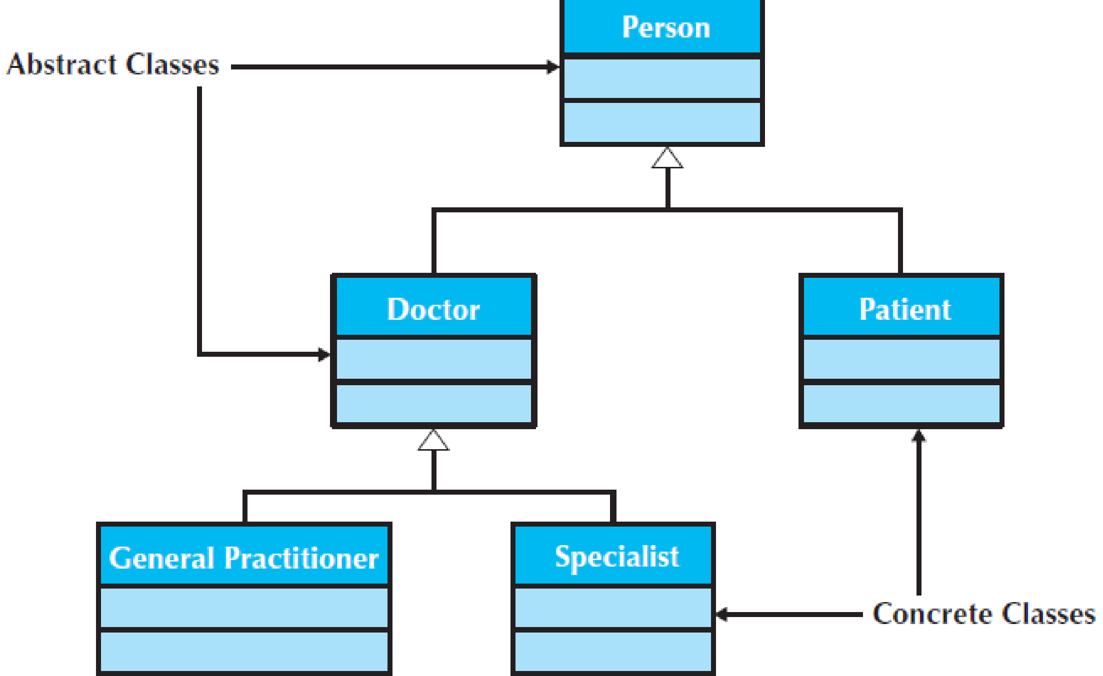
Common uses:

i) To model the vocabulary of a system.
ii) To model simple collaborations.
iii) To model a logical database schema.

Inheritance

- Inheritance, as an information systems development characteristic, was proposed in data Modeling.

- The data modeling literature suggests using inheritance to identify higher-level, or more general, classes of objects. Common sets of attributes and methods can be organized into super classes.
- Typically, classes are arranged in a hierarchy, whereby the super classes, or general classes, are at the top, and the subclasses, or specific classes, are at the bottom.
- In Figure, person is a super class to the classes Doctor and Patient. Doctor, in turn, is a super class to General Practitioner and Specialist.
- How a class (e.g., Doctor) can serve as a super class and subclass concurrently.
- The relationship between the class and its super class is known as the a-kind-of relationship.
- For example in Figure, a General Practitioner is a-kind-of Doctor, which is a-kind-of Person.
- Subclasses inherit the appropriate attributes and methods from the super classes above them. That is, each subclass contains attributes and methods from its parent super class.



- For example, both Doctor and Patient are subclasses of Person will inherit the attributes and methods of the Person class.
- Inheritance makes it simpler to define classes.
- Instead of repeating the attributes and methods in the Doctor and Patient classes separately, the attributes and methods that are common to both are placed in the Person class and inherited by those classes below it.
- Most classes throughout a hierarchy will lead to instances; any class that has instances is called a concrete class.
- For example, if Mary Wilson and Jim Maloney were instances of the Patient class, Patient would be considered a concrete class.
- The classes are referred to as abstract classes. Person is an example of an abstract class.

Rambaugh's Object Modeling Technique

- ✓ This method supports all of the main object oriented semantics ie) inheritance, aggregation, association etc.,
- ✓ It must cover analysis, design, and implementation using oo techniques.
- ✓ It supports real time applications and management information system.
- ✓ OMT is fast and ability approach for identifying attributes, methods, inheritance and association.
- ✓ It should not be over complex.
- ✓ The combination of object, dynamic and functional models supports many kinds of applications.

OMT phases:

i) Analysis – the results are object dynamic and functional model.

ii) System Design – the results are structure of the system along with high level strategy decision.

iii) Object design – produces the detailed design for object dynamic and functional model.
iv) Implementation - produces reusable extensible and robust code.

OMT MODELS:

- Object Model
- Dynamic Model
- Functional Model

Object Model:

- It is central to the method and user diagrams which are similar to "E-R diagrams".
- It describes structure of objects in a system.
- Classes and their attributes and operations are shown together with relationships between classes.
- The relationship which can be drawn includes inheritance, aggregation and association.
- It is graphically represented by class diagram.
- Each class represents individual objects.

Example:

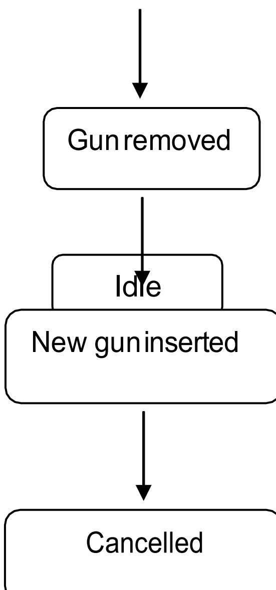
class
attributes
operations
vehicle
Color Width model

Move()

Repair()

Dynamic Model

- This model consists of a state transition diagram for each class have states, transition, events and actions.
- Each state receives one or more events and it makes the transition to the next state.
- The next state depends on the current state as well as events.



Functional Model

- This model uses a hierarchy of data flow diagram shows the flow of data between different processes in a business.
- However, the DFD'S are very useful to understand the system process.

Symbols used in data flow:

1. Process:

   Any fuction being performed. Example: verify password.

2. The data flow:

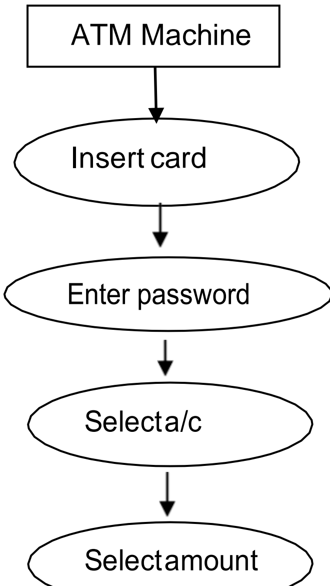
   Direction of data movement. Example: PINCODE

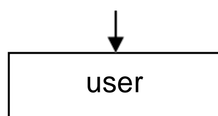
3. The data store:

   Location, where data are stored. Example: account

4. External entity:

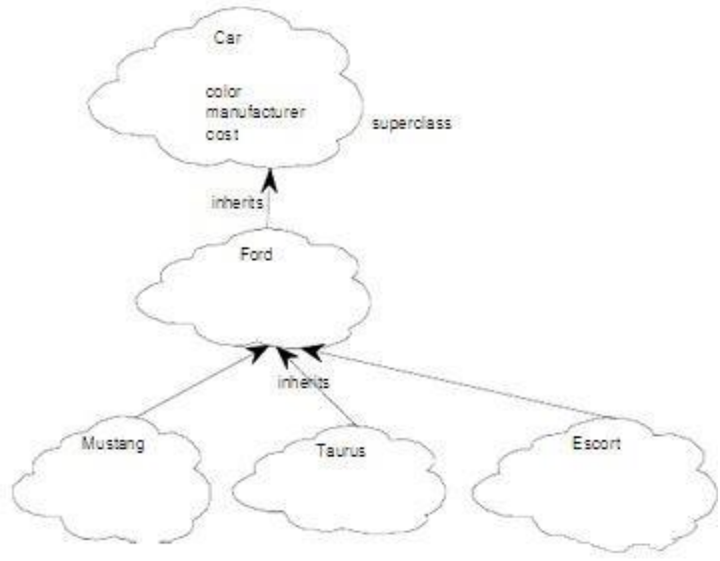
   Source or destination of data. Example: ATM card reader.





Booch methodology .

- The Booch methodology provides an Object Oriented development in the analysis and design phases.
- The analysis phase contains requirements, high level description of the system function and structure.
- It also defines classes and their attributes, inheritance, methods.
- Once the analysis phase is completed, the Booch methodology develops the architecture in the design phase.
- Booch uses large set of symbols.
- The Booch Methodology consists of the following diagrams:
    - Class diagram
    - Object diagram
    - State transition diagram
    - Module diagram
    - Process diagram
    - Interaction diagram
- The Booch methodology recommends a macro development process and micro development process.



i) **The Macro Development Process**

a. It controls the framework for micro development process
b. It can take time duration of weeks or even months.
c. It is technical management of a system.
d. Phases:
   i. Conceptualization
   ii. Analysis and development of the model
   iii. Design or create the system architecture
   iv. Evolution or implementation
   v. Maintenance

- **Conceptualization:**
  a. Requirements
  b. Project goals
  c. Prepare a prototype to prove the concept
- **Analysis and development:**
  a. Use class diagrams to describe roles and their responsibilities.
  b. Use object diagrams to describe the behavior of a system.

c. Use interaction diagrams to describe the behavior of a system.

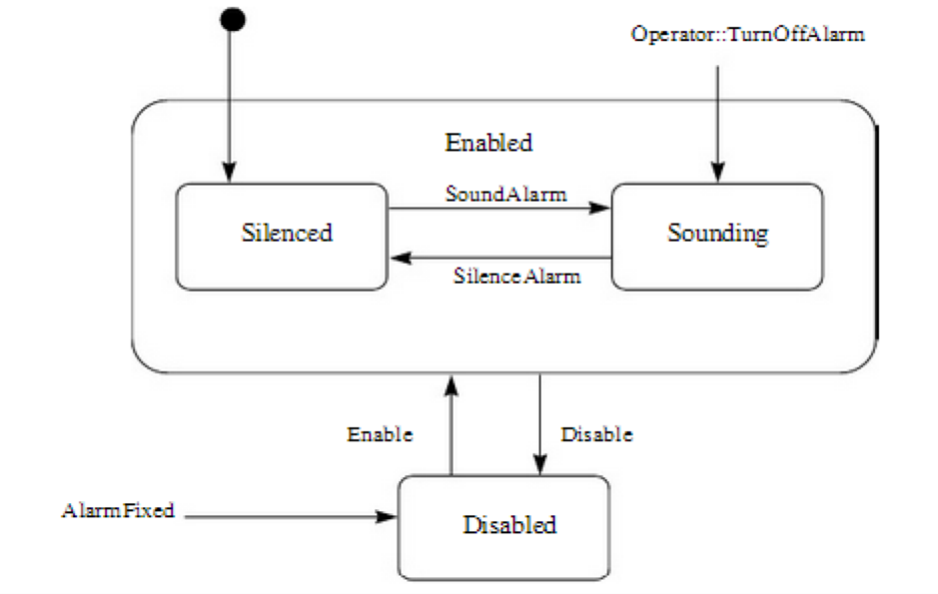
- **Design or system architecture**
  a. What class exists and relationships between classes.
  b. Use object diagrams to represent collaborations.
  c. Use module diagram to represent how objects and classes are declared.
  d. Use process diagrams to represent which processor to allocate process.
- **Evolution or implementation:**
  a. Create source code with desired programming language.
- **Maintenance:**
  a. Changes to the system to add new requirements and eliminates the bugs.

ii) The Micro Development Process:

a. Description of day-to-day activities
b. Steps:
   i. Identify classes and objects.
   ii. Identity classes, objects and semantics.
   iii. Identity classes, objects, & relationships
   iv. Identify classes, objects, interface and implementation.

Example:

Interaction diagram:



The Jacobson Methodology.

It contains

i) Use cases
ii) Object Oriented Business Engineering(OOBE)
iii) Object Oriented Software Engineering(OOSE)
iv) Objectory (Object Factory for software development)

Use cases

- Use cases are scenarios for understanding system requirements.
- Use case is a description of a system's behavior and its respond to a requirement from outside of that system. In other words a use case describes 'who' can do and 'what' with the system.
- A use case defines the interactions between external actors and the system.
- An actor specifies a role played by a person. It also describes the actor to achieve a particular goal.

- A use case is a methodology used in system analysis to identify, clarify, and organize system requirements.
- Use case is made up of possible sequences of interactions between system and users in a particular environment and related to a particular goal.
- It consists of a group of elements (classes and interfaces) and it includes all system activities.
- A use case has the following characteristics.

    - Organize functional requirements.
    - Models the goal of a system and actor.
    - Describes main flow of events.
    - Use case can use functionality of another one.

Object Oriented Software Engineering (OOSE)

- Suitable for large and real time systems.
- The development process is called use case driven developmental process. This process stress analysis, design, validation and testing.

Models for expressing all the software development phases

1) Use case model
2) Domain object Model
3) Analysis object Model
4) Implementation object Model
5) Test model

Use case model: It tells about outside actors and behavior of the system.

Domain object Model: Mentions the real world objects.

Analysis object Model: tells how source code should be carried out and written.

Implementation object Model: specifies all the implementation details such as DFD's & ER-model, database structure etc.,

Testing Model: To remove bugs.

Example: Unit testing, white box testing, black box testing, stress testing, system testing, integration testing etc.,

Object Oriented Business Engineering (OOBE)

- Mainly used for enterprise level.
- Main sources are use cases.
- Phases:

i) Analysis - Mention about the problems and requirements.
ii) Design & implementation - DBMS candidates to programming languages, GUI tools.
iii) Testing - Types of testing correct bugs.

Building Blocks of UML

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

(1) Things

Things are the most important building blocks of UML. Things can be:

- Structural
- Behavioral
- Grouping
- Annotational

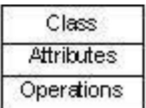
Structural things

The **Structural things** define the static part of the model. They represent physical and conceptual elements.

Following are the brief descriptions of the structural things.

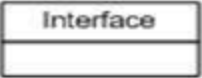
Class:

Class represents set of objects having similar responsibilities.



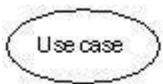
Interface:

Interface defines a set of operations which specify the responsibility of a class.



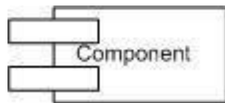
Collaboration:

Collaboration defines interaction between elements.



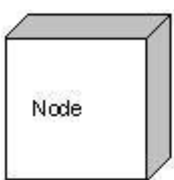
Use case:

Use case represents a set of actions performed by a system for a specific goal.



Component:

Component describes physical part of a system.



Node:

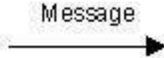
A node can be defined as a physical element that exists at run time.

Behavioral things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

Interaction:

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine:

State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

Grouping things:

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

Package:

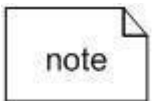
Package is the only one grouping thing available for gathering structural and behavioral things.

Annotational Things:

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one Annotational thing available.

Note:

A note is used to render comments, constraints etc of an UML element.



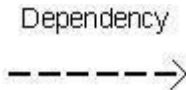
(2) Relationship

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

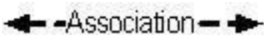
Dependency:

Dependency is a relationship between two things in which change in one element also affects the other one.



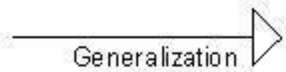
Association:

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



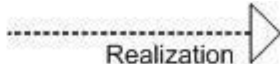
Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



Realization:

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.



UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

UML includes the following nine diagrams and the details are described in the following chapters.

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Activity diagram

7. State chart diagram
8. Deployment diagram
9. Component diagram

UML Class Diagram

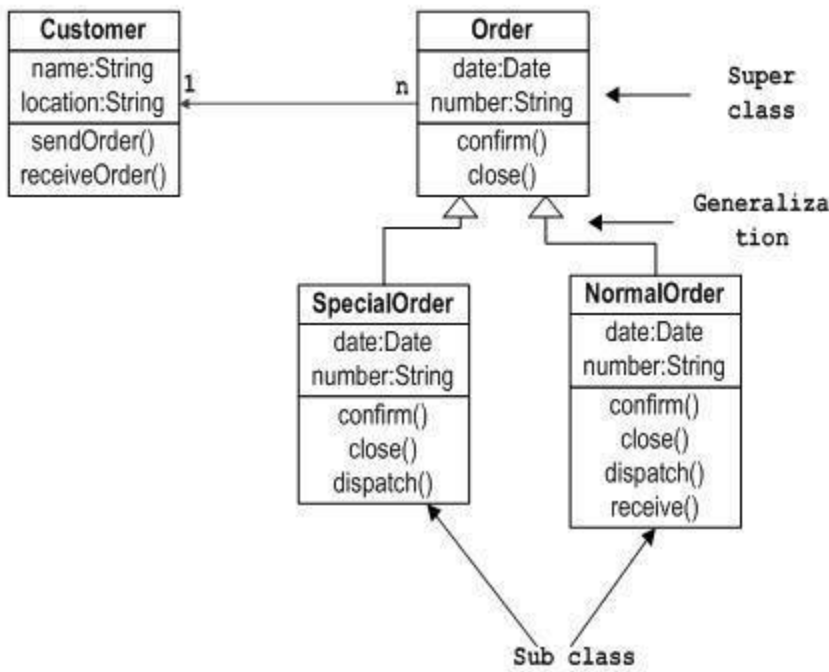
The class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing and documenting different aspects of a system but also for constructing executable code of the software application.

The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages.

The class diagram shows a collection of classes, interfaces, associations, collaborations and constraints. It is also known as a structural diagram.

Purpose:

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.



Class diagrams are used for:

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

UML Object Diagram

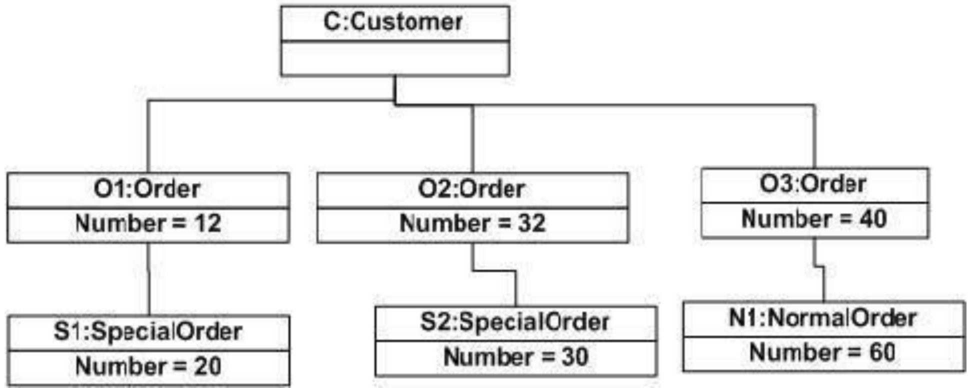
Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object diagrams are used to render a set of objects and their relationships as an instance.

Purpose:

- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behavior and their relationship from practical perspective



Object diagrams are used for:

- Making the prototype of a system.
- Reverse engineering.
- Modeling complex data structures.
- Understanding the system from practical perspective.

UML Component Diagram

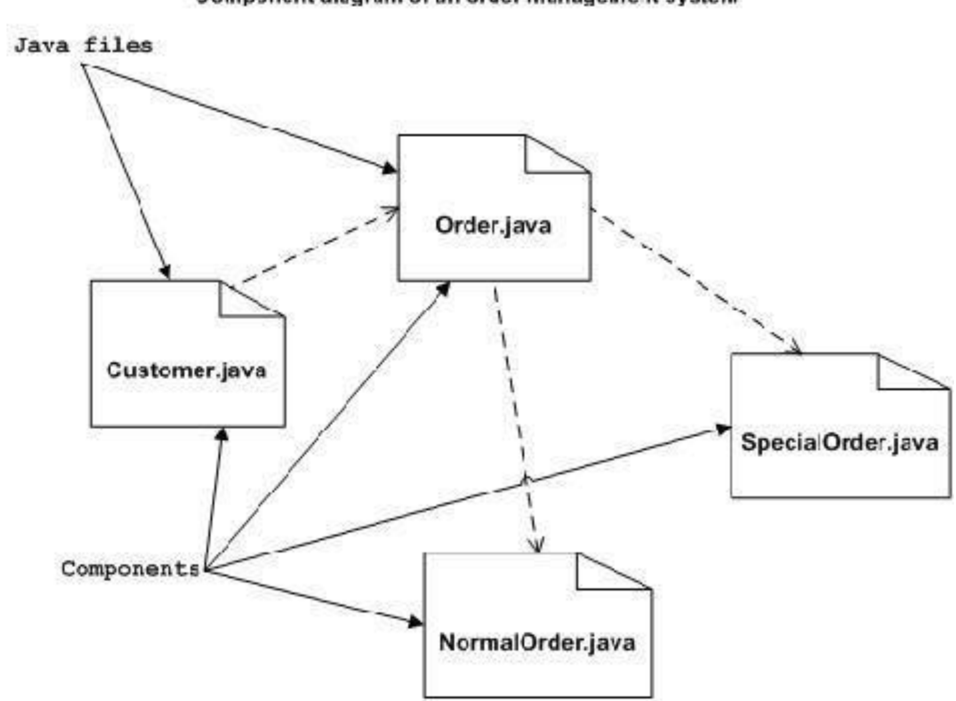
Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system.

Now the question is what are these physical aspects? Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node.

So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose:

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.



Now the usage of component diagrams can be described as:

- Model the components of a system.
- Model database schema.

- Model executables of an application.
- Model system's source code.

UML Deployment Diagram

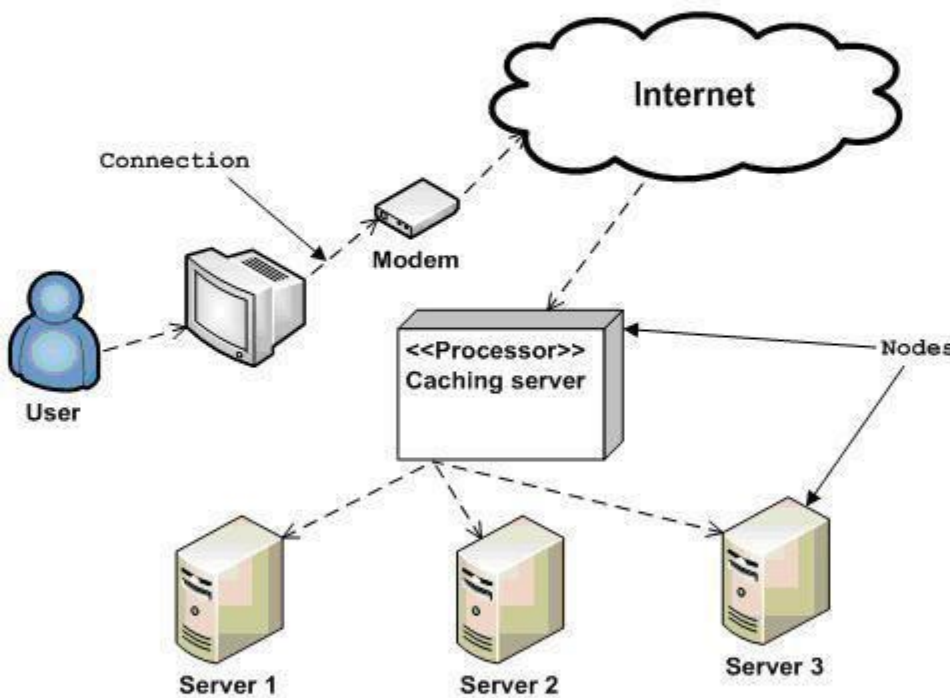
Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed.

So deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose:

- Visualize hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe runtime processing nodes.

Deployment diagram of an order management system



So the usage of deployment diagrams can be described as follows:

- To model the hardware topology of a system.
- To model embedded system.
- To model hardware details for a client/server system.
- To model hardware details of a distributed application.
- Forward and reverse engineering.

UML Use Case Diagram

To model a system the most important aspect is to capture the dynamic behavior. To clarify a bit in details, dynamic behavior means the behavior of the system when it is running /operating.

So only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML there are five

diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. So use case diagrams are consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

So to model the entire system numbers of use case diagrams are used.

Purpose:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

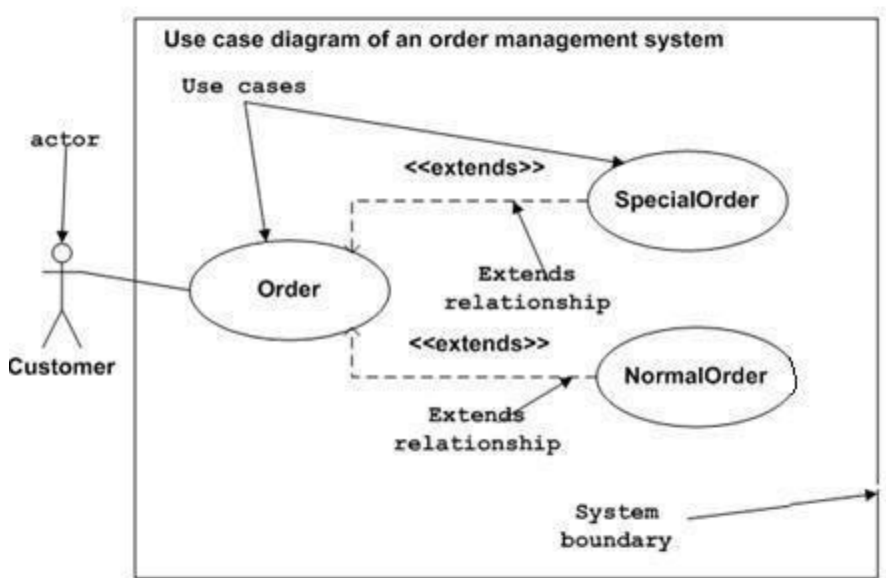


Figure: Sample Use Case diagram

So the following are the places where use case diagrams are used:

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

UML Interaction Diagram

From the name Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. So this interaction is a part of dynamic behavior of the system.

This interactive behavior is represented in UML by two diagrams known as Sequence diagram and Collaboration diagram. The basic purposes of both the diagrams are similar.

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Purpose:

- To capture dynamic behavior of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

We have two types of interaction diagrams in UML. One is sequence diagram and the other is a collaboration diagram.

The sequence diagram captures the time sequence of message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

In a brief the following are the usages of interaction diagrams:

- To model flow of control by time sequence.

- To model flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

UML State chart Diagram

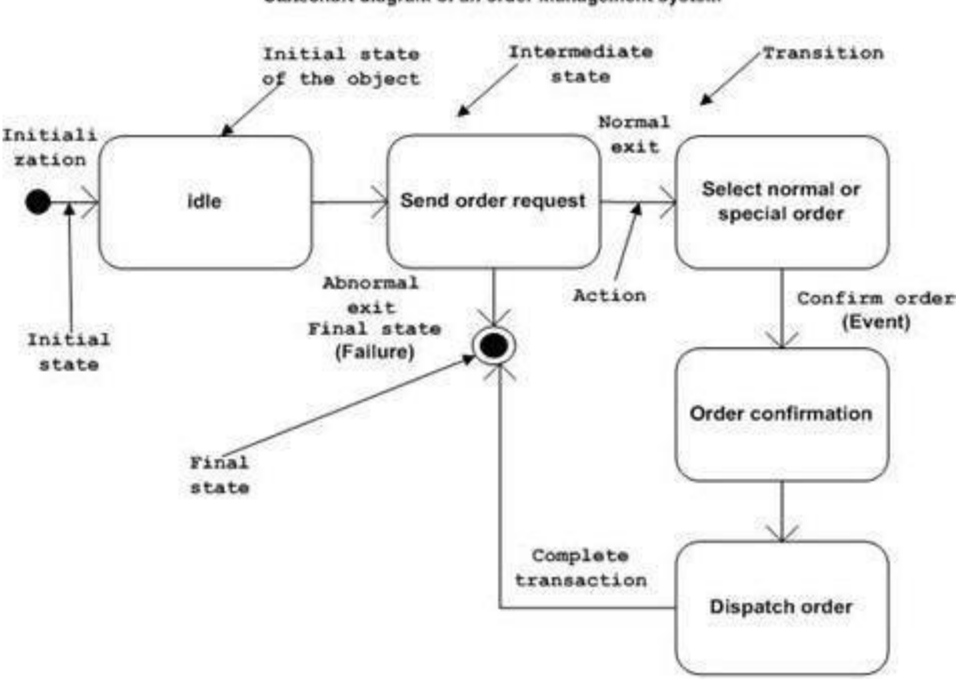
The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A State chart diagram describes a state machine. Now to clarify it state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity diagram is a special kind of a State chart diagram. As State chart diagram defines states it is used to model lifetime of an object.

Purpose:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.



So the main usages can be described as:

- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

UML Activity Diagram

Activity diagram is another important diagram in UML to describe dynamic aspects of the system.

Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.

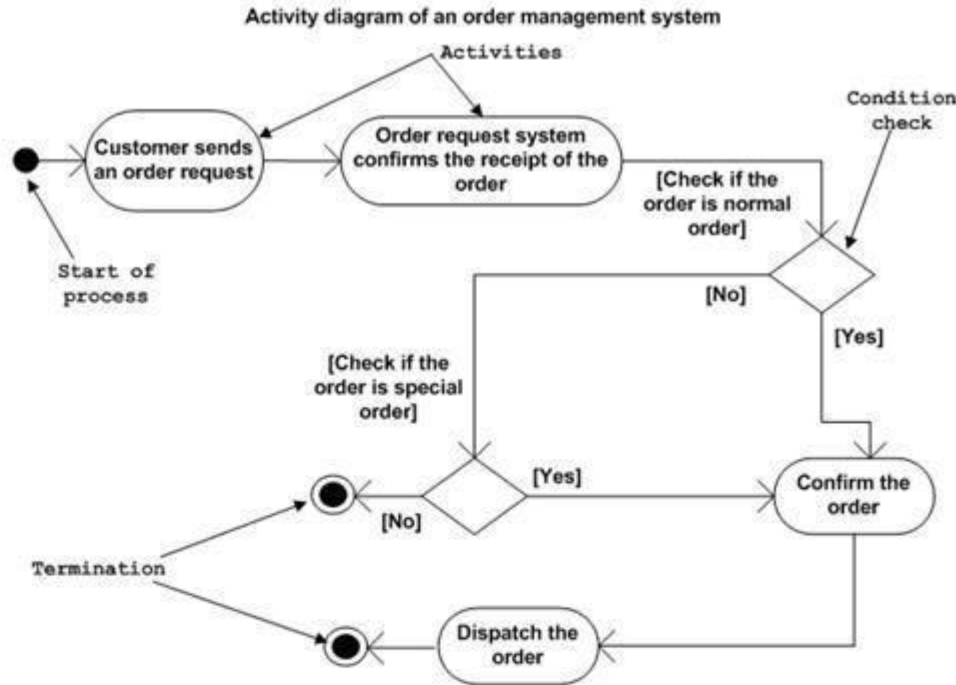
So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

Purpose:

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints



Following are the main usages of activity diagram:

- Modeling work flow by using activities.
- Modeling business requirements.
- High level understanding of the system's functionalities.
- Investigate business requirements at a later stage.

Unified Approach

The idea behind the UA is not to introduce yet another methodology. The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams.

The unified approach to software development revolves around (but is not limited to) the following processes and components.

The UA processes

- Use-case driven development.
- Object-oriented analysis.
- Object-oriented design.
- Incremental development and prototyping.
- Continuous testing.

UA Methods and Technology

- Unified modeling language (UML) used for modeling.
- Layered approach.

- Repository for object-oriented system development patterns and frameworks.
- Promoting Component-based development.

UA Object-Oriented Analysis: Use-Case Driven

- The use-case model captures the user requirements.
- The objects found during analysis lead us to model the classes.
- The interaction between objects provides a map for the design phase to model the relationships and designing classes.

UA Object-Oriented Design

- Booch provides the most comprehensive object-oriented design method.
- However, Booch methods can be somewhat imposing to learn and especially tricky to figure out where to start.
- UA realizes this by combining Jacobson et al.'s analysis with Booch's design concept to create a comprehensive design process.

Iterative Development and Continuous Testing

- The UA encourages the integration of testing plans from day 1 of the project.
- Usage scenarios or Use Cases can become test scenarios; therefore, use cases will drive the usability testing.

Modeling Based on the Unified Modeling Language

- The UA uses the unified modeling language (UML) to describe and model analysis and design phases of system development.

The UA Proposed Repository

• The requirement, analysis, design, and implementation documents should be stored in the repository, so reports can be run on them for traceability.

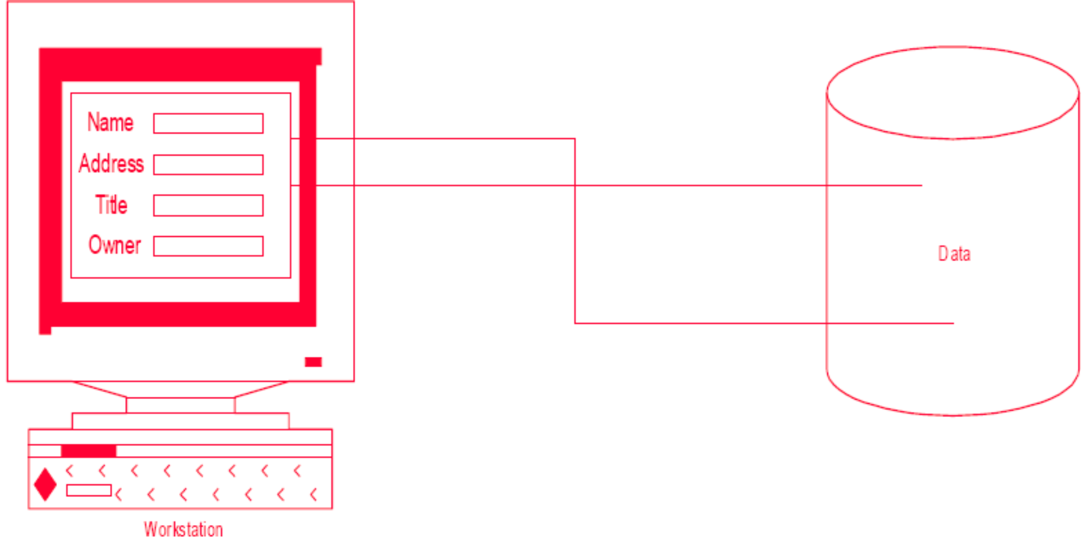
• This allows us to produce designs that are traceable across requirements, analysis, design, implementation, and testing.

The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known as two-layered architecture: interface and data.

Two-Layer Architecture

In a two-layer system, user interface screens are tied directly to the data through routines that sit directly behind the screens



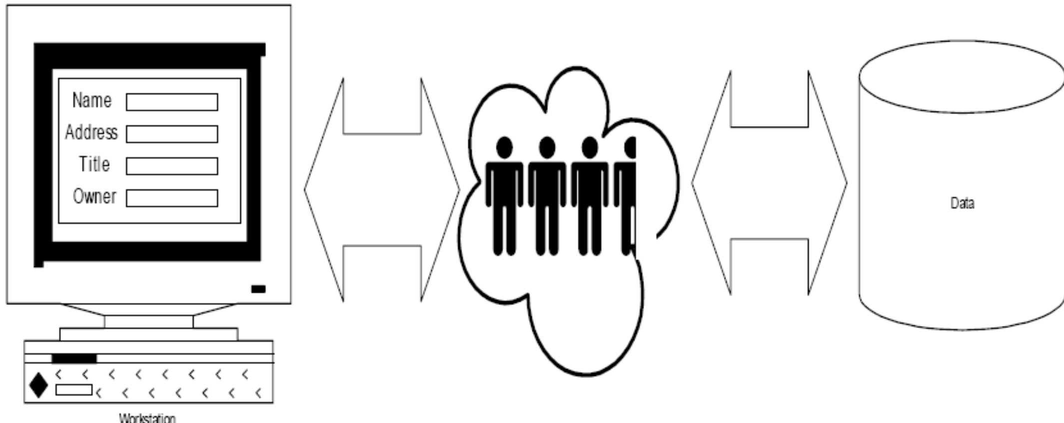
Problem with the Two-Layer Architecture

This approach results in objects that are very specialized and cannot be reused easily in other projects.

Three-Layer Architecture

Your objects are completely independent of how:

- they are represented to the user (through an interface) or
- how they are physically stored.



User Interface layer

This layer is typically responsible for two major aspects of the applications:

• Responding to user interaction

• Displaying business objects.

Business Layer

• The responsibilities of the business layer are very straight- forward:

• model the objects of the business and how they interact to accomplish the business processes.

Business Layer: Real Objects

These objects should not be responsible for:

1. Displaying details

2. Data access details

Access Layer

The access layer contains objects that know how to communicate with the place where the data actually resides, whether it is a relational database, mainframe, Internet, or file. The access layer has two major responsibilities: Translate request, Translate result.

Database Interface

The interface on a database must include a data definition language (DDL), a query, and data manipulation language (DML). These languages must be designed to fully reflect the flexibility and constraints inherent in the data model. Database systems have adopted two approaches for interfaces with the system.

i) To Embed a Database Language.
ii) To Extend the host programming language.

To Embed a Database Language:

- One is to embed a database language, such as structured query language (SQL), in the host programming language.
- This approach is a very popular way of defining and designing a database and its schema such as SQL, which has become an industry standard for defining databases.
- The problem with this approach is that application programmers have to learn and use two different languages.

To Extend the host programming language:

- To extend the host programming language with Database related constructs.

- This is the major approach, since application programmers need to learn only a new construct of the same language rather than a completely new language.
- Many of the currently operational databases and object-oriented database systems have adopted this approach; a good example is GemStone from Servio Logic, which has extended the Smalltalk object-oriented programming.

Database Schema and Data Definition Language

To represent information in a database, a mechanism must exist to describe or specify to the database the entities of interest.

- ✓ A data definition language (DDL) is the language used to describe the structure of and relationships between objects stored in a database.
- ✓ This structure of information is termed the database schema. In traditional databases, the schema of a database is the collection of record types and set types or the collection of relationships, templates, and table records used to store information about entities of interest to the application.
- ✓ For example, to create logical structure or schema, the following SQL command can be used:

```
CREATE SCHEMA AUTHORIZATION (creator)

CREATE DATABASE (database name)
```

For example,

```
CREATE TABLE INVENTORY

(Inventory_Number      Char(10) Not Null

 Description           Char(25) Not Null

 Price                 Decimal (9, 2));
```

- ✓ The structured query language (SQL) is the standard DML for relational DBMSs. SQL is widely used for its query capabilities.
- ✓ The query usually specifies
  - ➔ The domain of the discourse over which to ask the query.
  - ➔ The elements of general interest.
  - ➔ The conditions or constraints that apply.
  - ➔ The ordering, sorting, or grouping of elements and the constraints

- ✓ DML are either procedural or nonprocedural.
- ✓ A procedural DML requires users to specify what data are desired and how to get the data.
- ✓ A nonprocedural DML, like most databases' fourth generation programming language (4GLs), requires users to specify what data are needed but not how to get the data.
- ✓ Object-oriented query and data manipulation languages, such as Object SQL, provide object management capabilities to the data manipulation language.
- ✓ In a relational DBMS, the DML is independent of the host programming language.
- ✓ A host language such as C or COBOL would be used to write the body of the application.
- ✓ Typically, SQL statements then are embedded in C or COBOL applications to manipulate data.
- ✓ Once SQL is used to request and retrieve database data.
- ✓ The SQL retrieval must be transformed into the data structures of the programming language.
- ✓ A disadvantage of this approach is that programmers code in two languages,
  SQL and the host language:
  - ✓ For example, to check the table content, the SELECT command is used, followed by the desired attributes. Or, if weou want to see all the attributes listed, use the (*) to indicate all the

attributes: SELECT DESCRIPTION, PRICE FROM INVENTORY where inventory is the name of a table.

Multi-Database Systems

o Heterogeneous information systems facilitate the integration of heterogeneous information sources, where they can be structured, semi – structured & sometimes unstructured. Such heterogeneous information systems are referred to as federated multi-database systems

o A multi-database systems (MDBS) is a database system that resides unobtrusively on top of, say, existing relational and object databases & file systems & presents a single database illusion to users

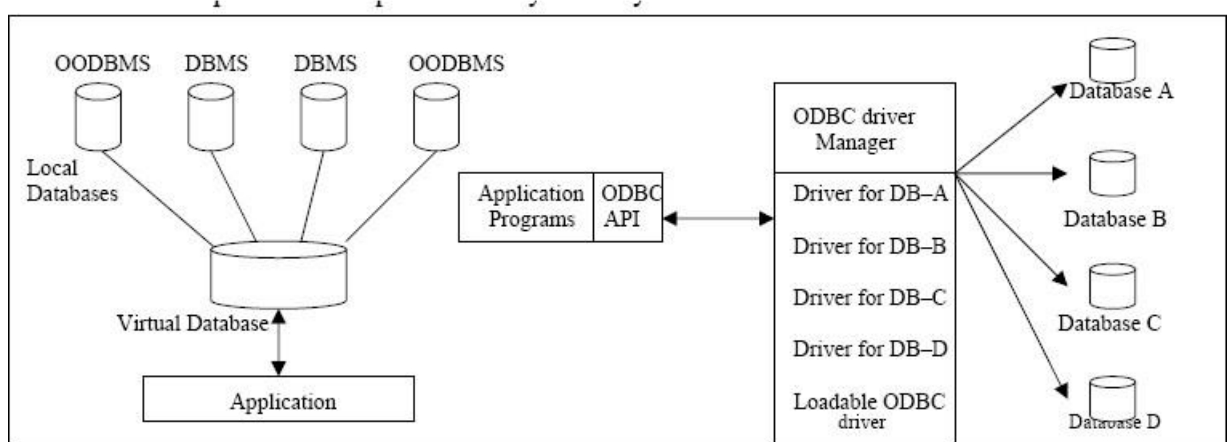
o The global schema is constructed by consolidating (integrating) the schemata of local databases; schematic differences (conflicts) among them are handled by neutralization (homogenization), the process of consolidating the local schemata

o The distinctive characteristics of multi-database systems: -

- ✓ Automatic generation of a unified global database schema from local databases, in addition to schema capturing and mapping for local databases.
- ✓ Provision of cross–database functionality (global queries, transactions) by unified schema.
- ✓ Integration of heterogeneous database systems with multiple databases
- ✓ Integration of data types other than relational data through use of tools – driver generators, etc

✓ Provision of a uniform but diverse set of interfaces (e.g. an SQL –style & C++) to access & manipulate data stored in local databases

o Open Database Connectivity: Multi-database Application Programming interfaces: The benefits of being able to port database applications by writing to an API for a virtual DBMS are so appealing to s/w developers that computer industry recently introduced several multi-database API's.



ODBC is an application programming interface that provides solutions to multi-database programming problem. Advantages:

1. It allows s/w developers to develop application without burden of learning MDBS APIs
2. It has ability to store data for various applications or data from different sources in any database & transparently access or combine the data on an as needed basis.
3. Details of back-end are hidden from users

The application interacts with the ODBC driver manager which sends the application calls (SQL statements) to database. The driver manager loads & unloads drivers, performs status checks, & manages multiple connections between applications & data sources

ODBC provides a mechanism for creating a virtual DBMS

Object Oriented Database Management Systems (OODBMS):

o To meet broader class of applications with unconventional & complex data type requirements, popularity of OOP lead to concept of OODBMS. It is combination of OOP and database technology which is popularly called as object oriented databases.

o The rules of OODBMS form combination of both OO language properties & DBMS properties.
The rules that make an object oriented system are as follows:

- ✓ The system must support complex objects which combination of atomic types of objects
- ✓ Object identity must be supported and existence independent of its values.
- ✓ Objects must be encapsulated with program & its data embodies separation of interface.
- ✓ The system must support types or classes with their individual concepts.
- ✓ The system must support inheritance whose advantage is shared codes & interfaces.
- ✓ The system must avoid premature binding also known as late binding or dynamic binding.
- ✓ The system must be computationally complete.
- ✓ The system must be extensible equal to status of system predefined types.
- ✓ These requirements are met by most OOP languages like C++, etc.

Then comes the set of rules make a DBMS are

- ✓ It must be persistent, able to remember an object state – data survive beyond execution time.
- ✓ It must be able to manage very large databases – storage & provide performance features.
- ✓ It must accept concurrent users – support notions of atomic, serializable transactions.

- ✓ It must be able to recover from h/w & s/w failures – return to a coherent state.
- ✓ Data query must be simple – graphical browser might fulfill this requirement sufficiently.

Oriented Design Process and Corollaries:

Design Processes:

During the design phase the classes identified in OOA must be revisited with a shift in focus to their implementation. New classes or attributes & methods are to added for implementation purposes & user interfaces.

The object–oriented design process consists of following activities

1. Apply design axioms to design classes, their attributes, methods, associations, structures & protocols. It constitutes two separate steps.

⇒ Refine & complete the static UML class diagram by adding details. This steps consists of.

Refine attributes.

Design methods & protocols by UML activity diagram to represent method's algorithm.

Refine associations between classes (if required).

Refine class hierarchy & design with inheritance (if required).

⇒ Iterate and refine again.

2. Design the access layer.

⇒ Create mirror classes: For every business class identified & created, create one access class.

⇒ Identify access layer class relationships.

⇒ Simplify classes & their relationships: Main goal – eliminate redundant classes & structures.

Redundant classes: Don't keep 2 classes that performs |||lr translate request & results.

Method classes: Try to eliminate classes with 1 or 2 methods by combining with others.

⇒ Iterate & Refine again.

3. Design the view layer classes.

⇒Design the macro level user interface, identifying view layer objects.

⇒ Design the micro level user interface, which includes the following activities.

Design view layer objects by applying the design axioms & corollaries

Build a prototype of view layer interface.

⇒ Test usability & user satisfaction

⇒ Iterate and refine

4. Iterate & refine the whole design. Reapply design axioms, if needed repeat preceding steps.

o Design must be traceable across requirements, analysis, design, code & testing. There must be clear step–by–step approach to design from the requirements model. All designed components must directly trace back to user requirements.

Corollaries: -

Many corollaries may be derived as a direct consequence of axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than original

axioms. They are also called design rules and are derived from 2 basic axioms

o 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.

◊ The main goal here is to maximize objects cohesiveness among objects & s/w components in order to improve coupling because only a minimal amount of essential information need be passed between components.

◊ Coupling is a measure of strength of association established by a connection from one object or s/w component to one another.

◊ The degree or strength of coupling between two components is measured by amount & complexity of information transmitted between them. Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface.

◊ Interaction coupling involves amount & complexity of messages between components. It is desirable to have little interaction.

◊ Inheritance is a form of coupling between super and sub classes. A sub class is coupled to its super class in terms of attributes and methods

◊ Cohesion reflects the "single–purposeless" of an object. Method cohesion function cohesion means that a method should carry only one function. A method that carries multiple functions is undesirable. Class cohesion means that all the class's methods & 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 single, clearly defined purpose. When we document, we 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.

◊ The smaller are classes, the better are chances of reusing them in other projects. Large & complex classes are too specialized to be reused.

◊ The primary benefit of s/w reusability is higher productivity. The s/w development team that achieves 80% reusability is 4 times as productive as team that achieves 20% reusability.

Corollary 4:
Strong mapping: There must be a string association between the physical system (analysis's object) & logical design (design's object)

Corollary 5:
Standardization: Promote standardization by designing inter changeable components and reusing existing classes or components

Corollary 6:
Design with inheritance: Common behavior (methods) must be moved ro super classes. The super class–subclass structure must make logical sense

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 relationship to other classes in the problem domain.

The object-oriented programming community has adopted use cases to a remarkable degree. The intersection among objects roles to achieve a given goal is called collaboration. Expressing these high level processes and interactions with customers in a scenario and analyzing it is referred to use-case modeling.

The objects in the incentive payroll system might include the following examples.
a. The employee, worker, supervisor, office administrator.

b. The paycheck.
c. The process used to make the product.

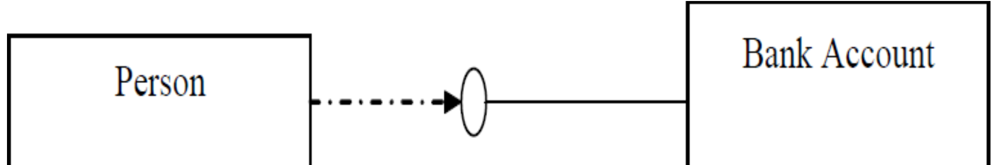
Design

1. Object-Oriented design requires move rigor up front to do things right.
2. We need to spend more time gathering requirements developing a requirements model and an analysis model, and then turning them into the design model.
3. Object-Oriented design centers on establishing design classes and their protocol.
4. Building class diagrams, user interfaces and usability based on usage and use cases.
5. The use-case concept can be employed through most of the activities of software development.

Class Interface Notation.

Class interface notation is used to describe the externally visible behavior of a class.

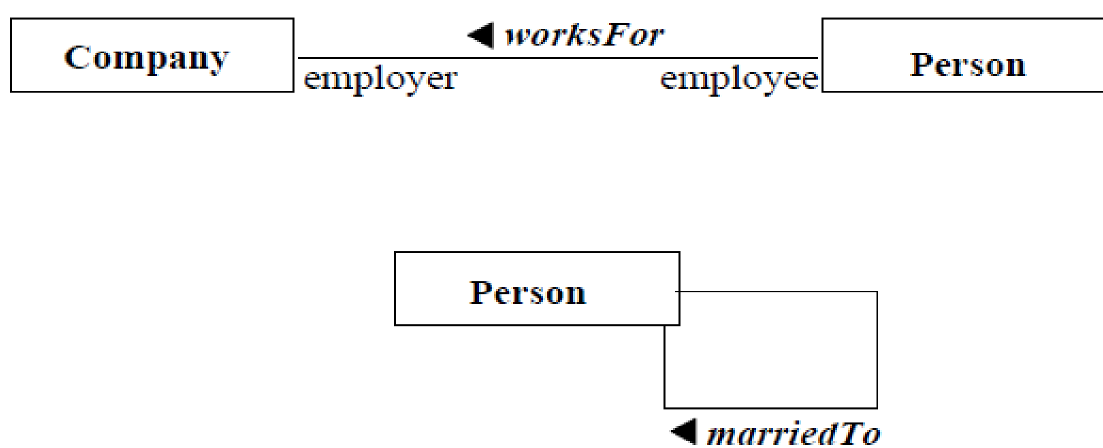
Example: an operation with public visibility.
Identifying class interfaces is a design activity of object-oriented system development
The UML notation for an interface is a small circle with the name of the interface connected to the class.
A class that requires the operations in the interface may be attached to the circle by a dashed arrow.
The dependent class is not required to actually use all of the operation



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 which is represented in Figure.

An association may have an association name.



Association Notation

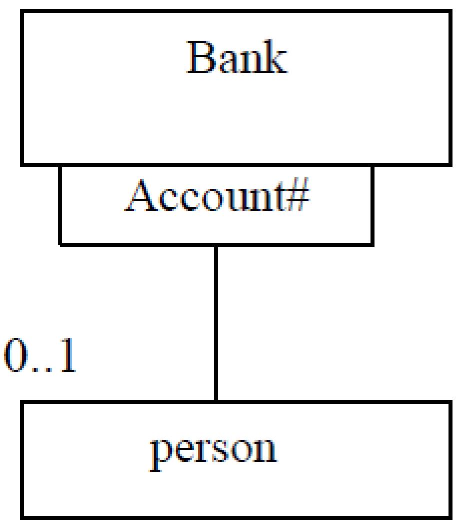
The association name may have an optional back triangle in it, which is illustrated in Figure.

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. Each object is an instance of the class. Classes are

organized hierarchically in a class tree, and subclasses inherit the behavior of their super classes. Good object oriented programming uses encapsulation and polymorphism, which when used in the definition of classes.
Objects have a lifetime.
They are explicitly created and exist for a period of time.
That traditionally, has been the duration of the process for which they were created.
A file (or) a database can provide support for objects having a longer lifetime longer than the duration of the process for which they well created.

Qualifier

• *Qualifier* is an association attribute. For example, a person object may be associated to a Bank object.
• An attribute of this association is the account#.
• The account# is the qualifier of this association.

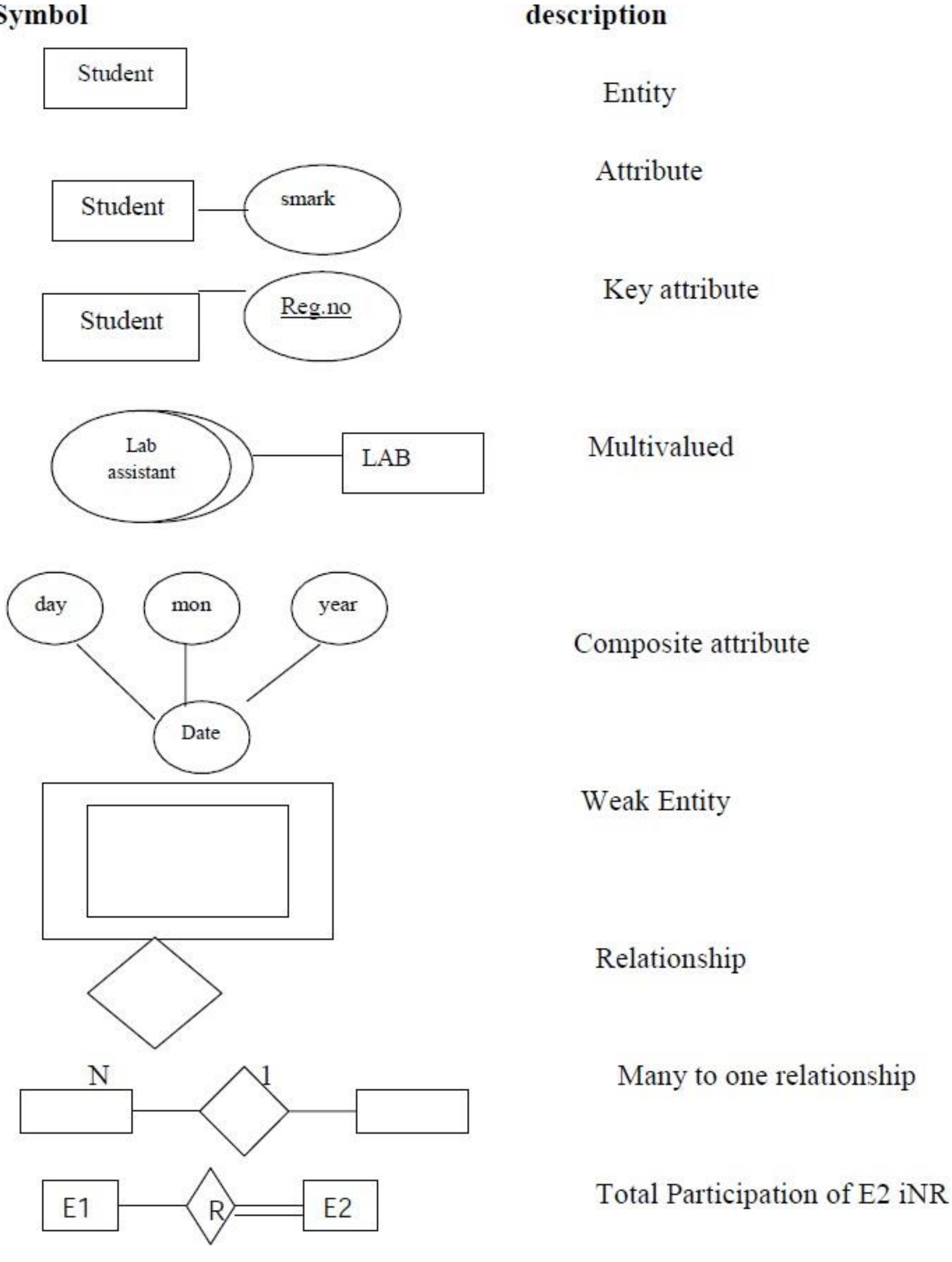


Binary and Entity Relationship

- The entity – relationship diagram focuses solely on data, representing a "data network" that exists for a given system.
- The ERD is especially useful for application in which data and the relationships that given a data are complex.

- Unlike the data flow diagram, data modeling considers data independently of the processing that transforms the data.
- The Object-relationship pair is the cornerstone of the data model.
- These pairs can be represented graphically using the entity relationship diagram (ERD).
- The ERD was originally proposed for the design of relational database system and has been extended by others.
- A set of primary components is identified for the ERD. Data objects, attributes, relationships and various types indicators.
- The primary purpose of the ERD is to represent data objects and their relationships.

Example Diagram Notation


Symbol
description
Student
Entity
Student
smark
Attribute
Student
Reg.no
Key attribute
Lab
assistant
LAB
Multivalued
day
mon
year
Date
Composite attribute
Weak Entity
Relationship
N
1
Many to one relationship
E1
R
E2
Total Participation of E2 iNR

CRC Approach and Naming Class

CRC Cards:

CRC – “Classes, Responsibility and Collaboration”, developed by Cunningham, Wilkmon and Beck.

i) CRC is a technique for identifying Classes, Responsibilities and their methods and attributes.
ii) CRC can help to identify classes.
iii) CRC is more teaching technique than the method identifying the class.
iv) CRC cards are 4”x6” index cards.
v) All information for an object is written on a card.
vi) Cheap, portable, readily available and familiar.

Example:

Classname	
Responsibilities	Collaborators

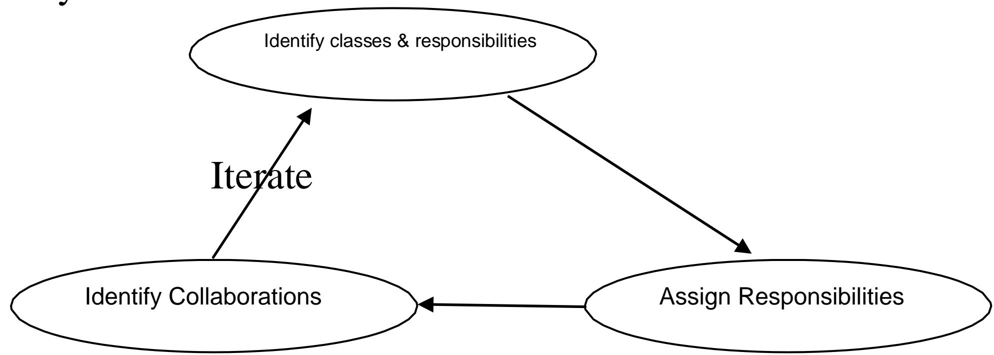
Class name -> left hand corner
Responsibility -> under left hand corner
Collaborations -> right hand side

vii) CRS stresses the importance of creating the objects.

CRC Process:

It consists of three steps.

1. Identify classes and its responsibilities
2. Assign responsibilities
3. Identify collaborators



a. Classes are identified and grouped by common attributes
b. Class names are written on to CRC cards.
c. The card also notes sub and super classes.
d. The responsibilities are distributed.
e. Collaboration is to identify how classes interact.

Example: The vianet Bank ATM system.

Class	Account Balance Number	Checking A/C(sub class) Saving A/C (sub class) Transaction
Responsibilities	Deposit Withdraw Get balance	

In the above atm card class account- is responsible to Bank-client class to keep track of balance, account number and other data of method.

It also provides certain responsibilities such as deposit, withdraw and display balance.

Guidelines for Naming Classes

- The class should describe a single object, so it should be the singular form of noun.
- Use names that the users are comfortable with.
- The name of a class should reflect its intrinsic nature.
- By the convention, the class name must begin with an upper case letter.
- For compound words, capitalize the first letter of each word - for example, Loan Window.

Noun Phrase Approach

Noun:

- Noun is a textual description is conceived to be classes and verbs to be methods of classes.
- All plurals are changed to singular, the nouns are listed and the list is divided into 3 categories.
  i) Relevant classes
  ii) Fuzzy classes
  iii) Irrelevant classes
- Disadvantage

  Assumes that the Requirements Document is complete and correct.

  Identifying tentative classes:

  Guide lines for selecting classes in an application:

  - Look for noun and noun phrases in the use cases.
  - Some classes are implicit.
  - Avoid computer implementation classes.
  - Carefully choose and define class names.

Selecting classes from the relevant and fuzzy categories:

- **Redundant classes:**

i) Don’t keep two classes that express the same information
ii) If more than one word is being used to describe the same idea, select the one that is most meaningful in the system.
iii) Eliminate duplication of classes having some information.
iv) Select any one, which is more meaningful by comparing it in all manners.
v) Choose appropriate vocabulary.

- **Adjective classes:**

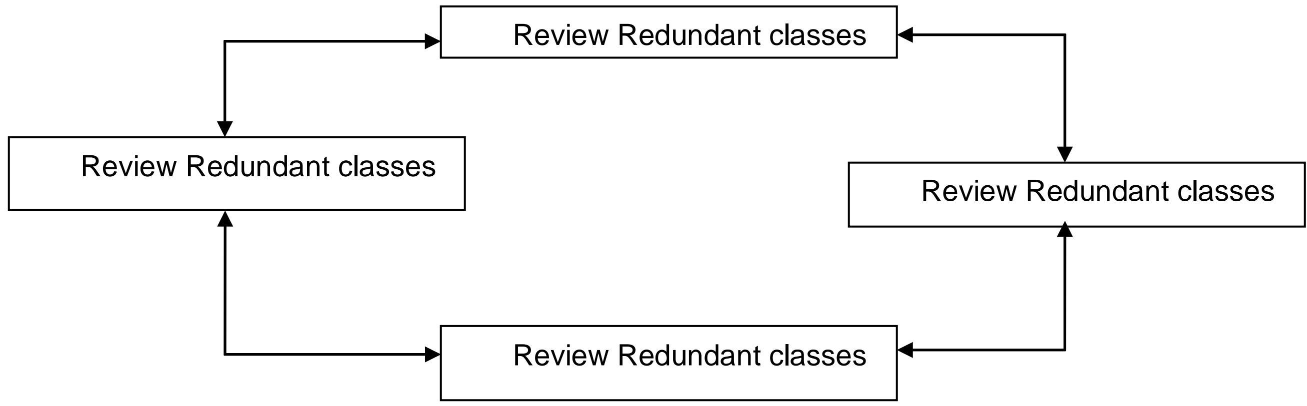
i) An adjective can suggest a different kind of object or different use of the same object.
ii) It is relevant.
i) Choose whether the object represented by the noun behave differently when the adjective is applied to it.

- **Attribute classes:**

i) Restarted as attributes and not as class
ii) Each class must have a purpose and every class should be clearly defined.
iii) Formulate a statement of purpose for each candidate.
i) Identifying relevant classes and eliminating irrelevant classes is an incremental process.
ii) Example: client status and demographic of client are not classes. But attributes of the client class.

- **Irrelevant classes:**

i) Each class must have a purpose and every class should be clearly defined and necessary.



Behavioral Model of UML

Interactions

Information is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.
A message is a specification of a communication between objects that conveys information with the o\expectation that activity will ensure.
An interaction may also be found in the representation of a component, node, or use case, each of which in the UML is really a kind of classifier.

Use cases

A use case is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor.

Graphically a use case is rendered as ellipse.

Use case diagrams

A use case diagram shows a set of use cases and actors and their relationships. Use case diagram commonly contains,

- Use cases
- Actor
- Dependency, generalization and association
- Relationship

Interaction diagrams

Interaction diagrams are used when you want to model the behavior of several objects in a use case.

It contains objects, links, messages, Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case.

Sequence diagrams

Sequence diagrams generally show the sequence of events that occur
A sequence diagram emphasis the some ordering of messages. Sequence diagrams describe interactions among classes in terms of
An exchange of message over time.
Features.
Object lifeline.
Focus control.

Collaboration diagrams

 Collaboration diagrams show the relationship between the objects and other order of messages
 The object is listed as icons and indicate the messages passed between them.
The numbers next to be messages are called sequence number
Features

- Path – indicate how are object is linked to another
- Sequence number =indicate the time order of a message

Activity diagrams
Activity diagrams describe the workflow behavior or system.
The diagrams describe the state are of activities by showing the sequence of activities performed.
Activity diagrams can show activities that are activities that are conditional or Parallel.
Activity diagrams illustrate the dynamic nature of a system o\by modeling the flow of control from activity to activity.
An activity represents an operation on some class in the system that result in a change in state of the system.

Event
An event is the specification of a significant occurrence that has a location in time and space.
Events may be external or internal.
External events are more that parts between that system and its actors.
Internal events are those that pass among objects that the inside the system.

Signals
A signal is a kind of event that represents the specification of an asynchronous stimulus communicated between instances.
A signal represents a named object that is thrown asynchronously by one object and then received by another.

State machines
A state machine is behavior that specifies the sequences of states an object goes through during its lifetime in response, together with its response to those events.

Processes and threads

A process specifies a heavy weight flow that can execute concurrently with other processes.
Thread specifies lightweight flow that can execute concurrently with other threads within the same process.

Time and space
A time mark is a denotation for the time at which an event occurs.
A time expression is an expression that evaluates to an absolute or relative value of time.
A time constraint is a semantic statement about the relative or absolute value of time.
Location is the placement of component on a mode.