OBJECT-ORIENTED SOFTWARE ENGINEERING

UNIT I

Learning Objectives

- **Object Oriented Concepts-** Review of Object and Classes, Links and association, Generalization and specialization, Inheritance and Grouping concepts, Aggregation and Composition, Abstract Classes and Polymorphism, Metadata, Constraints, Reuse.
- **Object Oriented Methodologies-** Introduction to Rational Unified Process, Comparison of traditional life cycle models versus object oriented life cycle models.
- **UML-** Origin of UML, 4+1 view architecture of UML
- **Architecture-** Introduction, system development is model building, architecture, requirements model, analysis model,the design model, the implementation model, test model.

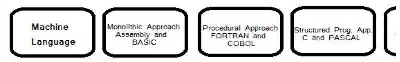
Evolution of Object Orientation

- The idea of object-oriented programming gained **momentum** in the 1970s and in the early 1980s.
- **Bjorn Stroustrup** integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially.
- In the early 1990s a group at Sun led by **James Gosling** developed a simpler version of C++ called Java that was meant to be a programming language for video-on-demand applications.
- This project was going nowhere until the group **re-oriented** its focus and marketed Java as a language for programming Internet applications.
- The language has gained **widespread popularity** as the Internet has boomed, although its market penetration has been limited by its inefficiency.

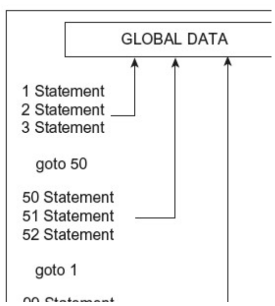
Evolution of Object Orientation

1. **Monolithic Programming Approach:** In this approach, the program consists of **sequence of statements** that **modify data**.

- All the **statements** of the program are **Global** throughout the whole program. The **program control** is achieved through the use of **jumps** i.e. **goto statements**.
- In this approach, **code is duplicated** each time because there is no support for the function. **Data** is **not fully protected** as it can be accessed from any portion of the program.
- So this approach is useful for designing **small** and **simple** programs. The programming languages like ASSEMBLY and BASIC follow this approach.



Evolution of Object Orientation



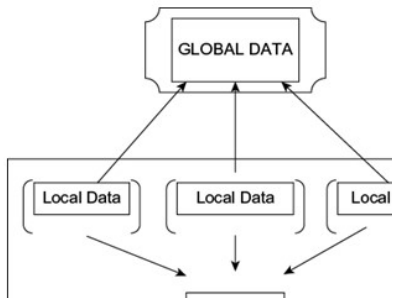
Evolution of Object Orientation

2. **Procedural Programming Approach:** This approach is **top down approach**. In this approach, a program is divided into **functions** that perform a **specific task**.

- This approach **avoids repetition of code** which is the main drawback of **Monolithic Approach**.
- The basic **drawback** of Procedural Programming Approach is that data is not secured because data is **global** and can be accessed by any function.
- This approach is mainly used for medium sized applications. The programming languages: **FORTRAN and COBOL** follow this approach.

•3. **Structured Programming Approach:** The basic principal of **structured programming approach** is to divide a program in functions and modules.

Evolution of Object Orientation



Evolution of Object Orientation

- The use of modules and functions makes the program more **comprehensible** (understandable). It helps to write **cleaner code** and helps to **maintain control** over each function. This approach gives importance to functions rather than data.
- It focuses on the development of large software applications. The programming languages: **PASCAL and C** follow this approach.

4. Object Oriented Programming Approach: The basic principal of the OOP approach is to **combine** both **data** and **functions** so that both can operate into a **single unit**. Such a unit is called an **Object**.

- This approach **secures data** also. Now a days this approach is used mostly in applications. The programming languages: **C++ and JAVA** follow this approach. Using this approach we can write any lengthy code.

Object Orientation Paradigm

- An approach to the solution of problems in which all **computations** are performed in context of objects.
- The objects are instances of **programming constructs**, normally called as **classes** which are **data abstractions** with **procedural abstractions** that operate on objects.
- A software system is a set of mechanism for performing certain **action** on certain data

Algorithm + Data structure = Program

- **Data Abstraction + Procedural Abstraction**

Object Orientation

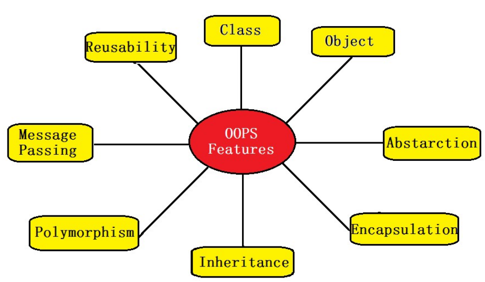
- **Object orientation** refers to a special type of programming paradigm that combines **data structures** with **functions** to create **re-usable objects.**
- The object-oriented (OO) paradigm is a **development strategy** based on the concept that systems should be **built** from a **collection of reusable components** called objects.
- Instead of separating data and functionality as is done in the structured paradigm, objects **encompass both**.
- **Why object orientation?**
 To create sets of **objects** that work together concurrently to produce s/w that better, model their problem domain that similarly system produced by traditional techniques.

Object Orientation Adaptation

Object orientation adapts to the following criteria's-

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
6. Higher level of abstraction
7. Encouragement of good programming techniques
8. Promotion of reusability

Object Orientated Features



Object Orientated Features

Object orientation adapts to the following criteria's-

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
6. Higher level of abstraction
7. Encouragement of good programming techniques
8. Promotion of reusability

Object Orientated Features

1. **OBJECT** - Object is a collection of number of **entities**. Objects take up space in the memory. Objects are **instances of classes**. When a program is executed , the objects interact by sending messages to one another. Each object contain **data** and **code** to manipulate the data. Objects can interact without having know details of each others data or code. **Each instance** of an object can hold its own **relevant data**.

2. **CLASS** - Class is a **collection** of **objects** of similar type. Objects are **variables** of the **type class**. Once a class has been defined, we can create any number of objects belonging to that class. Classes are **user define data types**. A class is a **blueprint** for any functional entity which defines its **properties** and its **functions**.

Object Orientated Features

3. DATA ENCAPSULATION – Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**. Class variables are used for storing data and functions to specify various operations that can be performed on data. This process of **wrapping up** of data and functions that operate on data as a single unit is called as data encapsulation. Data is **not accessible** from the outside world and only those function which are present in the class can access the data.

4. DATA ABSTRACTION- Abstraction (from the Latinn *abs* means *away from* and *trahere* means to draw) is the **process** of taking away or **removing characteristics** from something in order to reduce it to a **set of essential characteristics.** Advantage of data abstraction is **security**.

Object Orientated Features

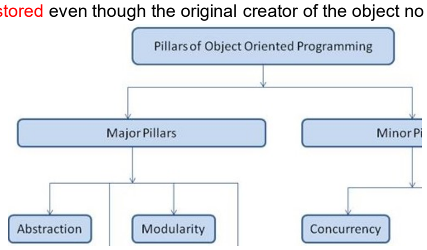
5. INHERITANCE- It is the process by which object of one class **acquire** the **properties** or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add **additional features** to an existing class **without modifying it.** This is possible by driving a new class from the existing one. **Advantage** of inheritance is **reusability** of the **code**.

6. MESSAGE PASSING - The process by which **one object** can interact with **other object** is called **message passing**.

7. **POLYMORPHISM** - A greek term means **ability to take more than one form.** An operation may exhibit different behaviours in different instances. The behaviour depends upon the **types of data** used in the operation.

Object Orientated Features

8. PERSISTENCE - The process that allows the state of an **object** to be saved to **non-volatile storage** such as a file or a database and later restored even though the original creator of the object no longer exists.



Benefits of OOPs

- **Code Reuse and Recycling**:
  Objects created for Object Oriented Programs can easily be reused in other programs.The code and designs in object-oriented software development are **reusable** because they are modeled directly out of the real-world problem-domain.

- **Design Benefits**:
  Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for **better designs** with **less flaws**.

- **Ease out development**: In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-Object Oriented ones.

Benefits of OOPs

- **Object orientation works at a higher level of abstraction**

  One of our most powerful techniques is the form of selective amnesia called 'Abstraction'. Abstraction allows us to ignore the details of a problem and concentrate on the whole picture.

- **Software life cycle requires no vaulting**

  The object-oriented approach uses essentially the same language to talk about analysis, design, programming and (if using an Object-oriented DBMS) database design. This **streamlines** the entire software development process, reduces the level of **complexity** and **redundancy**, and makes for a cleaner system architecture and design.

Benefits of OOPs

- **Data is more stable than functions**

  Functions are not the most stable part of a system, the data is. Over a period of time, the **requirements** of a system undergo radical change. New uses and needs for the software are discovered; new features are added and old features are removed. During the course of all this change, the **underlying heart- data** of the system remains comparatively constant.

- **Software Maintenance:**

  Legacy code must be dealt with on a daily basis, either to be improved upon or made to work with newer computers and software. An Object Oriented Program is much easier to **modify** and **maintain**. So although a lot of work is spent before the program is written, less work is needed to maintain it over time.

Application Areas of OOPS

- Real time systems.
- Simulation & Modelling.
- Object-oriented database system.
- Object-oriented Operating System.
- Graphical User Interface.
- Window based O.S. design.
- Multimedia Design.
- CIM/CAD/CAM Systems.
- Computer based Training & Education System.
- AI and Expert System.
- Neural Networks and parallel programming.
- Decision support and office automation system.

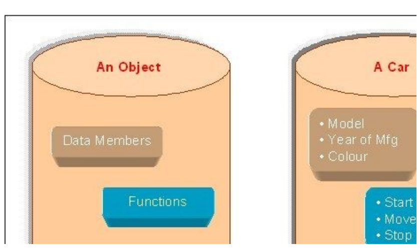
Object- *The CRUX of the matter!!*

- “An object is an **entity** which has a **state** and a defined set of **operations** which **operate** on that state.”
- The **state** is represented as a set of object attributes. The operations associated with the object **provide services** to other objects (clients) which request these services when some computation is required
- Objects are **created** according to some **object class definition**. An object class definition serves as a **template** for objects. It includes **declarations** of all the attributes and services which should be associated with an object of that class.
- An Object is anything, **real** or **abstract**, about which we store data and those **methods** that manipulate the data.
- An **object** is a component of a program that knows how to perform certain actions and how to **interact** with other elements of the program.

Object- *The CRUX of the matter!!*

- Each **object** is an instance of a particular **class** or **subclass** with the class's own methods or procedures and data variables. An object is what **actually runs** in the computer.
- Objects are the basic run time entities in an **object oriented system**.
- They match closely with **real time objects**.
- Objects take up **space in memory** and have an associated **address** like a Record in Pascal and a Structure in C.
- Objects interact by **sending Message** to one other. E.g. If “Customer” and “Account” are two objects in a program then the customer object may send a message to the account object requesting for bank balance without divulging the details of each other’s data or code.
- Code in object-oriented programming is organized around objects.

Object- A representation



Object- *Key Goals!!*

Goals of Object definition-

- Define Objects and classes
- Describe objects' methods, attributes and how objects respond to **messages**
- Define Polymorphism, Inheritance, data abstraction, encapsulation, and protocol
- Describe **objects relationships**
- Describe object persistence
- Understand meta-classes.

Object- *An Example*

Example:

Attributes
- I am a Car.
- I know my color,
- manufacturer, cost,
- owner and model.

It does things (methods)
- I know how to
- compute
- my payroll.

Attributes or **properties** describe object's state (data) and **methods** define its **behaviour**.

Object- Attributes and Methods

Object's 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.

Object's 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, carry passenger in it.

Objects- blueprints of classes
- The role of a class is to define the **state** and **behavior** of its instances.
- The class car, for example, defines the property color.
- Each individual car will have property such as "maroon," "yellow"

Classes – *The Blueprint !!*

- *A **class** is a blueprint of an object.*
- A class is a **group of objects** that share **common properties** & **behavior/ relationships**.
- In fact, **objects** are the variables of the **type class**.
- Classes are **user defined data types** and behaves like the built-in types of a programming language.
- **Class** are a **concept**, and the **object** is the embodiment of that **concept**.
- Each class should be designed and programmed to accomplish one, and only one, thing, in accordance to **single responsibility principle** of SOLID design principles.
- In the OOPs concept the variables declared inside a class are known as "**Data Members**" and the functions are known as "**Member Functions**"

Class Members

- A class has different members, and developers in Microsoft suggest to program them in the following order:
- **Namespace**: The namespace is a keyword that defines a **distinctive name** or last name for the class. A namespace categorizes and organizes the library (assembly) where the class belongs and avoids **collisions** with classes that share the same name.
- **Class declaration**: Line of code where the class name and type are **defined**.
- **Fields**: Set of **variables** declared in a class block.
- **Constants**: Set of constants declared in a **class block**.
- **Constructors**: A method or group of methods that contains code to **initialize** the class.

Class Members

- **Properties**: The set of **descriptive data** of an object.
- **Events**: Program **responses** that get fired after a user or application action.
- **Methods**: Set of **functions** of the class.
- **Destructor**: A method that is called when the class is **destroyed**. In managed code, the Garbage Collector is in charge of destroying objects; however, in some cases developers need to take extra actions when objects are being released, such as freeing handles or deallocating unmanaged objects.

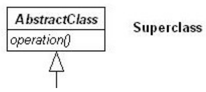
Classes – A Classification

A **Class** "is a set of objects that share a common s
a common behavior." [Booch 1994].

Abstract Classes cannot be instantiated directly.

– The main purpose of an abstract class is to define a comı
for its subclasses.

Concrete Classes are not abstract and can have iı

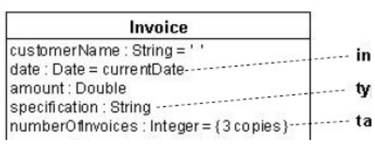


Attributes

An **Attribute** is a named data element within
describes the values that instances of the class

– Attributes show the states of an objects with attribuı

– Example: an Invoice class



Data Abstraction

- **General**: **Focus on the meaning**
  Suppress irrelevant "implementation" details
- It refers to the act of **representing essential features** without including the **background details** or **explanations**.
- Through the process of abstraction, a programmer hides all but the **relevant data** about an object in order to reduce complexity and increase efficiency.
- Abstraction tries to **minimize details** so that the programmer can focus on a **few concepts** at a time. This programming technique separates the **interface** and **implementation**.
- Once you have **modelled** your **object** using Abstraction , the same set of data could be used in **different applications**.

Data Abstraction- *The Motivation*

- **Client/user perspective (Representation Independence)**
  - Interested in **what** a program does, not **how**.
  - Minimize irrelevant details for clarity.
- **Server/implementer perspective (Information Hiding)**
  - Restrict users from making **unwarranted assumptions** about the implementation.
  - Reserve right to **change representation** to improve performance, … (maintaining behavior).

Data Abstraction- *Advantageous!!*

Advantages Of Abstraction

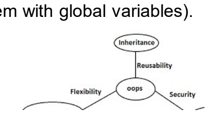
- The programmer does not have to write the **low-level code**.
- The programmer does not have to specify all the **register/binary-level steps** or care about the hardware or **instruction set details**.
- **Code duplication** is avoided and thus programmer does not have to repeat fairly common tasks every time a similar operation is to be performed.
- It allows **internal implementation details** to be changed without affecting the **users** of the abstraction.

Data Encapsulation

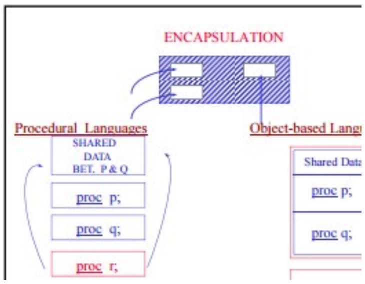
- The **wrapping up** of **data** & **functions** (that operate on the data) into a **single unit** (called class) is known as **ENCAPSULATION**.
- Encapsulation is the **mechanism** that **binds together code** and the **data** it manipulates and keeps both safe from outside interference and misuse.
- Enables **enforcing data abstraction**

Conventions are no substitute for enforced constraints.

- Enables **mechanical detection** of **typos** that manifest as "illegal" accesses. (Cf. problem with global variables).



Data Encapsulation- *An Insight!*



Inheritance

- Inheritance allows the reusability of an **existing operations** and **extending the basic unit** of a class without creating from the scratch.
- Inheritance is the **capability** of one class of things **to inherent properties** from other class.
- Supports the concept of **Hierarchical classification.**
- Ensures the **closeness** with **real world models**.
- Provides **Multiple Access Specifiers** across the **modules** (Public, Private & Protected)
- Supports **Reusability** that allows the addition of **extra features** to an existing class **without modifying it**.

Inheritance : *Sub-classing*

- **Code reuse**

derive Colored-Window from Window (also adds fields/methods)

- **Specialization: Customization**

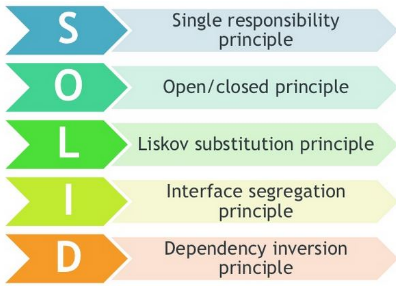
derive bounded-stack from stack (by overriding/redefining push)

- **Generalization: Factoring**

Commonality – code sharing to minimize duplication – update consistency

- Using two concepts of inheritance, **subclassing** (making a new class based on a previous one) and **overriding** (changing how a previous class works), you can organize your objects into a **hierarchy**.

SOLID Design Principles



SOLID Design Principles

S- A class should have one and only one reason to change, meaning that a class should have only one job.

O- Objects or entities should be open for extension, but closed for modification.

L- All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

I- A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

D- Entities must depend on abstractions not on concretions. The high level module must not depend on the low level module, but they should depend on abstractions.

Open-Closed Principle

- **Open-closed principle**
  - A class is closed because it can be **compiled**, stored in a **library**, and made available for use by its clients.
- **Stability**
  - A class is open because it can be **extended** by adding **new features (operations/fields)**, or by **redefining inherited features**.

- Inheritance allows the developers for reusing the available code
- A **subclass** can be treated as if it is a **super** class
- Objects of both super class and subclass can be created in the applications
- A class an be extended in which the **additional** and exclusive functionality can be placed without altering the super class
- Relationships among objects can easily be established

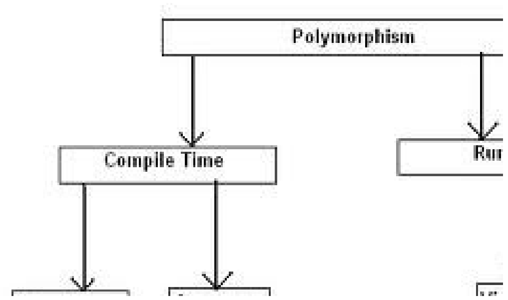
Polymorphism

- In object oriented programming, *polymorphism* refers to a programming language's **ability** to **process objects differently** depending on their data types or class.
- Polymorphism is the **quality** that allows one name to be used for two or more related but **technically different purposes**. In the following, each **graphical object** has the same services, although they are implemented differently.
- If you think about the Greek roots of the term, Polymorphism is the **ability** (in programming) to present the same interface for differing underlying forms (data types).
- For example, **integers** and **floats** are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different. They're rarely considered as objects in the usual term.

Polymorphism

- In object oriented programming, *polymorphism* refers to a programming language's **ability** to **process objects differently** depending on their data types or class.
- Polymorphism is the **quality** that allows one name to be used for two or more related but **technically different purposes**. In the following, each **graphical object** has the same services, although they are implemented differently.
- If you think about the Greek roots of the term, Polymorphism is the **ability** (in programming) to present the same interface for differing underlying forms (data types).
- For example, **integers** and **floats** are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different. They're rarely considered as objects in the usual term.

Polymorphism- *It's Variants!!*



Unified Object Modelling

- The UML effort started officially in October 1994, when Rumbaugh joined **Booch** at **Rational**.
- The Unified Modeling Language (UML) is a **standard language** for writing software blueprints. The UML may be used to **visualize**, **specify**, **construct**, and **document** the artifacts of a software intensive system.
- The UML is appropriate for **modelling systems** ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems.
- The UML is process **independent**, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.

Paradigm Shift

Structured Paradigm	Object-OrientedParad
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2′. Object-oriented analys
3. Design phase	3′. Object-oriented design
4. Implementation phase	4′. Object-oriented progra
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase

Traditional paradigm:

- Jolt between **analysis** (what) and **design** (how)

Object-oriented paradigm:

- Objects enter from **very beginning**

Paradigm Shift

Structured Paradigm	Object-Oriented Paradigm
2. Specification (analysis) phase • Determine what the product is to do	2′. Object-oriented analysis phase • Determine what the product is to do • Extract the objects
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design phase • Detailed design
4. Implementation phase • Implement in appropriate programming language	4′. Object-oriented programming phase • Implement in appropriate object-oriented programminglanguage

- Objects enter here

Analysis/Design Analogue

System analysis

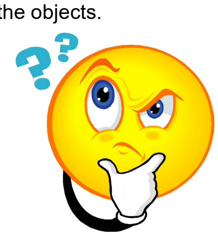
- Determine what has to be done
- Determine the **objects**

Design

- Determine how to do it
- Design the **objects**
- **Detailed design**—design each module

Object Oriented Thinking

- Identify all objects in this classroom and articulate their object diagrams. Specify each objects attributes and behaviors through this diagram. Correspondingly identify relationships between the objects.



Benefits of OO Thinking

- Ease to develop complex systems
- Systems are prone to change
- Systems with user interfaces
- Systems that are based on client/servermodel
- To build e-commerce/web based applications
- For enterprise application integration
- Improved quality, reusability, extensibility
- Reduce maintenance burden
- Financial benefits

Challenges in OO Thinking

- Mind-set transition
- Investment in training and tools
- Insist on testing
- More time and cost to analysis and design
- User involvement
- Provides only long term benefits
- Still the success is greatly depends on people involved

Links and Associations

- **Links** and **association** are the means for building the **relationship** among the **objects** and **classes**.
- Links and association , both are quite same feature but **links** establishing among the **objects** (instance) and **association** establishing among the **class**.

"Link is related to objects whereas association is related to classes"

- **Class diagrams** contain **associations**, and **object diagrams** contain **links**.
- Both associations and links represent **relationships**.
- Links as well as associations appear as **verbs** in a problem statement.

Links and Associations

- ***Can link and Association applied interchangeably?***
- No, You cannot apply the link and Association interchangeably.
- Since **link** is used represent the **relationship** between the **two objects.**
- But **Association** is used represent the **relationship** between the **two classes.**

Link ::	student:Abhilash	course:MCA
Association::	student	course

Links

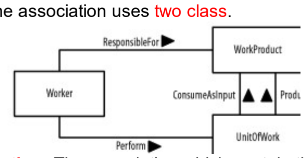
- In **object modelling** links provides a **relationship** between the **objects**.
- These objects or instance may be same or different in **data structure** and **behaviour**.
- Therefore a link is a **physical or conceptual connection** between instance (or **objects**).
- For **example**: Ram works for HCL company. In this example "**works for**" is the link between "Ram" and "HCL company". Links are relationship among the objects(instance).
- **Types of links**:
  1.One to one links
  2.one to many and many to one links
  3. many to many

Associations

- The object modelling describes as a **group of links** with **common structure** and **common semantics**.
- **"*Association*** *is a* ***relationship*** *between* ***classifiers*** *which is used to show that instances of classifiers could be either* ***linked*** *to each other or combined* ***logically*** *or* ***physically*** *into* *some aggregation*."
- All the links among the object are the forms of **association** among the **same classes**.
- The association is the **relationship** among **classes**.
- UML specification categorizes association as **semantic relationship**. Some other UML sources also categorize association as a **structural relationship**. Wikipedia states that association is **instance level** relationship and that associations can only be shown on **class diagrams**.

Degree of Association

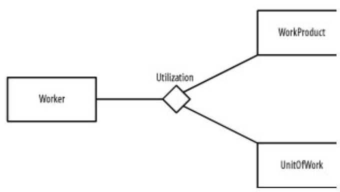
- **Unary Association**: the association can be defined on a **single class**. This type of association called unary (or **singular**) **association**.
- **Binary Association**: The binary association contain the **degree of two classes**. The association uses two class.



- **Ternary Association**: The association which contain the **degree of three classes** is called **ternary association**. The ternary Association is an atomic unit and cannot be subdivided into binary association without losing information.

Degree of Association

- **Quaternary Association**: The Quaternary Association exists when there are **four classes associated**.
- **Higher degree Association**: The higher order association are more **complicated** to draw , implement because when **more than four class** need to be associated then it seems a hard task.



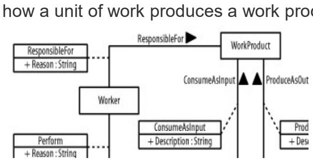
Association Classes

- Association classes may be applied to both **binary** and **n-ary associations**.
- Similar to how a class defines the characteristics of its objects, including their **structural features** and **behavioural features**, an *association class* may be used to define the **characteristics** of its **links**, including their **structural features** and behavioural features. These types of classes are used when you need to maintain information about the **relationship itself**.
- In a UML class diagram, an association class is shown as a class attached by a **dashed-line path** to its association path in a binary association or to its **association diamond** in an n-ary association.
- The **name** of the **association class** must match the **name** of the **association**.

Binary Association Classes

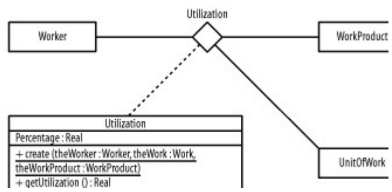
The following example shows **association classes** for the binary associations in the most basic notation for binary association classes.

The association classes track the following information:
•The reason a worker is responsible for a work product
•The reason a worker performs a unit of work
•A description of how a unit of work consumes a work product
•A description of how a unit of work produces a work product.



n-ary Association Classes

- The following example shows an association class for the n-ary association in the most basic notation for n-ary association classes.
- The association class tracks a **utilization percentage** for workers, their units of work, and their associated work products.



Association Ends

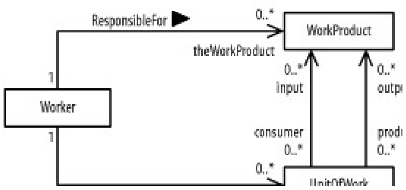
- An ***association end*** is an **endpoint** of the line drawn for an association, and it connects the **association** to a **class**.
- An association end may include any of the following **items** to express more **detail** about how the class relates to the other class or classes in the association:
  - ✓Role name
  - ✓Navigation arrow
  - ✓Multiplicity specification
  - ✓Aggregation or composition symbol
  - ✓Qualifier

I. Rolenames

- A *rolename* is optional and indicates the **role** a **class** plays **relative** to the **other classes** in an association, how the other classes "*see*" the class or what "*face*" the class projects to the other classes in the relationship.
- A rolename is shown near the end of an association attached to a class.
- For example, a work product is seen as input by a unit of work where the unit of work is seen as a consumer by the work product; a work product is seen as output by a unit of work where the unit of work is seen as a producer by the work product, as shown in the figure

I. Rolenames

Binary association ends



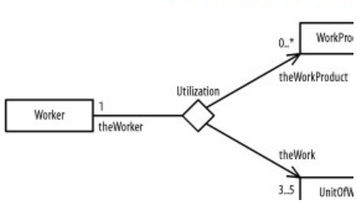
II. Navigation

- *Navigation* is **optional** and indicates *whether a class may be referenced from the other classes in an association.*
- Navigation is shown as an **arrow** attached to an association end **pointing** toward the class in question.
- If no arrows are present, associations are assumed to be **navigable** in all directions, and all classes involved in the association may reference one another.
- For example, given a worker, you can determine his work products and units of work. Thus, arrows pointing towards work product and units of work. Given a unit of work, you can determine its input and output work products

II. Navigation

- Given a worker, you can reference his work products and units of work to determine his utilization, but given a work product or unit of work, you are unable to determine its utilization by a worker.

N-ary association ends



III. Multiplicity

- *Multiplicity* (which is optional) indicates **how many objects** of a **class** may relate to the **other classes** in an association. Multiplicity is shown as a comma-separated sequence of the following:
  - Integer intervals
  - Literal integer values
- **Intervals** are shown as a *lower-bound .. upper-bound* string in which a **single asterisk** indicates an **unlimited range**. No asterisks indicate a closed range.
- For example- 1 means one, 1..5 means one to five, 1, 4 means one or four, 0..* and * mean zero or more (or many), and 0..1 and 0, 1 mean zero or one.
- There is no default multiplicity for association ends. Multiplicity is simply undefined, unless you specify it.

 U1.67

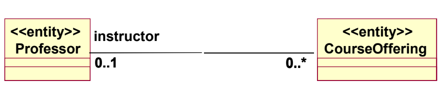
III. Multiplicity Indicators

Unspecified	
Exactly one	1
Zero or more (many, unlimited)	0..*
One or more	1..*
Zero or one (optional scalar role)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6

 U1.68

III. Multiplicity

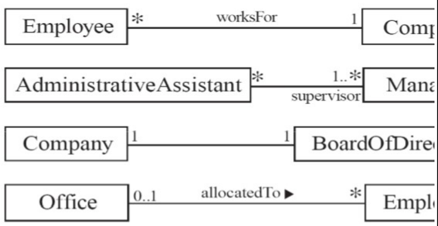
- Multiplicity is the **number of instances** of one class **relates to instance of another class.**
- For the following association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course Offering, there may be either one or zero Professor as the instructor.



 U1.69

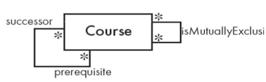
Labelling Associations

- Each association can be labelled, to make nature of the association

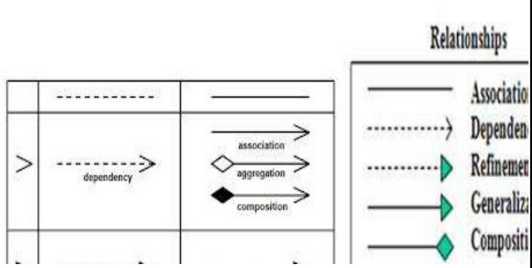


Reflexive Associations

- It is possible for an **association** to **connect a class to itself.**
- There are two main types:
  - **Symmetric** and **Asymmetric**.
- **Asymmetric Reflexive Associations:** The ends of the association are semantically different from each other, even though the associated class is the same. Examples include parent-child, supervisor-subordinate and predecessor-successor.
- **Symmetric Reflexive Associations:** There is **no logical difference** in the semantics of each association end. In other words, students who have taken one course cannot take another in the set. Umple uses the keyword **'self'** to identify this case.

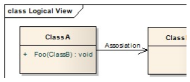


Association Nomenclature

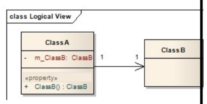


Association- *Further more!*

- The most abstract way to describe static relationship between classes is using the **Association** link, which simply states that there is ***some kind of a link or a dependency*** between two classes or more.
- **Weak Association** - ClassA may be linked to ClassB in order to show that one of its methods includes **parameter** of ClassB instance, or returns instance of ClassB.



- **Strong Association** - ClassA may also be linked to ClassB in order to show that it holds a **reference** to ClassB instance.

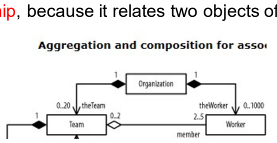


Association- *Further more!*

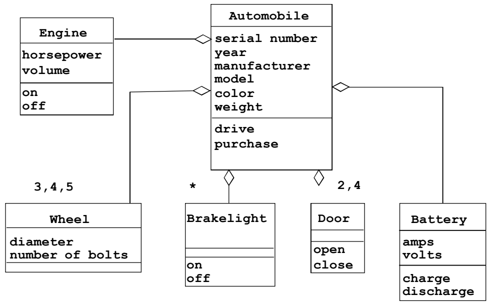
- In **Object-oriented programming**, one object is related to other to use functionality and service provided by that object.
- This relationship between two objects is known as the *association* in object oriented general software design and depicted by an arrow in Unified Modelling language or UML.
- Both **Composition** and **Aggregation** are the form of **association** between two objects, but there is a **subtle difference between composition and aggregation**, which is also reflected by their UML notation.
- *The composition is stronger than Aggregation.*
- In Short, a **relationship** between two objects is referred as an **association**, and an **association** is known as **composition** when ***one object owns other*** while an **association** is known as **aggregation** when *one object uses another object*.

Aggregation *(Shared Association)*

- Aggregation is **whole-part relationship** between an **aggregate**, the whole, and its **parts** where the part can exist **independently** from the aggregate.
- This relationship is often known as a *has-a relationship*, because the **whole has its parts**.
- Aggregation is shown using a **hollow diamond** attached to the class that represents the **whole**.
- Creating a **circular relationship** to allow for sub-teams is known as a reflexive relationship, because it relates two objects of the same class.



Aggregation Scenario



Aggregation- *When to use!*

- As a general rule, you can mark an association as an aggregation if the following are true:

□ You can state that

  □ *The parts 'are part of' the aggregate*
  □ *The aggregate 'is composed of' the parts*

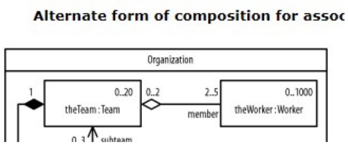
□ When something owns or controls the aggregate, then they also own or control the **parts**.

Composition *(Non-shared)*

- Composition, also known as **composite aggregation**, is a **whole-part relationship** between a **composite** (the whole) and its **parts**, in which the parts must **belong only to one whole** and the whole is responsible for **creating** and **destroying** its parts when it is created or destroyed.
- This relationship is often known as a *contains-a relationship*, because the **whole contains its parts**.
- Composition is shown using a **filled diamond** attached to the class that represents the whole.
- For example, an organization contains teams and workers, and if the organization ceases to exist, its teams and workers also cease to exist.
- Composition also may be shown by **graphically nesting classes**, in which a nested class's multiplicity is shown in its upper-right corner and its **rolename** is indicated in front of its class name.

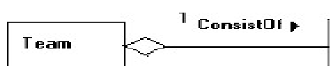
Composition

- Separate the **rolename** from the **class** name using a **colon**.
- A composition indicates a **strong ownership** and **coincident lifetime** of parts by the whole (i.e., *they live and die as a whole*).



Aggregation vs. Composition

Example: Aggregation



Aggregation vs. Containment

- **Aggregation** is the relationship between the whole and a part. We can add/subtract some properties in the part (slave) side. It won't affect the whole part.
- Best **example** is Car, which contains the wheels and some extra parts. Even though the parts are not there we can call it as car.
- But, in the case of **containment** the whole part is **affected** when the part within that got affected.
- The **human body** is an apt example for this relationship. When the whole body dies the parts (heart etc.) are dead.

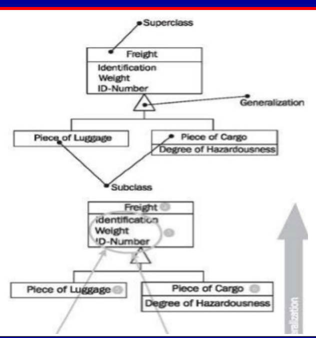
System Complexity Measure

- **System complexity** can be measured simply by looking at a UML class diagram and **evaluating the association**, **aggregation**, and **composition relationship lines**.
- The way to measure complexity is to determine *how many classes can be affected by changing a particular class*.
- If class *A* exposes class *B*, then any given class that uses class *A* can theoretically be affected by changes to class *B*.
- The **sum** of the number of **potentially affected classes** for every class in the system is the total system complexity.

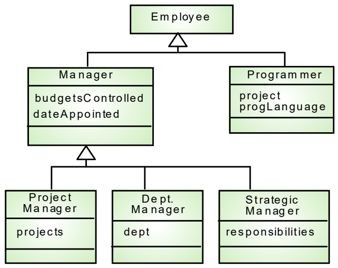
Generalization

- *Generalization is the process of extracting shared characteristics from two or more classes*, and **combining** them into a **generalized superclass**.
- Shared characteristics can be **attributes**, **associations**, or **methods**.
- Generalization is a process of **defining** a super class from a given *set of semantically related entity set*.
- Generalization uses a ***"is-a" relationship*** from a **specialization** to the **generalization class**.
- Common **structure** and **behaviour** are used from the specialization to the generalized class.

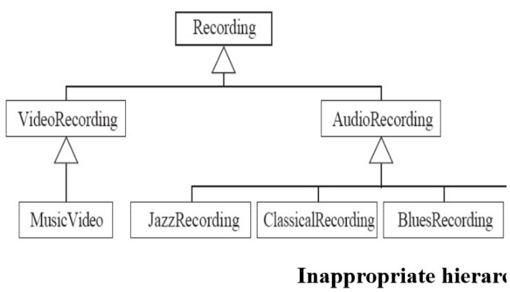
Generalization Example



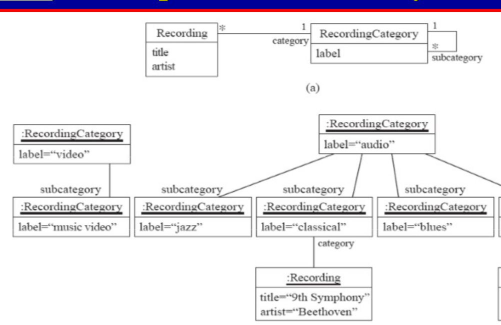
Generalisation Hierarchy



Avoid unnecessary Generalization



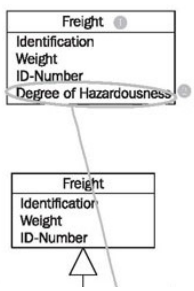
Improvised Hierarchy



Specialization

- *Specialization* means creating **new subclasses** from an **existing class**.
- If it turns out that certain **attributes**, **associations**, or **methods** only apply to some of the objects of the class, a *subclass* can be created.
- The most **inclusive class** in a generalization/specialization is called the **superclass** and is generally located at the **top** of the diagram.
- The more **specific classes** are called **subclasses** and are generally placed **below** the superclass.

Specialization Example

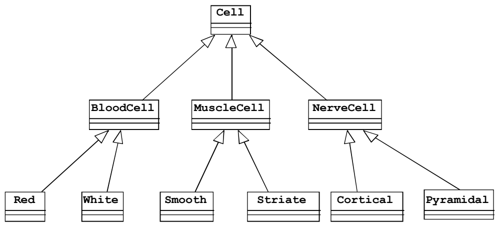


Inheritance

- The **generalization/specialization** relationship is implemented in object oriented programming languages through **inheritance**.
- Object-oriented programming allows classes to ***inherit*** commonly used **state** and **behaviour** from other classes.
- A class that is **derived** from another class is called a ***subclass*** (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a ***superclass*** (also a *base class* or a *parent class*).
- When you want to create a new class and there is already a class that includes some of the code that you want, you derive it.
- A subclass inherits all the ***members*** (fields, methods, and nested classes) from its superclass.
- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

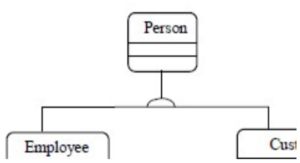
Inheritance

- Models "kind of" hierarchy
- Powerful notation for sharing similarities among classes while preserving their differences
- UML Notation: An **arrow with a triangle**



Inheritance Hierarchy

- The **generalized** class at the top and the **specialized** classes below.
- The specialized class names should reflect the class they were **specialized from**.
- For example, employee was specialized from Person.



Aggregation vs Inheritance

- Both associations describe trees (hierarchies)
  - **Aggregation** tree describes a-part-of relationships (also called **and-relationship, Has –a Relationship, containership**)
  - **Inheritance** tree describes "**kind-of**" relationships (also called **or-relationship, is-a relationship**)
- Aggregation **relates instances** (involves two or more *different* **objects**)
- **Inheritance relates classes** (a way to structure the description of a *single* object)

Realization

- 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.

Dependency

- Change in **structure** or **behaviour** 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 this happens.

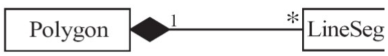
- ***Example:*** Relationship between shape and circle is dependency

Dependency

- It is the relationship between **dependent** and **independent classes**.
- Any change in the independent class will *affect the states* of the dependent class.
- A dependency is a relation between two classes in which a change in ***one may force changes*** in the other although there is no explicit association between them.
- A stereotype may be used to denote the type of the dependency.
  - Indicates a semantic relationship between **two (or more) classes**
  - It indicates a **situation** in which a change to the target element may require a change to the source element in the dependency
  - A dependency is shown as a **dashed arrow** between two model elements

Propagation

- A mechanism where an **operation** in an **aggregate** is implemented by having the aggregate perform the operation on its **parts**.
- At the same time, **properties** of the **parts** are often **propagated** back to the **aggregate**.
- *Propagation is to aggregation as inheritance is to generalization.*
- The major difference is-
  - Inheritance is an implicit mechanism
  - Propagation has to be programmed when required.



Constraints

- To **restrict** ways in which a class can **operate** we add constraints.
- **OCL** is a **specification language** designed to formally specify constraints in software modules.
- Types of constraints
  - Invariant
    - ✓ Must be always true
    - ✓ Defined on class attributes
  - Pre-condition
    - ✓ Defined on a method
    - ✓ Checked before execution
    - ✓ Frequently used to validate input parameter
  - Post-condition
    - ✓ Defined on a method
    - ✓ Checked after method execution
    - ✓ Frequently used to describe how values were changed by method

Constraints

- Constraint defines some **functional relationship** between **entities** of an **object**.
- The term **entity** includes objects , classes , attributes , links and association.
- It mean constraints can be implemented on the **objects**, **classes** , **attributes** , **links** as well as on **association** too.

Think about it!

- For the following cases, indicate whether the relationship should be an ordinary association, a standard aggregation, a composition, a dependency, a Realization. Justify your answer.
  - a. Student taught by teacher
  - b. Department has Teachers
  - c. House and Rooms
  - d. Person and electric switch (to start a fan).
  - e. Code snippet

    import B;

    public class A { public void method1(B b) { // . . . }..}
  - f. Code snippet

    import B3;

    public class A3 implements B3 {   // . . .}

Suggested Sequence

- Identify a first set of **candidate classes**
- Add **associations** and **attributes**
- Find **generalizations**
- Find **specializations**
- List the **main responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the **entire process** until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
- Don't be too **disorganized**.
- Don't be too **rigid** either.

Object Oriented Lifecycle Model

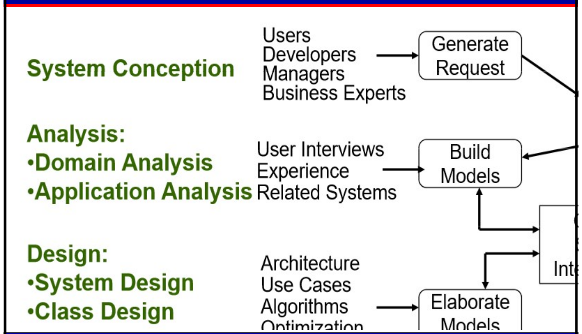
- Object Oriented Methodology (OOM) is a **system development approach** encouraging and facilitating **re-use** of **software components**.
- The object-oriented systems analysis and design methodology classification emerged in the **mid- to late 1980s** as businesses began to seriously consider object-oriented-programming languages for developing and implementing systems.
- The **Object Oriented Methodology** of Building Systems takes the **objects** as the basis.
- For this, first the **system** to be developed is observed and **analyzed** and the **requirements** are defined as in any other method of system development.
- Once this is done, the **objects** in the required system are identified. For **example** in case of a Banking System, a customer is an object, and even an account is an object.

OOLCM - *Representation*

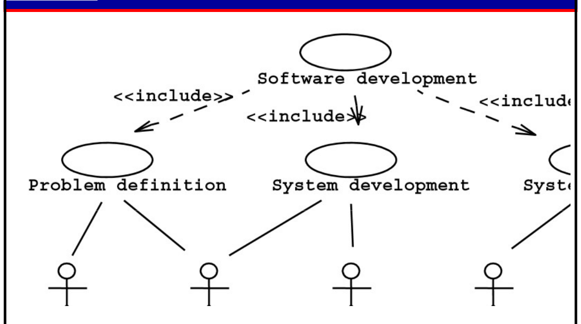
-----------TECHNIQUES & TOOLS REPRESEN

	System Flows	Data	Process Logic
TRADITIONAL	System Flowchart	Forms, Layouts, Grid Charts	English I Playscrip Program HIPO
STRUCTURED	Data Flow Diagram	Data Dictionary, Data Structure Diagrams, E-R Diagrams	Decision Structure Structure Warnier/
DATA MODELING (INFO RMATION ENGINEERING)	Business Area Analysis, Process Model	Business Area Analysis, E-R Diagrams	Business Design
OBJECT-	Object Model	Object Model	Static &

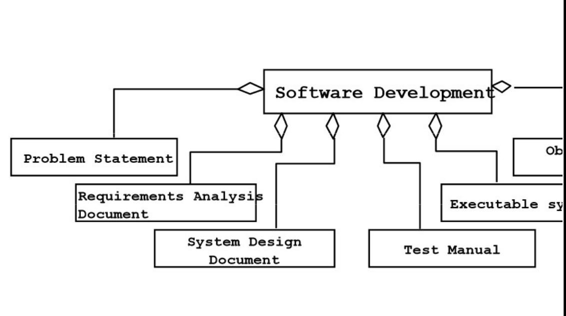
Object Oriented Lifecycle Model



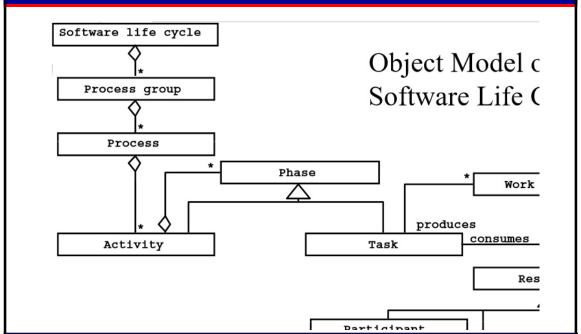
Software Development- Usecase Model



Software Development- Simple Object Model



Software Development- Object Model



OOLCM- Analysis

- Object Modelling is based on **identifying** the **objects** in a system and their **interrelationships**.
- As in any other system development model, system analysis is the **first phase** of development in case of Object Modelling too.
- In this phase, the **developer** interacts with the **user** of the system to find out the **user requirements** and **analyses** the system to understand the **functioning**.
- Based on this system study, the analyst prepares a **model** of the **desired system**.
- This model is purely based on what the system is *required to do*.
- At this stage the implementation details are not taken care of. Only the model of the system is prepared based on the **idea** that the system is made up of a set of **interacting objects**.
- The important **elements** of the system are **emphasized**.

OOLCM- Design

- System Design is the next development stage where **the overall architecture** of the **desired system** is decided.
- The system is organized as a **set of sub systems interacting** with each other.
- While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in **system analysis** as well as what is required out of the new system by the end user.
- As the basic philosophy of Object-Oriented method of system analysis is to perceive the system as a *set of interacting objects*, a bigger system may also be seen as a set of **interacting smaller subsystems** that in turn are composed of a set of **interacting objects**.

OOLCM- Design

- While designing the system, the stress lies on the **objects** comprising the system and not on the processes being carried out in the system as in the case of traditional Waterfall Model where the processes form the important part of the **system**.

Object Orientation in Design

- In this phase, the details of the **system analysis** and **system design** are **implemented**.
- The ***Objects identified*** in the system design phase are **designed**.
- Here the implementation of these objects is decided as the **data structures** get defined and also the ***interrelationships*** between the **objects** are defined.
- **Object Oriented Philosophy** is very much similar to real world and hence is gaining popularity as the systems here are seen as a ***set of interacting objects*** as in the real world.
- To implement this concept, the ***process-based structural programming*** is not used; instead **objects** are created using **data structures**.
- Just as every programming language provides various data types and various variables of that type can be created, similarly, in case of **objects** certain *data types are predefined*.

Object Orientation in Design

- For example, we can define a data type called **pen** and then create and use several **objects** of this data type. This concept is known as creating a **class**.
- **Class**: A class is a ***collection of similar objects***. It is a template where certain basic characteristics of a set of objects are defined. The class defines the **basic attributes** and the **operations** of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created instances of the class are created as per the requirement of the case.
- **Abstraction**: **Classes** are built on the basis of abstraction, where a *set of similar objects* are observed and their **common characteristics** are listed. Of all these, the characteristics of concern to the system under observation are picked up and the class definition is made. The attributes of no concern to the system are left out. This is known as **abstraction**.

Object Orientation in Design

The abstraction of an object varies according to its application. For **instance**, while defining a pen class for a stationery shop, the attributes of concern might be the pen color, ink color, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc.

- **Inheritance**: Inheritance is another important concept in this regard. This concept is used to apply the idea of *reusability of the objects.* A new type of class can be defined using a **similar existing class** with a few new features. For **instance**, a class vehicle can be defined with the basic functionality of any vehicle and a new class called car can be derived out of it with a few modifications. This would save the developers time and effort as the classes already existing are reused without much change.

OOLCM- Implementation

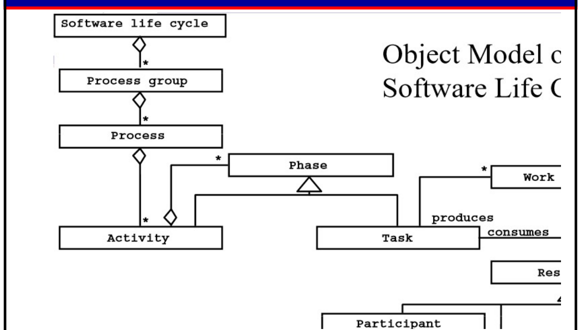
- During this phase, the **class objects** and the interrelationships of these **classes** are translated and actually coded using the programming language decided upon.
- The **databases** are made and the complete system is given a ***functional shape***.
- The complete OO methodology revolves around the **objects** identified in the system.
- When observed closely, every object exhibits some **characteristics** and **behaviour**.
- The objects recognize and respond to certain **events**.
- For example, considering a Window on the screen as an object, the size of the window gets changed when resize button of the window is clicked.

OOLCM- Implementation

- Here the clicking of the button is an event to which the window responds by changing its state from the old size to the new size. While developing systems based on this approach, the analyst makes use of certain models to analyse and depict these objects. The methodology supports and uses three basic Models:
  - **Object Model** - This model describes the **objects** in a system and their **interrelationships**. This model observes all the objects as **static** and does not pay any attention to their dynamic nature.
  - **Dynamic Model** - This model depicts the **dynamic** aspects of the system. It portrays the **changes** occurring in the **states** of various objects with the events that might occur in the system.
  - **Functional Model** - This model basically describes the **data** transformations of the system. This describes the *flow of data* and the changes that occur to the data throughout the system.

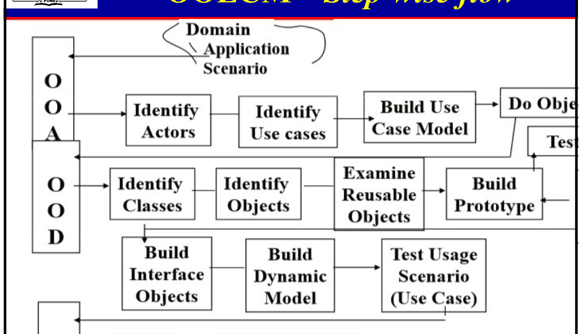
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.115

OOLCM- *Detailing!*



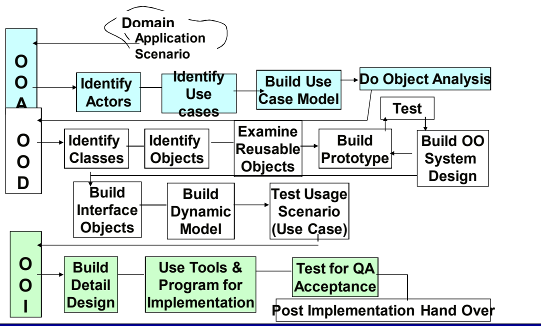
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.116

OOLCM – *Step-wise flow*



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.117

Life Cycle Model OOSAD



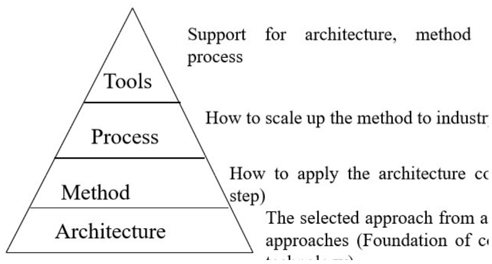
Object Orientation- *Advantages*

- Object Oriented Methodology closely represents the **problem domain**. Because of this, it is easier to produce and understand **designs**.
- The objects in the system are **immune** to **requirement changes**. Therefore, allows changes more easily.
- Object Oriented Methodology designs **encourage** more **re-use.** New applications can use the existing modules, thereby **reduces** the *development cost* and *cycle time*.
- Object Oriented Methodology approach is more **natural**. It provides nice **structures** for thinking and abstracting and leads to **modular design**.

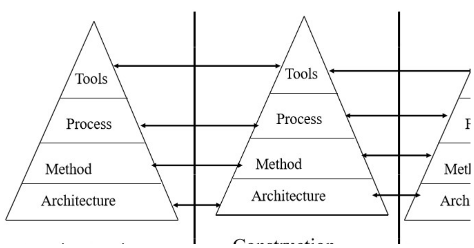
Software Development- Industrial Process

- Process must yield a **foreseeable result**, **irrespective** of which **individual** performed the job.
- **Volume of output** doesn't effect the **process**.
- Possibility to **allocate parts** of the **process** to several **manufacturers** and **subcontractors**.
- Possible to make use of **pre-defined building blocks** and **components**.
- Possible to **plan** and **calculate** the process with **great precision**.
- Each **person** trained for an **operation** must perform it in a **similar manner.**

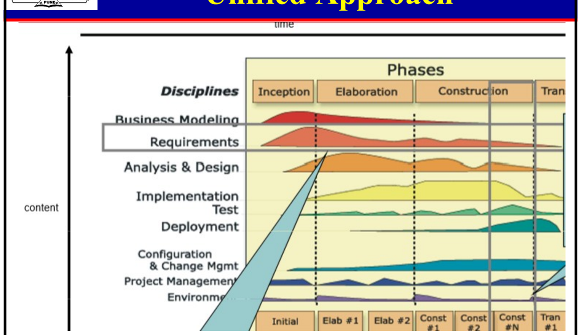
Rational Enterprise Philosophy



Multiple Activities- R.E.



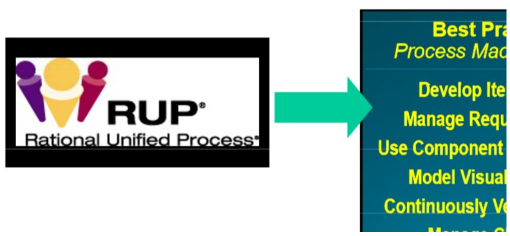
Unified Approach



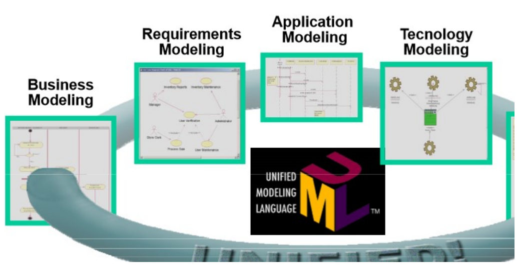
Unified Approach

- Based on the best practices
- Unify the modeling efforts of Booch, Rum Jacobson
- Revolves around the processes and concepts
  - Use-case driven development
  - Object Oriented Analysis
  - Object oriented Design
  - Incremental development and prototyping

RUP implements best practices



One language for all practitioners



Unified Approach Phases

Has four phases

- Inception
  - ✓**Understand problem**
- Elaboration
  - ✓**Understand Solution**
- Construction
  - ✓**Have a Solution**
- Transition

 U1.127

Disciplines of RUP

1. Business Modelling
2. Requirements
3. Analysis and Design
4. Implementation
5. Test
6. Deployment

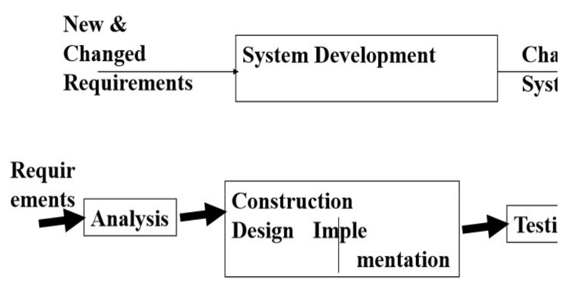
 U1.128

System Development Characteristics

- Part of a larger activity
- System development
- Transition from analysis to construction
- Requirements are inputs to system development
- A system is output system development
- Parties interested in system development are customer, direct and indirect users, etc.

 U1.129

System Development



Responsibility-Driven Design

- In Responsibility-Driven Design, a model is developed from the **requirements specification** by the **extraction** of **nouns** and **verbs** from the specification.
- This provides a basis for the **actual implementation.**
- In RDD, for each class, **different responsibilities** are defined which specify the **roles of the objects,** and their **actions.**
- In order to fulfill these responsibilities, classes need to **collaborate** with each other. Collaborations are defined to show *how the objects will interact.*
- The responsibilities are further grouped into **contracts** which define a **set of requests** that objects of the class can support.
- These contracts are further refined into **protocols**, which show the **specific signature** of each operation**.**

Responsibility-Driven Design

- The RDD modelling process includes two **phases**:
- **Exploratory**: The exploratory phase has three goals-- finding the **classes**, determining **responsibilities**, and identifying **collaborations**. This is commonly done with the CRC Cards.
- **Analysis**: The analysis phase involves refining the **object's behaviour** and the **service definitions** specified in the exploratory phase. These activities include defining **interfaces** (protocols) and constructing **implementation specifications** for each class.

Class Responsibility Collaboration

Classes
- Extract **noun phrases** from the **specification** and build a list
- Identify candidates for abstract **super classes**
- Use categories to look for missing classes
- Write a short statement for the **purpose** of each **class**

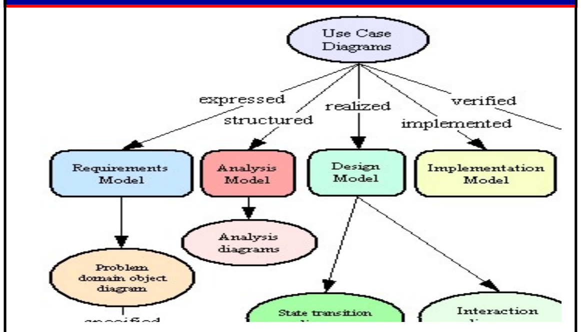
Responsibilities
- Find **responsibilities**
- Assign responsibilities to **classes**
- Find additional responsibilities by looking at the **relationships** between classes

Collaborations
- Find and list **collaborations** by examining responsibilities associated with classes
- Identify additional collaborations by looking at **relationships between classes**
- **Discard** and classes that take part in no collaboration (as client or server)

Object-Oriented Software Engineering

- Object-oriented software engineering (OOSE) is an **object modeling language** and **methodology**.
- Object-Oriented Software Engineering (OOSE) is a **software design technique** that is used in software design in object-oriented programming.
- OOSE is developed by **Ivar Jacobson** in 1992. OOSE is the first object-oriented design methodology that employs use cases in software design. OOSE is one of the precursors of the Unified Modeling Language (UML), such as Booch and OMT.
- It includes a **requirements**, an **analysis**, a **design**, an **implementation** and a **testing** model.
- **Interaction diagrams** are similar to UML's sequence diagrams. State transition diagrams are like UML **statechart diagrams.**

Object-Oriented Software Engineering



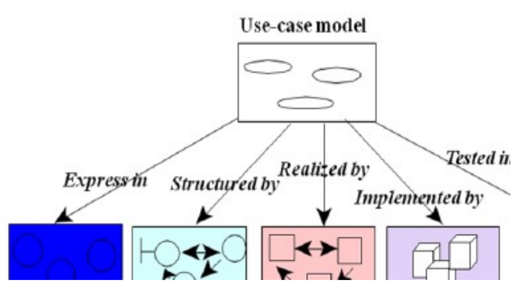
Jacobson OOSE

- Object-Oriented Software Engineering (OOSE) is a **software design technique** that is used in **software design** in object-oriented programming.Originated from **Objectory** (Object Factory for software development)
- OOSE is developed by **Ivar Jacobson** in 1992. OOSE is the first object-oriented design methodology that employs use cases in software design. OOSE is one of the **precursors** of the Unified Modeling Language (**UML**), such as Booch and OMT.
- It includes a **requirements**, an **analysis**, a **design**, an **implementation** and a **testing** model.
- *Interaction diagrams* are similar to UML's *sequence diagrams*. State transition diagrams are like UML *statechart diagrams*.

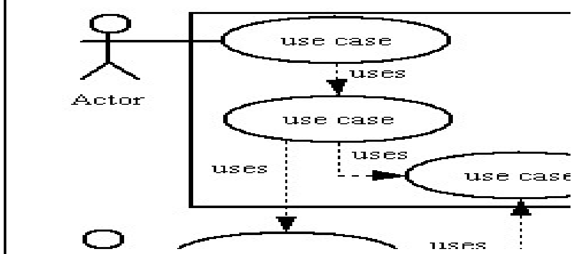
Jacobson OOSE

- Aim to fit the development of **large real-time system**
- **Stress traceability** among the different phases (Backward & forward)
- Supports OO concepts of **classification**, **encapsulation** and **inheritance**.
- Abstraction is promoted by levels.
- Adds "*use cases*" to the OO approach.
- **Composite data** and **activity definition** is not strongly enforced and services are also regarded as **objects**.
- **Reuse** is supported by **component libraries**.
- Guidance for analysis is less comprehensive.
- Target applications: like **HOOD real-time systems** and **engineering systems**.

Jacobson's Use Case Model



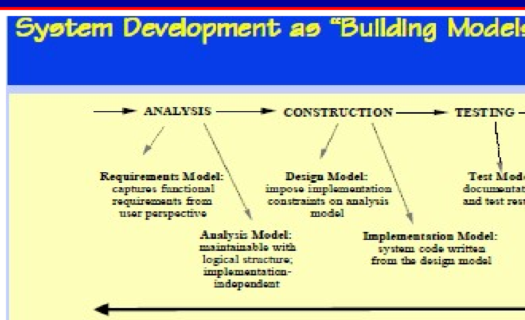
Jacobson’s Use Case Diagram



Objectory

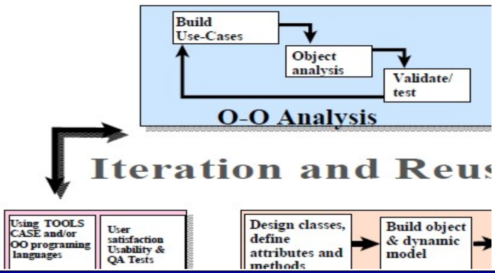
- **Discipline process** for the industrialized development of software, based on a *use case driven design*
- Built around several different models
  - Requirement Model
  - Domain object model
  - Analysis Model
  - Design Model
  - Implementation model
  - Test model

OOSE Models



Object Oriented System Dev

Object-Oriented Systems Development Approach



OOSE- Requirement Model

- Two different models are developed in **OOSE;** the **Requirements Model** and the **Analysis Model.**
- These are based on **requirement specifications** and discussions with the **prospective users**.
- The first model, the **Requirements Model**, should make it possible to ***define the system*** and to define what functionality should take place within it.
- For this purpose we develop a **conceptual picture** of the system using problem **domain objects** and also **specific interface descriptions** of the system if it is meaningful for this system.
- We also describe the system as a **number of use cases** that are performed by a number of actors.

OOSE- Analysis Model

- The Analysis Model consisting of various **object classes**: ***control object, entity objects,*** and ***interface objects***.
- The purpose of this model is to find a **robust and extensible structure** for the **system** as a **base for construction**.
- Each of the **object types** has its own **special purpose** for this robustness, and together they will offer the total functionality that was specified in the Requirements Model.

OOSE- Construction

- We build our system through **construction** based on the Analysis Model and the Requirements Model created by the analysis process.
- The construction process lasts until the **coding** is completed and the included units have been tested.
- There are three main reasons for a construction process:
  1) The Analysis Model is not sufficiently formal.
  2) Adaptation must be made to the actual implementation environment.
  3) We want to do internal validation of the analysis results.

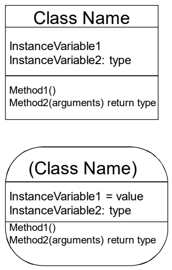
OOSE- Construction

- The construction activity produces two models, the **Design Model** and the **Implementation Model**.
- Construction is thus divided into two phases; **design** and **implementation**, each of which develops a model.
- The **Design Model** is a further **refinement** and **formalization** of the Analysis Model where consequences of the implementation environment have been taken into account.
- The **Implementation model** is the actual **implementation** (code) of the system.

OOSE- Testing

- Testing is an activity to **verify** that a **correct system** is being built.
- Testing is traditionally an **expensive activity**, primarily because many faults are not detected until late in the development.
- To do effective testing we must have as a **goal** that every test should detect a fault

Object Model Notation



Classes are represented as rectangles;

The class name is at the top, followed by attributes (instance variables) and methods (operations)

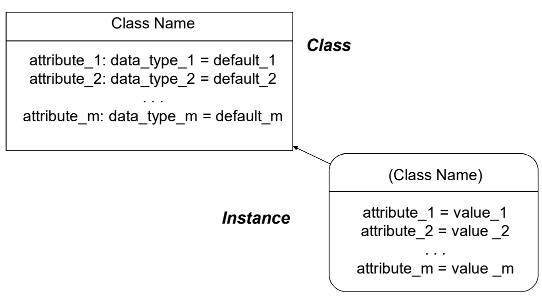
Depending on context some information can be hidden such as types or method arguments

Objects are represented as rounded rectangles;

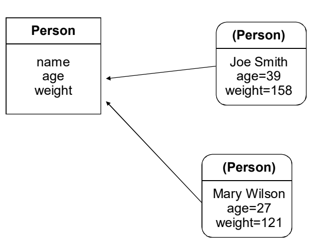
The object's name is its classname surrounded by parentheses

Instance variables can display the values that they have been assigned; pointer types will often point (not shown) to the object being referenced

OMT Instantiation Notation



Instantiation - Example



Inheritance

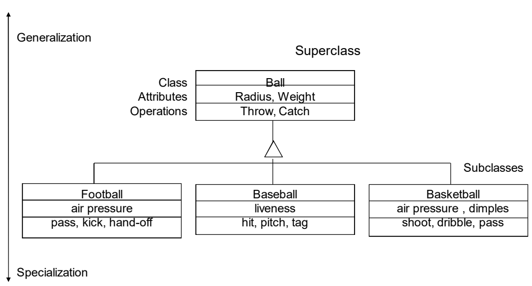
Classes with similar **attributes** and operations may be organized **hierarchically**

Common attributes and operations are factored out and assigned to a broad superclass (*generalization*)

- Generalization is the "**is-a**" relationship
- Super classes are ancestors, subclasses are descendants

Classes iteratively refined into subclasses that *inherit* the attributes and operations of the superclass (*specialization*)

OMT Inheritance Notation



Association and Links

An **association is a relation among two or more classes** describing a group of links, with common structure and semantics

A **link is a relationship or connection between objects** and is an instance of an association

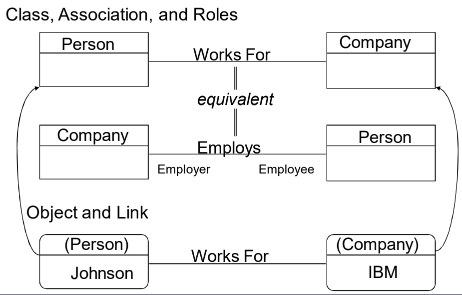
A link or association is inherently *bi-directional*

- the name may imply a direction, but it can usually be inverted
- the diagram is usually drawn to read the link or association from left to right or top to bottom

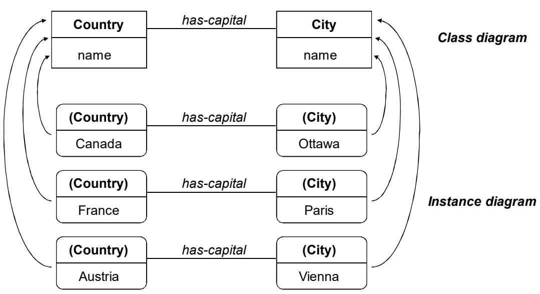
A role is one end of an association

- roles may have names

OMT Association Notation



Association and Links



Aggregation

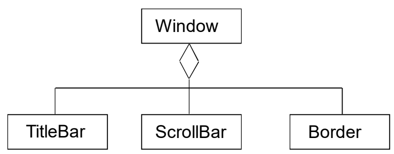
Aggregation is a special form of association that indicates a **"part-of"** relationship between a whole and its parts

Useful when the parts do not have **independent existence**

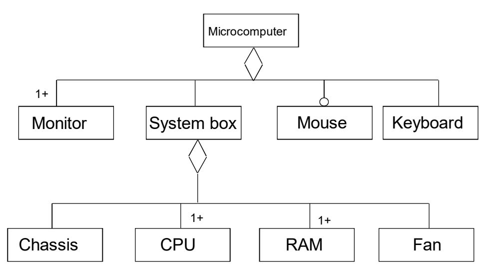
- A part is ***subordinate*** to the whole

In an aggregation, properties and operations may be propagated from the whole to its parts

OMT Aggregation Notation



OMT Multilevel Aggregation



States and Transitions

A *state* is an **interval** between **events** (values of relevant variables to the problem…)

- it may have an activity that can trigger starting, intermediate and ending events
- defined in terms of a subset of object attributes and links

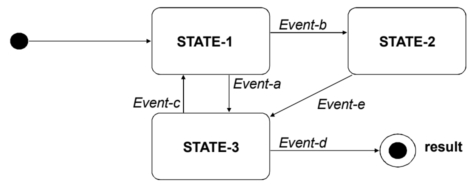
A state *transition* is a change in an object's attributes and links

- it is the response of an object to an event
- all transitions leaving a state must correspond to distinct events

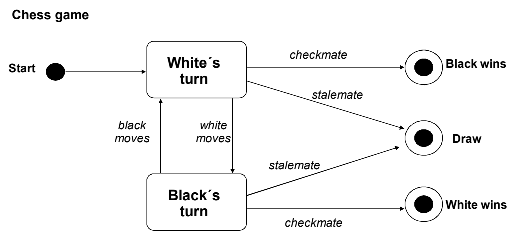
OMT State Notation

tates represented as nodes: **rounded rectangles with state name**
- initial state represented as solid circle
- final state represented as bull's eye

transitions represented as edges between nodes and labeled with an *event name*



OMT State Diagram - Example

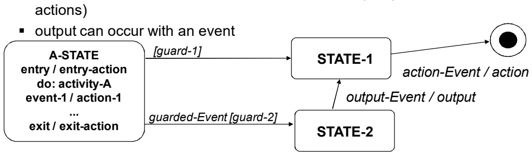


Guards, Activities and Actions

Guards are Boolean conditions on attribute values
- transition can only happen when guard evaluates to "true"
- automatic transitions occur as soon as an activity is complete (check guard!)

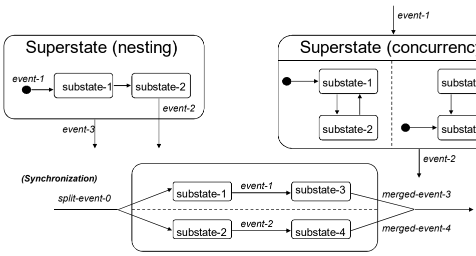
Activities take time to complete
- activities take place within a 'state'

Actions are relatively instantaneous
- actions take place on a transition or within a state (entry, exit, event actions)
- output can occur with an event

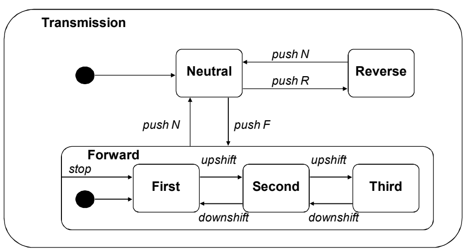


OMT State Relationships

States can be nested or concurrent
Events can be split and merged



State Generalization: example



Structured vs. Unified Process

Criteria	***Structured Methodology***	***Object Oriented (Unified Process)***
Use of development activities (Planning, Analysis..)	Each activity covers a whole phase in SDLC	All activities run in each phase, N-times (iterations)
Names of development phases	Planning, Analysis, Design, Implementation, Installation/Testing	Inception, Elaboration, Construction, Transition
Appropriate to use	When system goals certain, static IT	When system goals less certain, dynamic IT
Modeling technique	Data Flow Diagrams, Entity-Relationship Diagrams	Diagrams defined by *Unified Modeling Language* (Use Cases, Class Diagrams…)
Relation to reality	Predictive	Adaptive

Unified Approach

•UA based on methodologies by Booch, Rumbaugh and Jacobson tries to combine the **best practices**, **processes** and **guidelines** along with the object management groups in unified modelling language.

•UA utilizes the *unified modeling language* (UML) which is a set of notations and conventions used to describe and model an application.

Goals:

- Define **Objects** and **classes**
- Describe objects' **methods**, **attributes** and how objects respond to messages
- Define **Polymorphism**, **Inheritance, data abstraction, encapsulation**, and **protocol**
- Describe **objects relationships**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.166

Object-Oriented Software Dev

Object-Oriented Methodology

- Development approach used to build **complex systems** using the concepts of object, class, polymorphism, and inheritance with a view towards ***reusability***
- Encourages software engineers to think of the problem in terms of the **application domain** early and apply a **consistent approach** throughout the entire life-cycle

Object-Oriented Analysis and Design

- Analysis models the "*real-world*" requirements, independent of the implementation environment
- Design applies object-oriented concepts to develop and communicate the **architecture** and details of how to meet requirements

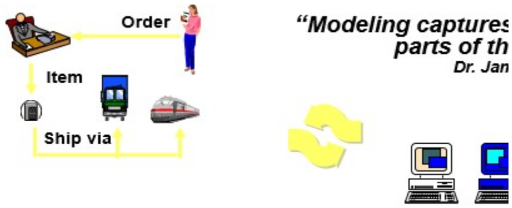
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.167

Visual Modelling

- **Mapping real-world process** of a **computer system** with a ***graphical representation*** is called visual modelling.
- Visual Modeling is a ***way of thinking*** about **problems** by using **graphical models** of **real-world ideas.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM U1.168

Visual Modelling



"Modeling captures
parts of th
Dr. Jan

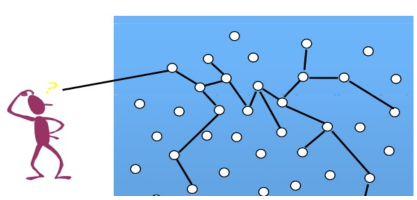
Visual Modeling is
modeling

Benefits of Visual Modelling

- Captures Business Process
- Enhance Communication
- Manage Complexity
- Define Architecture
- Enable Reuse

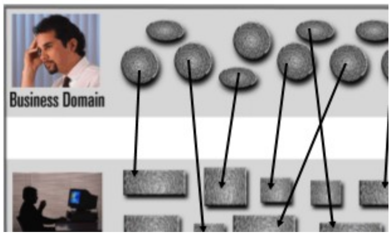
I. Capture Business Processes

- When we create **use cases,** visual modeling allows us to **capture business processes** by defining the ***software system requirements*** from the **user's perspective.**

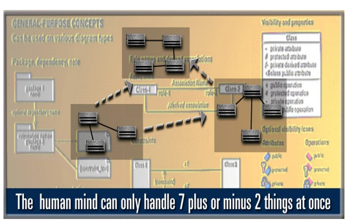


II. A Communication Tool

Use visual modeling to capture business object



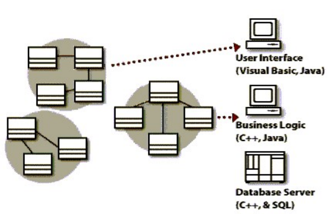
III. Manages Complexity



Systems today typically have hundreds or even thousands of classes. These classes must be organized in such a way as to allow viewing by many different groups of people; often with their own viewing needs.

Visual modeling provides the capability to display modeling elements in many ways, so that they can be viewed at different levels of abstraction.

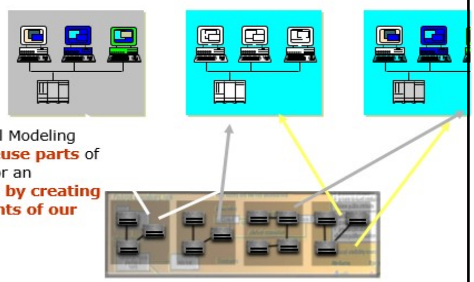
IV. Defines Software Architecture



Visual modeling provides the **capability** to capture the **logical software architecture independent** of the **implementation language**.

As **system design** progresses, the **implementation language** is determined and the **logical architecture** is **mapped** to the **physical architecture**.

V. Promotes Re-use



With Visual Modeling **we can reuse parts** of a system or an application **by creating components of our design**.

Design Method Evolution

1997	OOAD with UML
1990	OOAD
1986	CASE
1986	OOP
1980	4GL
1980	Data Modeling
1980	Structured Analysis
1975	Structured Design

UML Origin

A product of the "**design wars**" of the 1980's

Grady Booch, James Rumbaugh, and others had competing styles.

`94: Rumbaugh leaves GE to join Booch at Rational Software

"Method wars over. We won." Others feared achieving standardization the Microsoft way.

'95: Rational releases **UML 0.8**; Ivars Jacobson (use cases) joins Rational→"The Three Amigos"

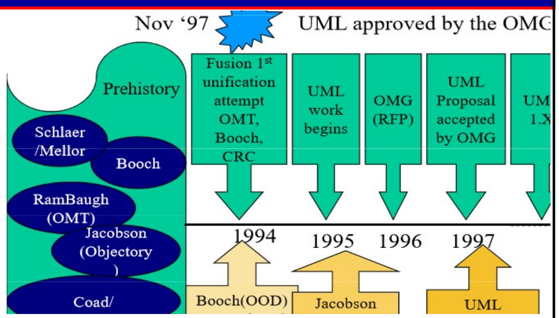
'96: Object Management Group sets up task force on methods

'97: Rational proposed **UML 1.0** to **OMG**. After arm twisting and merging, **UML 1.1** emerges

'99: After several years of revisions and drafts, **UML 1.3** is released

Now **UML 1.5**….

UML Origin



What is UML

- "***The Unified Modeling Language is a family of graphical notations, that help in describing and designing software systems, particularly software systems built using the object-oriented style.***"
- UML first appeared in **1997**
- UML is **standardized**. Its content is controlled by the Object Management Group (OMG), a group of companies.
- UML can be applied to diverse **application domains** (e.g., banking, finance, internet, aerospace, healthcare, etc.) It can be used with all major object and component **software development methods** and for various **implementation platforms** (e.g., J2EE, .NET).

What is UML

- UML stands for **Unified Modelling Language**
- The UML combines the best of the best from
  - Data Modelling concepts (Entity Relationship Diagrams)
  - Business Modelling (work flow)
  - Object Modelling
  - Component Modelling
- The UML is the **standard language** for **visualizing**, **specifying**, **constructing**, and **documenting** the artifacts of a *software-intensive system*
- It can be used with all **processes**, throughout the **development life cycle,** and across different **implementation technologies**

How it all begun…

UML Contributors

Meyer
Before and after conditions

Harel
Statecharts

Gamma, et al
Frameworks and patterns,

HP Fusion
Operation descriptions and message numbering

Booch
Booch method

UNIFIED MODELING LANGUAGE

Rumbaugh
OMT

Embley
Singleton classes and high-level view

Jacobson
OOSE

Wirfs-Brock
Responsibilities

Shlaer - Mellor
Object lifecycles

Odell
Classification

UML Supports

Objects

Business Objects

Relationships

large sc

ORDBMS
Oracle

Classes

UNIFIED MODELING LANGUAGE

applicat

Components
Microsoft

Scena

Goals of UML

1) Provide users with a **ready-to-use, expressive** visual modeling language

2) Provide **extensibility** and **specialization** mechanisms to **extend** the **core concepts**.

3) Be **independent** of particular **programming languages** and **development processes**.

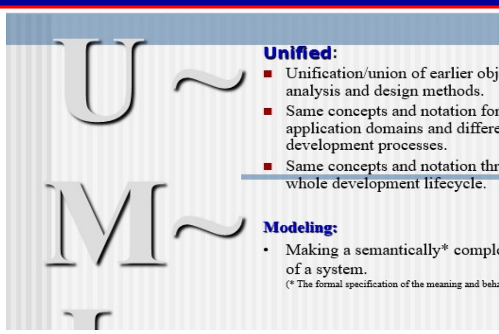
4) Provide a *formal basis for understanding* the modeling language.

5) Encourage the **growth** of the **OO tools market.**

6) Support **higher-level development concepts** such as **collaborations**, **frameworks**, **patterns** and **components**.

7) **Integrate best practices.**

UML



UML- A language for…

- The UML is a language for
  - **visualizing**
  - **specifying**
  - **constructing**
  - **documenting**

  a software-intensive system

- UML can also be applied outside the **domain** of software development.

UML- A language for…

1. **Visual Modelling**

'A picture is worth a thousand words'

- Use standard **graphical notations**
- **Semi-formal**
- Captures **business processes** from **enterprise information systems** to **distributed web-based applications** and even to hard **real time embedded systems.**

UML- A language for…

2. **Specifying**

•**Building models** that are
- Precise
- Unambiguous
- Complete

•**Symbols** are based upon
- Well-defined syntax
- semantics

•Addresses the **specification** of all important **analysis**, **design** and **implementation decisions**.

UML- A language for…

3. **Constructing**

•Models are related to **object oriented programming languages**

•**Round-trip engineering** requires tools and human invention to information loss.

- **Forward engineering-** direct mapping of a UML model into code.
- **Reverse engineering-** reconstruction of a UML model from an implementation.
- **Re-engineering-** Understanding existing software and modifying it.

UML- A language for…

4. **Documenting**
•Architecture

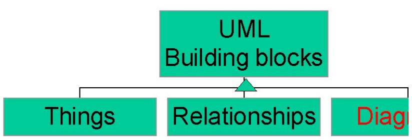
•Requirements

•Tests

•Activities
- ✓Project Planning
- ✓Release Management

3 basic building blocks of UML

Building Blocks of UML



3 basic building blocks of UML

- **Things**
  important modeling concepts
- **Relationships**
  tying individual things together
- **Diagrams**
  grouping interrelated collections of thin

Why model?

- **Analyse** the **problem-domain**
  - simplify reality
  - capture requirements
  - visualize the system in its entirety
  - specify the structure and/or behaviour of the system
- **Design** the **solution**
  - document the solution - in terms of its structure, behavior, etc.

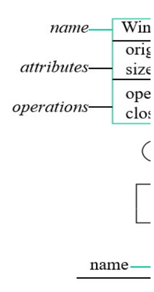
Conceptual Model of UML

- A conceptual model needs to be formed by an individual to understand UML.
- **UML contains three types of building blocks: things, relationships, and diagrams.**
- **Things**
  - Structural things
    - ✓Classes, interfaces, collaborations, use cases, components, and nodes.
  - Behavioral things
    - ✓Messages and states.
  - Grouping things
    - ✓Packages
  - Annotational things
    - ✓Notes
- **Relationships**: dependency, association, generalization ,composition ,link ,aggregation etc..
- **Diagrams**: class, object, use case, sequence, collaboration, statechart, activity, component and deployment

I. Structural Things- *7 Things*

1. Class

A description of a set of objects that share the same attributes, operations, relationships, and semantics.



2. Interface

A collection of operations that specify a service (for a resource or an action) of a class or component. It describes the externally visible

I. Structural Things

3. Collaboration

- Define an interaction among two or more cl
- Define a society of roles and other element:
- Provide cooperative behavior.
- Capture structural and behavioral dimensio
- UML uses 'pattern" as a synonym

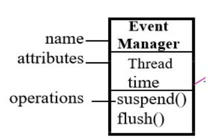
I. Structural Things

4. Use Case

- A sequence of actions that produce an obsei result for a specific actor.
- A set of scenarios tied together by a commor goal.
- Provides a structure for behavioral things.
- Realized through a collaboration

I. Structural Things

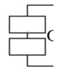
5. Active Class



- Special class whose objects own one or m *processes* or *threads*.

I. Structural Things

6. Component

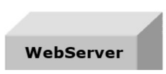


- Replaceable part of a system.
- Components can be packaged logically.
- Conforms to a set of interfaces.
- Provides the realization of an interface.

I. Structural Things

7. Node

- Element that exists at *run time*.
- Represents a *computational resource*.
- Generally has memory and processing pov



II. Behavioral Things

- **Verbs** of UML Model
- **Dynamic parts** of UML models- *behaviour over time and space.*
- Usually **connected** to **structural things** in UML.

II. Behavioral Things

Two primary kinds of behavioral things:

Interaction
behavior of a set of objects comprising of a set o exchanges within a particular context to acc specific purpose.

State Machine
behavior that specifies the sequences of states ar an interaction goes through during its lifetime in re events, together with its responses to those events

III. Grouping Things

Packages –
- one primary kind of grouping.

- General purpose mechanism for organizing eleme groups.
- Purely conceptual; only exists at development time.
- Contains behavioral and structural things.
- Can be nested.

IV. Annotational Things

- Explanatory parts of UML models
- Comments regarding other UML element called adornments in UML)

Note is the one primary annotational thing in UM

- best expressed in informal or formal text.

Conceptual Model- Relationship

- Dependency
- Association
- Generalization

Conceptual Model - Diagrams

•Graphical representation of a set of elements.

•Represented by a connected graph:
- •Vertices are things;
- •Arcs are behaviors.

•5 most common views built from 9 diagram type

1.Class Diagram; Object Diagram
2. Use case Diagram
3. Sequence Diagram; Collaboration D
4. Statechart Diagram
5. Activity Diagram

Different perspectives of a system

A System
- Functional Model
  - Use Case Di
- Structure Model
  - Class/Object
- Behavior Model
  - Activity Diag
  - Sequence Di
  - Collaboratio
  - Statechart D
- Implementation Model
  - Component

Architectural Views and Diagrams

- **User model view**
  - relies on ***use case diagrams*** to describe the problem and its solution from the perspective of the customer or end user of a product
- **Structural model view**
  - describes static aspects of the system through ***class diagrams*** and ***object diagrams***
- **Behavioral model view**
  - specifies dynamic aspects of the system through ***sequence diagrams, collaboration diagrams, state diagrams, and activity diagrams***

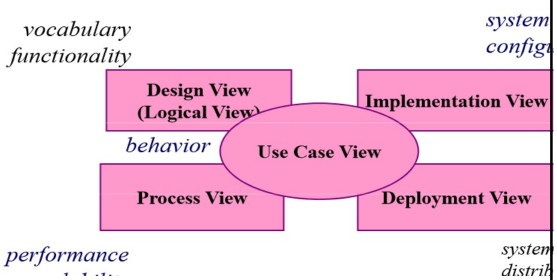
Architectural Views and Diagrams

- **Implementation model view**
  - concentrates on the **specific realization** of a solution, and depicts the organization of solution components in ***component diagrams***
- **Environment model view**
  - shows the **configuration of elements** in the environment, and indicates the mapping of solution components to those elements through **deployment diagrams**

4+1 Architecture of UML

- Architecture refers to the **different perspectives** from which a **complex system** can be **viewed.**
- The architecture of a *software-intensive system* is best described by five interlocking views:
  - **Use case view**: system as seen by **users, analysts** and **testers**.
  - **Design view**: **classes, interfaces and collaborations** that make up the system.
  - **Process view**: **active classes** (threads).
  - **Implementation view**: **files** that include the **system**.
  - **Deployment view**: **nodes** on which **SW resides**.

4+1 Architecture of UML



UML Concepts- The 4+1 View

- Use Case view
  - ✓Understandability
- Logical View
  - ✓Functionality
- Process View
  - ✓Performance
  - ✓Scalable
  - ✓Throughput
- Implementation View
  - ✓Software management
- Deployment View
  - ✓System topology
  - ✓Delivery
  - ✓Installation

I. Use Case View(*User View*)

- Technique to capture business process fro perspective.
- Encompasses the behavior as seen by users and testers.
- Specifies forces that shape the architecture.
- Static aspects captured in use case diagrams.
- Dynamic aspects captured in interaction

II. Design View(*Logical View*)

- Encompasses classes, interfaces, and collabor that define the vocabulary of a system.
- Supports functional requirements of the system.
- Static aspects captured in class diagrams and diagrams.
- Dynamic aspects captured in interaction, state

III. Process View

- Encompasses the threads and processes concurrency and synchronization.
- Addresses performance, scalability, and throu
- Static and dynamic aspects captured as in

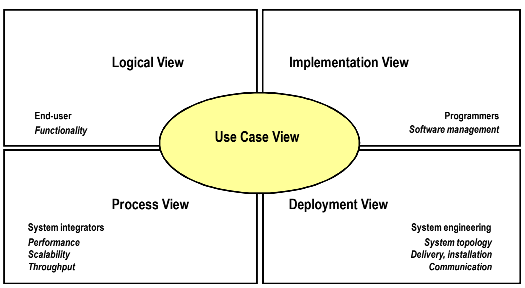
IV. Implementation View

- Encompasses components and files used to ass release a physical system.
- Addresses configuration management.
- Static aspects captured in component diagrams.
- Dynamic aspects captured in interaction, sta

V. Deployment View

- Encompasses the nodes that form the system topology.
- Addresses distribution, delivery, and installation.
- Static aspects captured in deployment diagrams.

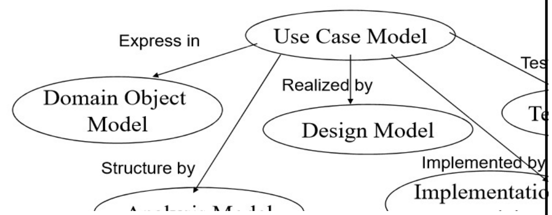
Representing System Architecture



UML Concepts in a nutshell

- Display the boundary of a system & functions using use cases and actors
- Illustrate use case realizations with diagrams
- Represent a static structure of a system u diagrams
- Model the behavior of objects with state diagrams
- Reveal the physical implementation ar

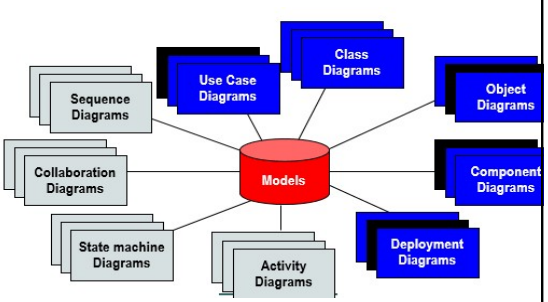
UML Concepts in a nutshell



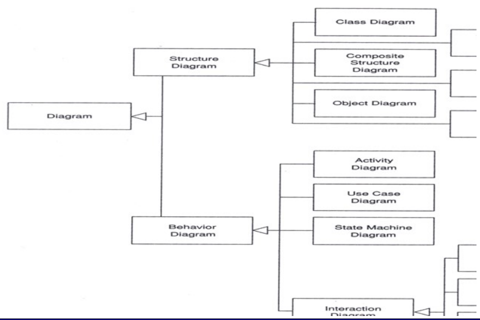
UML Diagram Classification

- **User model view**
  - use case diagrams
- **Structural model view**
  - class diagrams
  - object diagrams
- **Behavioral model view**
  - sequence diagrams
  - collaboration diagrams
  - state machine diagrams
  - activity diagrams
- **Implementation model view**
  - component diagrams
- **Environment model view**
  - deployment diagrams

UML Standardized Diagrams

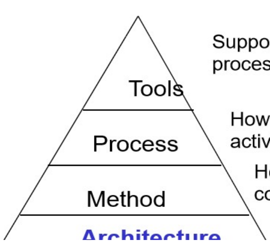


UML Diagram Classification



Architecture

System Architecture



Support for architecture, method process

How to scale up the method to activity

How to apply the concept(step by step)

The selected approac

System Architecture

- **System development** includes the development of **different models** of a software system
- **Aim** is to find **powerful modeling language**, **notation or modeling technique** for each model
- Set of **modeling techniques** defines **architecture** upon which the system development method is based
- The **architecture of a method** is the denotation of its set of modeling techniques

System Architecture

- **Modeling technique** is described by means of syntax, semantics and pragmatics
  - **Syntax** (How it looks)
  - **Semantics** (What it means)
  - **Pragmatics** (rules of thumb for using modeling technique)

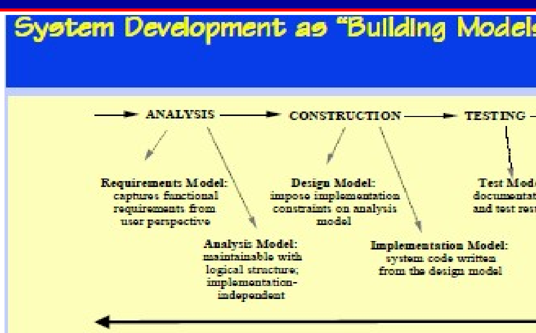
System Development

- **System development** is the work that occurs when we develop **computer support** to aid an **organizational procedures.**
- *System development is model building.*
- Commences with **identification** of **requirements**.
- **Specification** can be used for **contract** and to **plan** and **control development process**.
- Complex processes are often handled poorly. **OOSE** steps in from start to end of system life cycle.

Objectory Models

- Discipline process for the industrialized **development of software,** based on a **use-case driven design**
- **Built around several different models**
  - Requirement Model
  - Domain object model
  - Analysis Model
  - Design Model
  - Implementation model
  - Test model

System development is *Model Building*



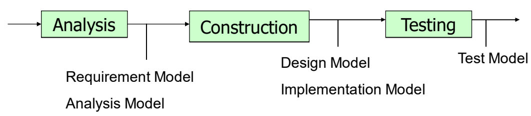
Model Architecture

Five different models

- The requirement model
  - ✓**Aims to capture the functional requirements**
- Analysis model
  - ✓**Give the system a strong and changeable object structure**
- Design model
  - ✓**Adopt and refine the object structure to the current implementation environment**

Model Architecture

- Implementation model
  - **Implement the system designed so far**
- Test model
  - **Verify whether the right system has been built or not**



Model Architecture

The Analysis Model

• Consisting of **various object classes: control object, entity objects,** and **interface objects.**

•The purpose of this model is to find a **robust** and **extensible structure** for the system as a **base for construction.**

• Each of the **object types** has its own special purpose for this robustness, and together they will offer the **total functionality** that was specified in the **Requirements Model**.

Model Architecture

The Construction Model

- We build our system through **construction** based on the **Analysis Model** and the **Requirements Model** created by the **analysis process**.
- The construction process lasts until the **coding is completed** and the included **units** have been tested.
- There are three main reasons for a construction process:

1) **The Analysis Model** is **not sufficiently formal**.
2) **Adaptation** must be made to the **actual implementation environment.**
3) We want to do **internal validation** of the **analysis results.**

Model Architecture

•The construction activity produces **two models,** the **Design Model** and the **Implementation Model.**

• Construction is thus divided into **two phases**; ***design and implementation***, each of which develops a model.

- ✓ The **Design Model** is a further **refinement** and **formalization** of the **Analysis Model** where consequences of the **implementation environment** have been taken into account.
- ✓ The **Implementation model** is the **actual implementation (code)** of the system**.**

Model Architecture

The Testing Model

• Testing is an activity to **verify** that a **correct system is being built.**

• Testing is **traditionally** an expensive activity, primarily because many ***faults are not detected*** until late in the **development**.

• To do **effective testing** we must have as a **goal** that *every test should detect a fault*

Development Processes

- Instead of focusing on how a **specific project** should be **driven**, the focus of the process is on how a certain product should be *developed* and *maintained* during its life cycle
- Divide the development work for a specific product into **processes**, where each of the processes describes *one activity of the management of a product.*
- **Processes** works in a highly **interactively** manner.
- Process handles the **specific activity** of the system development

Development Processes

- Architecture forms the **basis** of the method and process, that is the concept of each model
- Development can be regard as a set of **communicating processes**
- **System development** depends on these processes

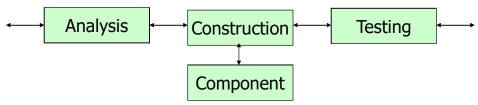
Development Processes

- All **development work** is managed by these processes.
- Each **process** consist of a **number of communicating sub processes.**
- **Main processes are**
  - Analysis
  - Construction
    - ✓**Component**
  - Testing

Processes and Models

- **Models** of the system created during **development**
- To **design models process description** is required
- Each process takes *one or several models* and transform it into other models
- Final model should be **complete** and **tested**, generally consists of source code and documentation
- **System development** is basically concerned with developing models of the system

Development Processes



I. Analysis Process

- Creates **conceptual picture**
- **Output** are **requirement model** and **Analysis model**
- Requirement Model
  - ✓**Done by use cases in the use case model**
  - ✓**Form the basis of construction and testing process**
  - ✓**Forms the basis of analysis Model**

Development Processes

- ✓**Basis of system structure**
- ✓**Specify all the logical objects to be included in the system and how these are related and grouped**
- ✓**Provide input for the construction process**

II. Construction Process

- Develops **design model** and **implementation model**
- Includes the implementation and results in complete system
- **Design Model**
  - ✓**Each object will be fully specified**
  - ✓**Consider the implementation constraints**

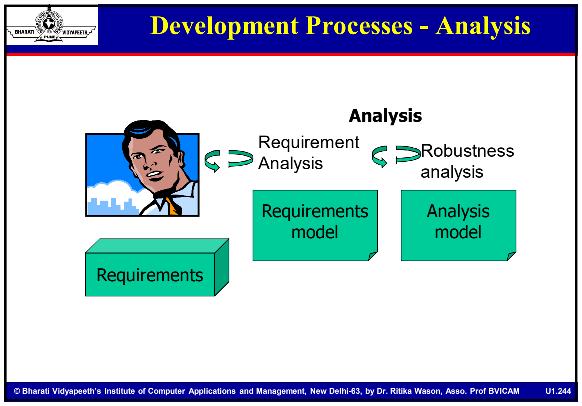
Development Processes

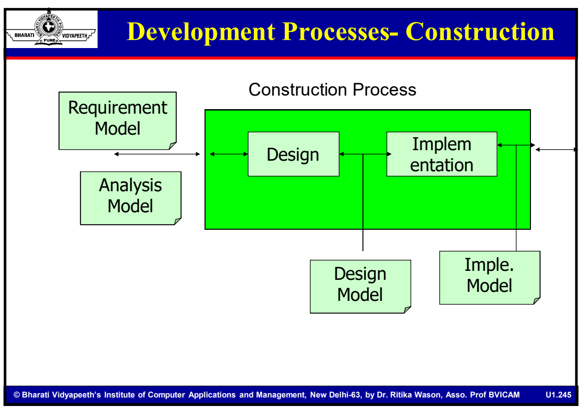
III. Testing Process

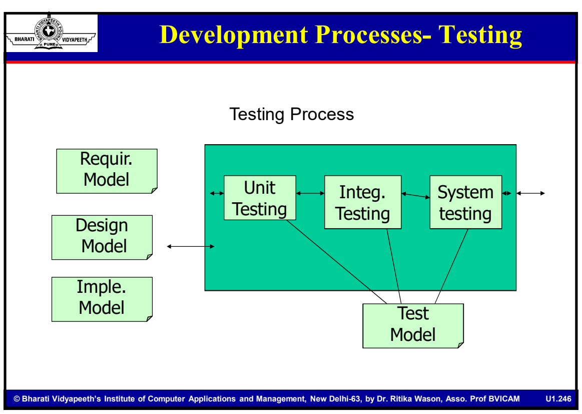
- **Integrates** the **system**, **verifies** it and decides whether it should be **delivered**

IV. Component development process

- Communicates with the **construction process**
- **Develops** and **maintain** **components**


Development Processes - Analysis
Analysis
Requirement Analysis
Robustness analysis
Requirements model
Analysis model
Requirements
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM
U1.244


Development Processes- Construction
Construction Process
Requirement Model
Analysis Model
Design
Implem entation
Design Model
Imple. Model
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM
U1.245


Development Processes- Testing
Testing Process
Requir. Model
Design Model
Imple. Model
Unit Testing
Integ. Testing
System testing
Test Model
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof BVICAM
U1.246

OOSE Analysis Models

- Object-oriented software engineering (OOSE) proposes two analysis models for understanding the **problem domain**
  - Requirements Model
  - Analysis Model

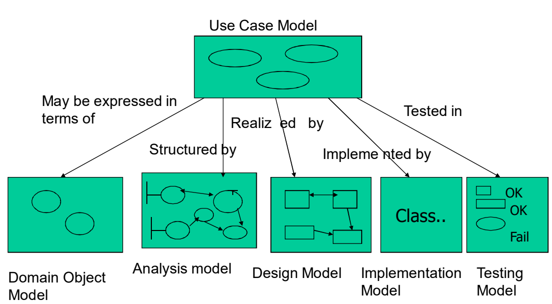
The **requirements model** serves two main **purposes**
- To delimit the **system**
- To define the **system functionality**

Requirement Model

- **Conceptual model** of the system is developed using:
  - **Problem domain objects**
  - **Specific interface descriptions** of the system (if meaningful to the system being developed)

⊗ The system is described as a *number of use cases* that are performed by a *number of actors*
- **Actors** constitute the **entities** in the **environment** of the system
- **Use cases** describe what takes places **within the system**
- A use case is a **specific way** of **using** the system by performing **some part** of the system functionality

Requirement Model

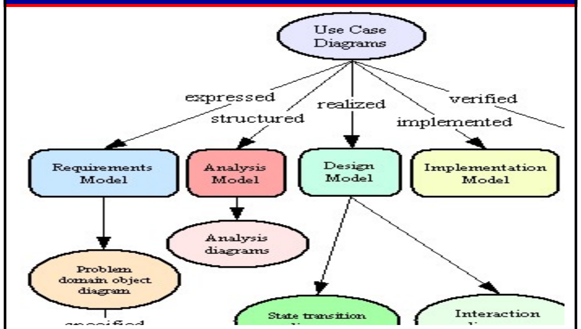


Requirement Model

❑ The requirements model for the will comprise three main models of representation:

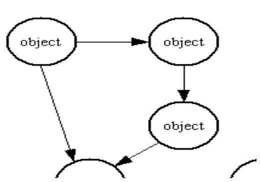
- The use case model
- The problem domain model
- User interface descriptions

Requirement Model



Problem Domain Object Model

Provides a logical view of the system, which is used to specify the use cases for use case diagrams



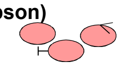
Object Oriented Analysis

- **Identifying objects**: Using concepts, CRC cards, stereotypes, etc.
- **Organising the objects**: classifying the objects identified, so similar objects can later be defined in the same class.
- **Identifying relationships between objects**: this helps to determine inputs and outputs of an object.
- **Defining operations of the objects**: the way of processing data within an object.
- **Defining objects internally**: information held within the objects.

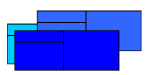
 U1.253

Object Oriented Analysis Approaches

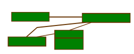
1. Analysis model with stereotypes (Jacobson)
   - **Boundaries, entities, control.**



2. CRC cards (Beck, Cunningham)
   - **Index cards and role playing.**



3. Conceptual model (Larman)
   - **Produce a "light" class diagram.**



A good analyst knows more than one strategy and even may mix strategies

 U1.254

The Analysis Phase

- Begins with a **problem statements** generated during **system conception.**
- In software engineering, analysis is the process of **converting** the user requirements to system specification (system means the software to be developed).
- System specification, also known as the **logic structure**, is the developer's view of the system.
- **Function-oriented analysis**
  - Concentrating on the **decomposition** of complex functions to simply ones.
- **Object-oriented analysis**
  - Identifying **objects** and the **relationship** between objects.

 U1.255

Analysis Model

⊗ The analysis model gives a **conceptual configuration** of the system.

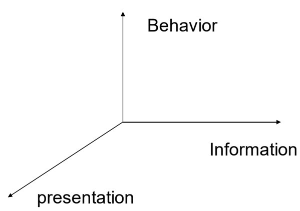
It consists of:

- The entity objects
- Control objects
- Interface objects

⊗ The analysis model forms the **initial transition** to **object-oriented design**

Dimensions of Analysis Model

Dimensions of the analysis model



Analysis Model- *Objects*

Entity object

- Information about an entity object is stored even after a use case is completed.

Control object

- A control object shows functionality that is not contained in any other object in the system

Interface object

- Interface objects interact directly with the environment

Requirement Model Structured in Analysis Model

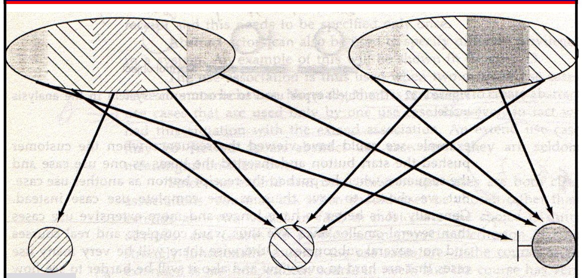


Figure 7.13 Each use case is distributed among analysis objects. Several use cases can have objects in common.

Design Model

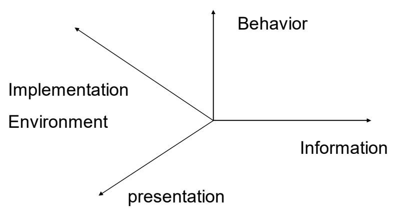
- Developed based on the analysis model
  - Implementation environment is taken into consideration
- The considered **environment factors** includes
  - Platform
  - Language
  - DBMS
  - Constraints
  - Reusable Components
  - Libraries
  - so on..

Design Model

- Design objects are different from analysis objects
- **Models**
  - Design object interactions
  - Design object interface
  - Design object semantics
    - ✓ **(i.e., algorithms of design objects' operations)**
- More closer to the **actual source code**

Design Model Dimensions

Dimensions of the Design model



Design Model

- Use **block term** in place of object
- Sent from one block to another to trigger an execution
- A typical **block** is **mapped** to **one file**
- To manage system **abstractly** subsystem concept is introduced
- **Analysis Model** is viewed as **conceptual** and **logical model,** whereas the **design model** should take as closer to the **actual source code**
- Consist of explained source code
- OO language is desirable since all fundamentals concepts can easily be mapped onto **language constructs**
- Strongly desirable to have an easy match between a **block** and the **actual code module**

Implementation Model

- Consists of **annotated source code.**
- Object oriented language is desirable since all **fundamental concepts** can be easily **mapped** onto **language constructs**.
- Strongly desirable to have an easy **match** between a **block** and the **actual code module.**

Test Model

Fundamental concepts are **test specifications** and the **test results**

References

1. Ivar Jacobson, “Object Oriented Software Engineering”, Pearson, 2004.
2. Grady Booch, James Runbaugh, Ivar Jacobson, “The UML User Guide”, Pearson, 2004
3. R. Fairley, “Software Engineering Concepts”, Tata McGraw Hill, 1997.
4. P. Jalote, “An Integrated approach to Software Engineering”, Narosa, 1991.
5. Stephen R. Schach, “Classical & Object Oriented Software Engineering”, IRWIN, 1996.
6. James Peter, W Pedrycz, “Software Engineering”, John Wiley & Sons
7. Sommerville, “Software Engineering”, Addison Wesley, 1999.
8. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/userguide.html
9. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/statemachinediagram.html
10. http://www.developer.com/design/article.php/2238131/State-Diagram-in-UML.htm
11. http://www.mariosalexandrou.com/methodologies/extreme-programming.asp