UNIT 1 AN OVERVIEW OF OBJECT ORIENTED SYSTEM DEVELOPMENT

Introduction - Object Oriented System Development Methodology - Why Object Orientation - Overview of Unified Approach - Object Basics: Object Oriented Philosophy - Objects - Classes - Attributes - Object Behavior and Methods, Encapsulation and Information Hiding - Class Hierarchy - Polymorphism - Object Relationships and Associations - Aggregations and Object Containment - Object Identity - Static and Dynamic Binding - Persistence. Object Oriented Systems Development Life Cycle: Software Development Process - Building High Quality Software - Usecase Driven Approach - Reusability.

Introduction:

Object-Oriented Development uses "objects" to model real world objects. A car or a laptop can be considered as object. While traditional programming views software as a collection of functions, an object oriented system concentrates on the objects that combines data and functionality together.

The traditional approach mostly focussed on structured system development and the technique used was usually referred to as the Structured Analysis and Design Technique (SADT).

Object

An object is something which has its own identity and can be easily compared to a real world object like a car or a laptop. An object contains a state and some behavior. The state of an object is the properties of the object at a particuler time, and behavior is the functions it will perform. The behaviour of an object is usually described using methods, and these methods will be part of the object itself. So you don't have to refer anywhere else for object's functionality, whereas in function based traditional approach you need to remember all the methods and their location. For instance, in java, the state of an object is the set of values of an object's variables at any particular time and the behaviour is implemented as methods.

Class

Objects are grouped into a class. A class can be defined as a group of objects with the same structure and behaviour. Class can be considered as the blueprint or definition or a template for an object and describes the properties and behavior of that object, but without any actual existence. An object is a particular instance of a class which has actual existence and there can be many objects (or instances) for a class. In the case of a car or laptop, there will be a blueprint or design created first and then the actual car or laptop will be built based on that. We do not actually buy these blueprints but the actual objects.

Object oriented development

In Object-Oriented Development, we apply object orientation across every system development activity such as requirement analysis, design, programming, testing, and maintenance. For instance, an object oriented analysis (OOA) will usually have the following steps:

- Identifying the system functionality

- Identifying the involved objects
- Recognizing the object classes
- Analysing the objects to fulfil the system functionality

Object-Oriented Programming (OOP) is based on object oriented features like Encapsulation, Polymorphism, Inheritance and Abstraction. These features are usually referred to as the OOPs concepts. We will see more about analysis, design, testing and maintenance later.

Encapsulation

Encapsulation is the process of wrapping up of data (as properties) and behavior (as methods) of an object into a single unit (a class). Encapsulate in plain English means *to enclose or be enclosed in or as if in a capsule*.

Encapsulation enables **data hiding,** hiding irrelevant information from the users of a class and exposing only the relevant details required by the user. We can expose our operations to the outside world and hide the details of what is needed to perform that operation. We can thus protect the internal state of an object by hiding its attributes from the outside world, and then exposing them through setter and getter methods. Now modifications to the object internals are only controlled through these methods.

Example from real world: Consider the smart phone you are using now. You are not worried about the internal operations of the smart phone. You only know and care about the operations or functions it exposes to you such as making call, sending SMS or using your apps.

Inheritance

Inheritance describes the relationship between two classes. A class can get some of its characteristics from a parent class and then add unique features of its own. For example consider a Vehicle parent class and its child class Car. Vehicle class will have all common properties and functionalities for all vehicles in common and Car will inherit those common properties from the Vehicle class and then add those properties which are specific to a car. Here, Vehicle is known as base class, parent class, or superclass. Car is known as derived class, Child class or subclass.

In multiple inheritance a class can inherit from more than one parent, but due to its complexity some languages like Java doesn't fully support it. In multi-level inheritance there will be many levels of inheritence like A inheriting from B and B inheriting from C and so on.

Polymorphism

Polymormism means that one operation can be used for different purposes. Poly means many and morph means form. For instance, Java supports different kinds of polymorphism like oveloading, overriding, parametric etc. In overloading, the multiple methods having same name can appear in a class, but with different signature. Overriding is defining a method in a subclass with the same name and type signature as a method in its superclass and the overriden method is called at runtime based on the object at runtime.

Abstraction

In plain English, abstract means a concept or idea not associated with any specific instance and does not have a concrete existence. Abstraction in OOP emphasizes what is important and what is not important at a particular level of detail. Abstraction refers to the ability to make a class abstract at a level and define some of the behaviour at each level, relevant to the current perspective. Abstraction allows the programmer to focus on few concepts at a time. Java provides interfaces and abstract classes for describing abstract types.

Coupling vs Cohesion

Coupling and cohesion are two important concepts in the Object Oriented design and hence we will briefly discuss it here.

Coupling is the degree to which one class knows about another class. If the knowledge is only through exposed interfaces (data hiding), it is called loosely coupled. If the knowledge is more like accessing data members directly, it is called tightly coupled. We should try to make our code as loosely coupled as possible. Even though you make some change in a class adhering strictly to the class's API, tight coupling can make other classes that use this class not working properly after the change.

The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. The more focused a class is, the higher its cohesiveness, which is a good thing. The key benefit of high cohesion is that such classes are typically much easier to maintain than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

In summary, loose coupling and high cohesion are desirable, whereas tight coupling and less cohesion can lead you to problems in the long run.

Advantages of object oriented software development

Some of the advantages of object oriented software development are:

- Less maintenance cost mostly because it is modular.
- Better code reusability due to features such as inheritance and hence faster development.
- Improved code reliability and flexibility
- Easy to understand due to realworld modelling.

- Better abstraction at object level.
- Reduced complexity during the transitions from one development phase to another.

Object Oriented System Development Methodology:

Object Oriented Methodology (OOM) is a system development approach encouraging and facilitating re-use of software components. With this methodology, a computer system can be developed on a component basis which enables the effective re-use of existing components and facilitates the sharing of its components by other systems.

 Object oriented systems development methodology develops software by building objects

that can be easily replaced , modified and reused.

 It is a system of cooperative and collaborating objects.

 Each objects has attributes (data) and methods (functions).

Why an object orientation?

Object oriented systems are

 Easier to adapt to changes

 More robust

 Easier to maintain

 Promote greater design and code reuse

 Creates modules of functionality

Reasons for working of object oriented systems:

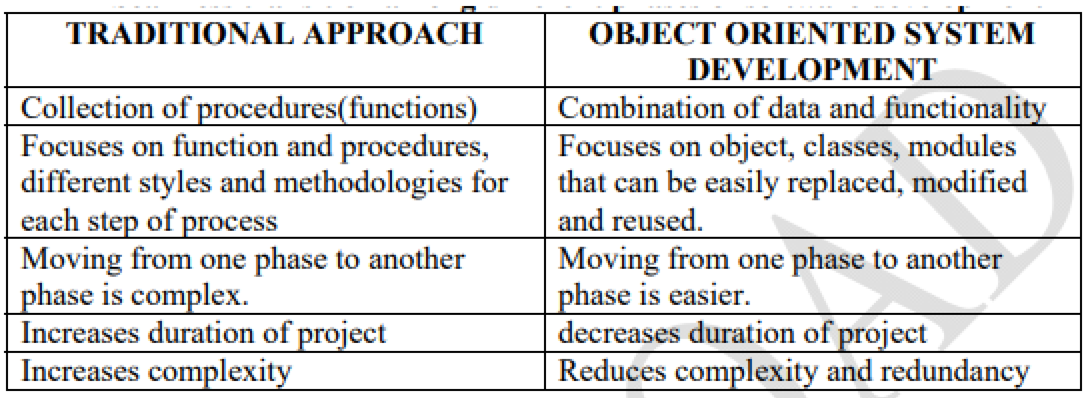
 Higher level of abstraction

 Seamless transition among different phases of software development

 Encouragement of good programming techniques

 Promotion of reusability

 Higher level of abstraction



TRADITIONAL APPROACH	OBJECT ORIENTED SYSTEM DEVELOPMENT
Collection of procedures(functions)	Combination of data and functionality
Focuses on function and procedures, different styles and methodologies for each step of process	Focuses on object, classes, modules that can be easily replaced, modified and reused.
Moving from one phase to another phase is complex.	Moving from one phase to another phase is easier.
Increases duration of project	decreases duration of project
Increases complexity	Reduces complexity and redundancy

Encouragement of good programming techniques

 Changing one class has no impact on other classes

 But there is communication between classes through interface

 Promote clear design

 Implementation is easy

 Provides for better overall communication

 Promotion of reusability

 Objects are reusable because they are modeled directly out of real world problem domain.

 Object orientation approach support abstraction at object level.

 Object encapsulates both data and functions. So they work at higher level of abstraction.

 So designing, coding, testing and maintaining the system are much simpler.

 Seamless transition among different phases of software development

 Encouragement of good programming techniques

 Changing one class has no impact on other classes

 But there is communication between classes through interface

 Promote clear design

 Implementation is easy

 Provides for better overall communication

 Promotion of reusability

 Objects are reusable because they are modeled directly out of real world problem

domain.

OBJECT ORIENTATION:

Object oriented methods enable us to create sets of objects that work together synergistically to produce software that better module their problem domains than similar systems produced by traditional techniques. The system created using object oriented methods are easier to adapt changing requirements, easier to maintain, more robust, promote greater design.

The reasons why object orientation works

- ➢ High level of abstraction.
- ➢ Seamless transition among different phases of software development.
- ➢ Encourage of good programming techniques.
- ➢ Promotion of reusability.

Top-down approach:

It supports abstraction of the function level.

Objects oriented approach :

It supports abstraction at the object level. The object encapsulate both the data (attributes) and functions (methods), they work as a higher level of abstraction. The development can proceed at the object level, this makes designing, coding, testing, and maintaining the system much simpler.

Seamless transition among different phases of software Development

Traditional Approach:

The software development using this approach requires different styles and methodologies for each step of the process. So moving from one phase to another requires more complex transistion.

Object-oriented approach:

We use the same language to talk about analysis, design, programming and database design. It returns the level of complexity and reboundary, which makes clearer and robust system development.

Encouragement of good programming techniques:

A class in an object-oriented system carefully delineates between its interface and the implementation of that interface. The attributes and methods are
encapsulated within a class (or) held together tightly. The classes are grouped into subsystems but remain independent one class has no impact on other classes. Object oriented approach is not a magical one to promote perfect design (or) perfect code. Raising the level of abstraction from function level to object level and focusing on the real-world aspects of the system, the object oriented method tends to

- Promote clearer designs.
- Makes implementation easier.
- Provide overall better communication.

Promotion of Reusability:

Objects are reusable because they are modeled directly out of real world. The classes are designed generically with reuse. The object orientation adds inheritance, which is a powerful technique that allows classes to built from each other. The only different and enhancements between the classes need to be designed and coded. All the previous functionality remains and can be reused without change.

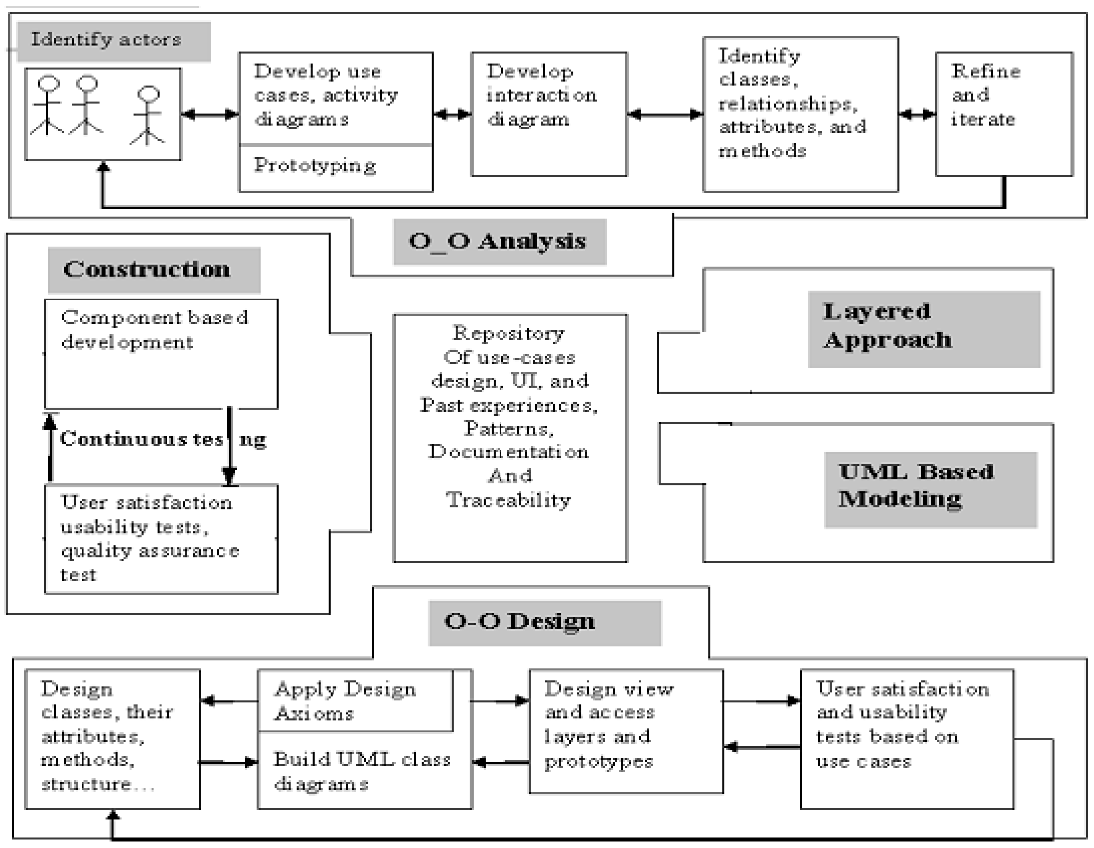
Overview of Unified Approach

The UML (Unified Modeling Language) is a set of notations and conventions used to describe and model an application.The Unified Approach (UA) specifies the tasks or steps to develope an application,The heart of UA is Jacobson's usecase.The UA consists of the following concepts:

- Usecase driven approach
- Utilizing the UML for modeling
- Object oriented analysis
- Object oriented design
- Repositories of reusable classes and maximum reuse
- The layered approach
- Incremental development and prototyping
- Continuous testing

The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs. In its simplest usage, you capture a use case by talking to typical users and discussing the various ways they might want to use the system. The use cases are entered into all other activities of the UA.

The UA establishes a unifying and unitary framework around their works by utilizing the UML to describe the model and document the software development process. The idea behind the UA is not to introduce yet another methodology. The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams for better understanding of object-oriented concepts and system development.


Identify actors
Develop use cases, activity diagrams
Prototyping
Develop interaction diagram
Identify classes, relationships, attributes, and methods
Refine and iterate
O_O Analysis
Construction
Component based development
Continuous testing
User satisfaction usability tests, quality assurance test
Repository Of use-cases design, UI, and Past experiences, Patterns, Documentation And Traceability
Layered Approach
UML Based Modeling
O-O Design
Design classes, their attributes, methods, structure...
Apply Design Axioms
Build UML class diagrams
Design view and access layers and prototypes
User satisfaction and usability tests based on use cases

The processes and components of the unified approach

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

1. Use-case driven development
2. Object-oriented analysis
3. Object-oriented design
4. Incremental development and prototyping
5. Continuous testing

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

Object-Oriented Analysis

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. It concentrates on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented require viewing the system from the user's perspective rather than that of the machine. OOA process consists of the following steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

ii. Object-Oriented Design

Booch, provides the most comprehensive object-oriented design method. Ironically, since it is so comprehensive, the method can be somewhat imposing to learn and especially tricky to figure out where to start. Rumbaugh et al.'s and Jacobson et al.'s high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson et al.'s analysis and interaction diagrams, Booch's object diagrams, and Rumbaugh et al.'s domain models. Furthermore, by following Jacobson et al.'s life cycle model, we can produce designs that are traceable across requirements, analysis, design, coding, and testing. OOD Process consists of:

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

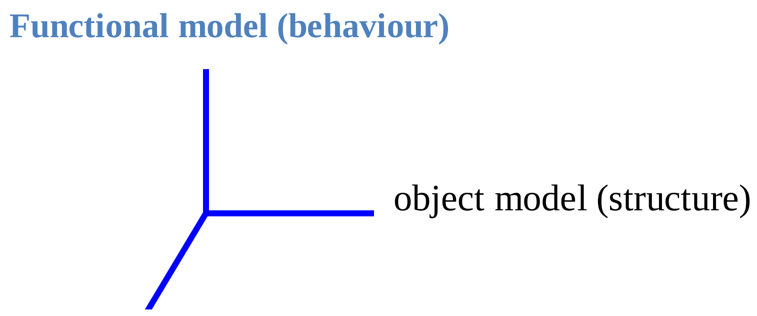
iii. Iterative Development and Continuous Testing

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

OBJECT BASICS:

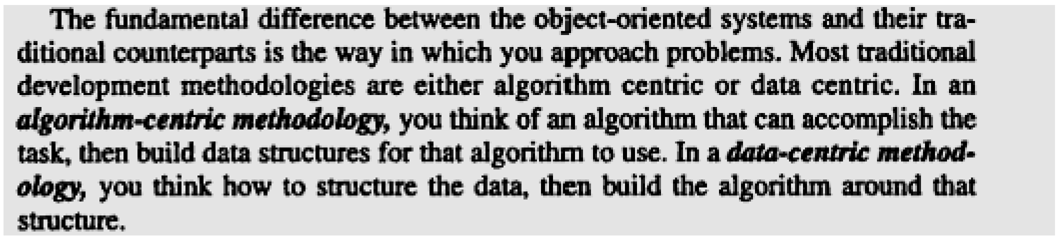
INTRODUCTION TO THE OO APPROACH

- Object-oriented approaches attempt to build models that mimic the "real world"
- Many systems we want to build involve things that exhibit behaviour as well as having structure, and also may exhibit time-dependent aspects
- OO approaches attempt to consistently capture <u>ALL</u> these aspects [*Dennis Hart 2001, Chris Loken 2002*]



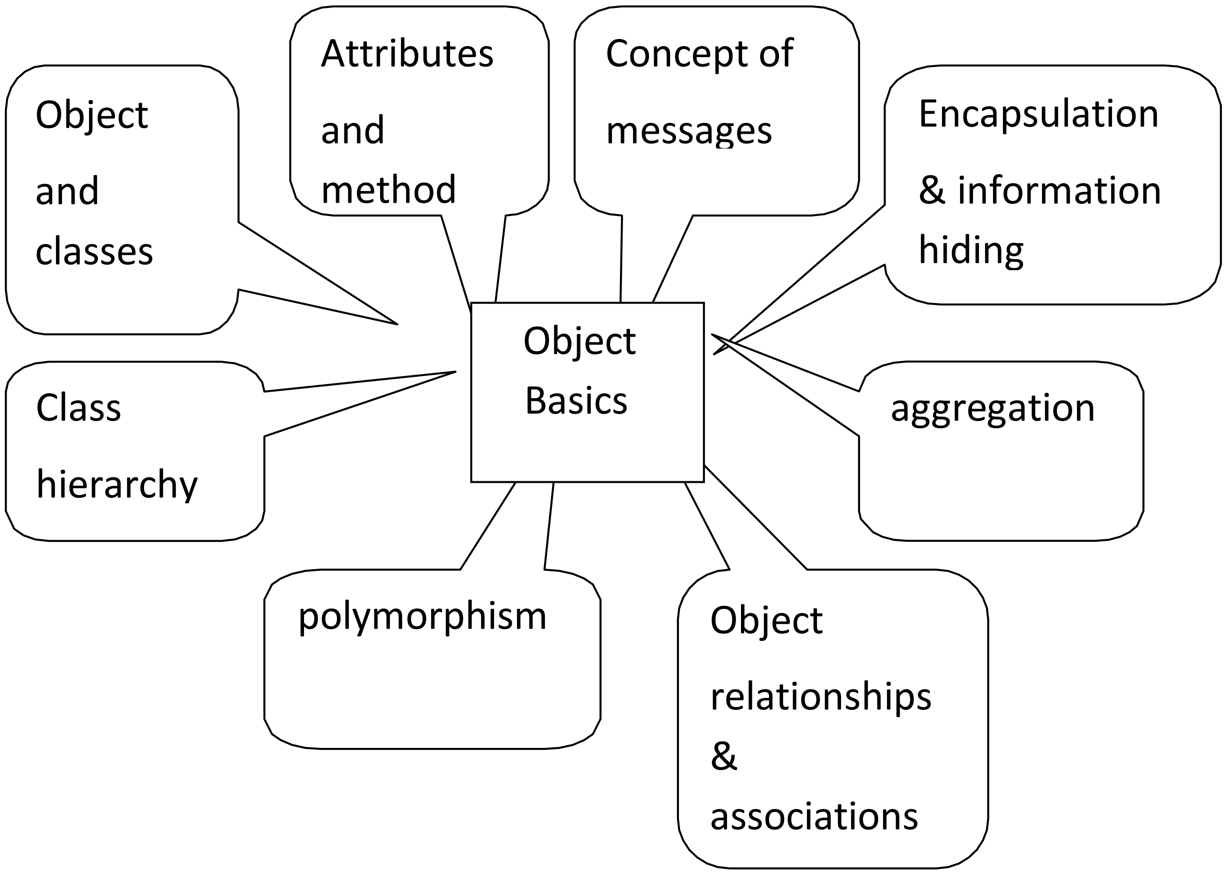
Dynamic model (temporal aspects)

- Object orientation emerged during the 1970s, particularly in programming languages like Simula, Smalltalk, Eiffel and, later, C++ and Java
- Now there are also object-oriented analysis and design techniques and tools (e.g. UML), databases and development
- Object-orientation attempts to unify many aspects of the systems analysis and design process that were disparate before
- Important aspect of OO ==> potential for *re-use of development work* when OO techniques are used
- Object-orientation has also been argued to be a more natural approach than other techniques

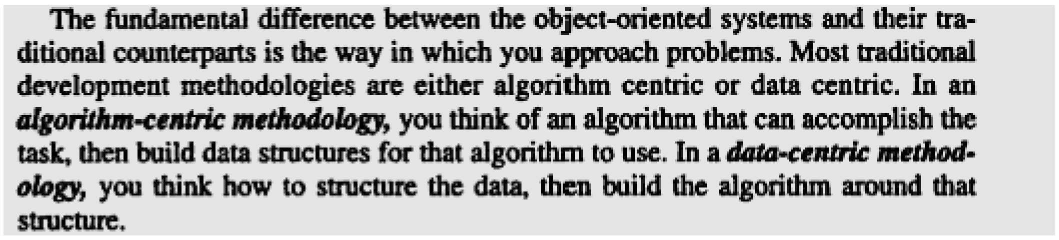


The fundamental difference between the object-oriented systems and their traditional counterparts is the way in which you approach problems. Most traditional development methodologies are either algorithm centric or data centric. In an ***algorithm-centric methodology,*** you think of an algorithm that can accomplish the task, then build data structures for that algorithm to use. In a ***data-centric methodology,*** you think how to structure the data, then build the algorithm around that structure.

OO CONCEPTS

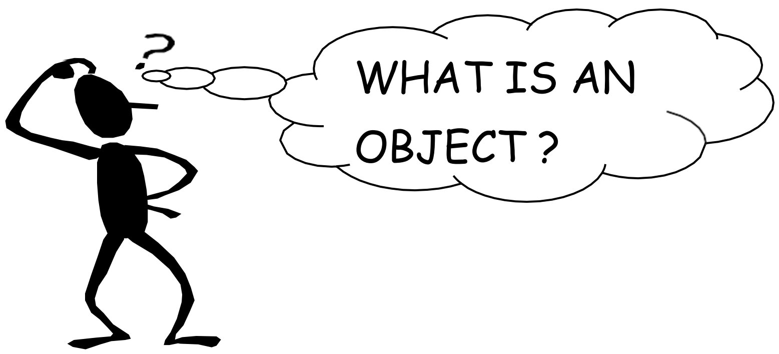


- Now there are also object-oriented analysis and design techniques and tools (e.g. UML), databases and development
- Object-orientation attempts to unify many aspects of the systems analysis and design process that were disparate before
- Important aspect of OO ==> potential for *re-use of development work* when OO techniques are used
- Object-orientation has also been argued to be a more natural approach than other techniques



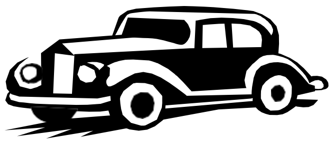
The fundamental difference between the object-oriented systems and their traditional counterparts is the way in which you approach problems. Most traditional development methodologies are either algorithm centric or data centric. In an ***algorithm-centric methodology,*** you think of an algorithm that can accomplish the task, then build data structures for that algorithm to use. In a ***data-centric methodology,*** you think how to structure the data, then build the algorithm around that structure.

❒ OBJECT AND CLASSES

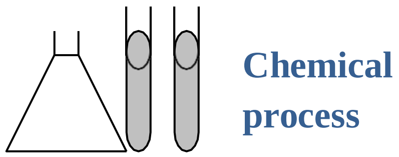


- informally , an object represents an entity, either physical, conceptual, or software

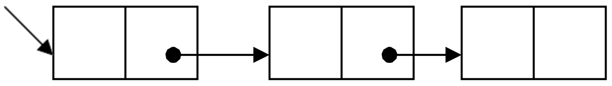
  - physical entity



  - conceptual entity



  - software entity



Linked List

- A more formal definition

An object is a concept, abstraction, or thing with sharp boundaries and meaning for an

- In an OO system, everything is an object
- *Classes* are used to distinguish one type of object from another
- A *Class* = a set of objects that share a common structure and a common behaviour

A single object is simply an *instance* of a class

- A class is a specification of structure (instance variables), behavior(methods), and inheritance for objects
- Classes are an important mechanism for classifying objects
- Main role of a class is to define the properties and procedures (the state & behavior) and applicability of its instances.

- Ex :

Class : CAR

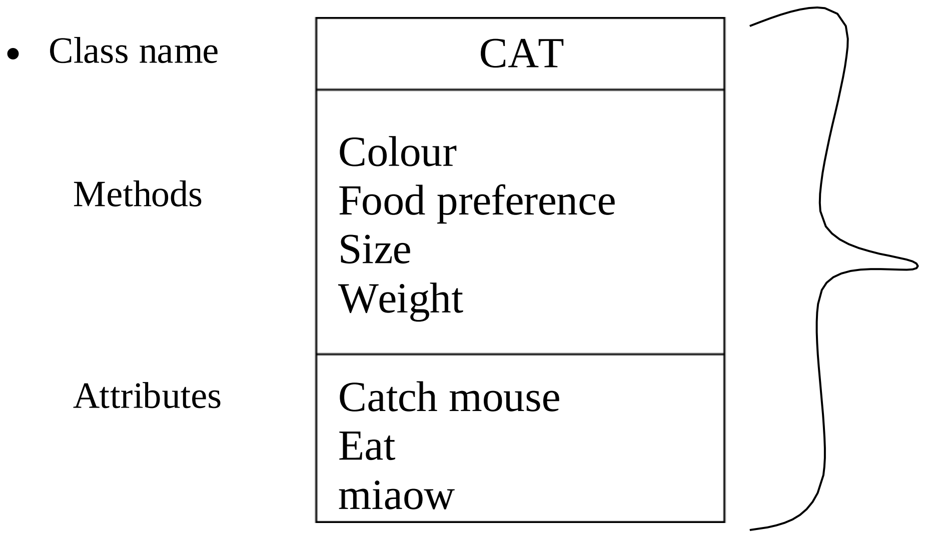
Defines the *property* = color
Each individual car will have a value for this property, such as maroon, red, blue or black

- In an OO system, a *method* or behavior of an object is defined by its class
- Each object is an instance of a class

ATTRIBUTES AND METHODS

- Objects can be described by their properties (attributes) and methods (operations)

  Ex:



- object behavior is described in methods or procedures
- a method = a function or procedure that is defined for a class and typically can access the internal state of an object of that class to perform some operation
- operations are things an object does or can have done to it
- in an object model, all data is stored as attributes of some object
- the attributes of an object are manipulated by the operations

CONCEPT OF MESSAGES

- Objects interact with each other by sending and receiving *messages*
- Messages are similar ro procedure calls in traditional programming languages
- Objects perform operations in response to messages

Ex : when you press on the brake pedal of a car, you send a STOP message to the car object. The car object knows how to respond to the STOP message

ENCAPSULATION & INFORMATION HIDING

- Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings
- An object is said to encapsulate the data and a program

User cannot see the inside of the object "capsule", but can use the object by calling the object's methods

- Encapsualtion or information hiding is a design goal of an OO system

In object-oriented system, everything is an object and each object is responsible for itself.

For example:

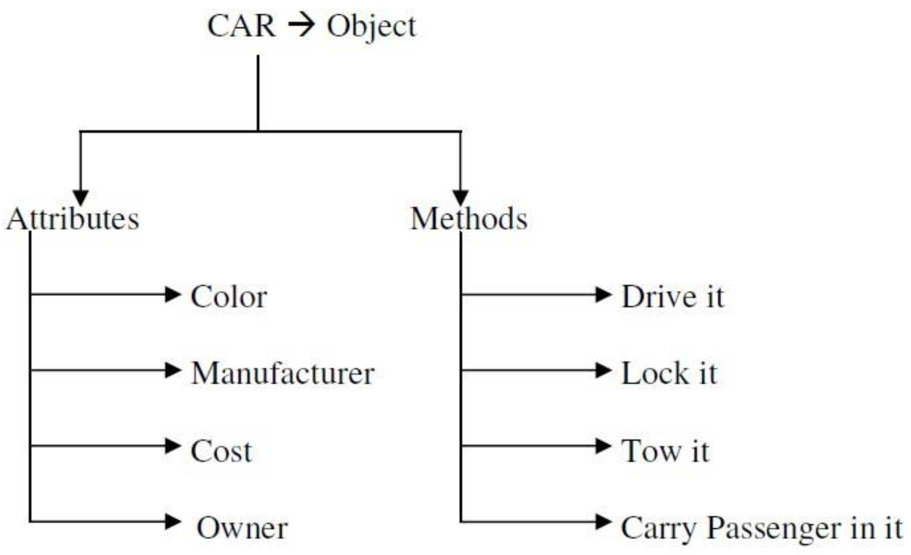
- Windows applications needs windows object that can open themselves on screen and either display something or accept input.
- Windows object is responsible for things like opening, sizing, and closing itself.
- When a windows display something, that something is an object. (ex)chart.
- Chart object is responsible for maintaining its data and labels and even for drawing itself.

Review of objects:

The object-oriented system development makes software development easier and more natural by raising the level of abstraction to the point where applications can be implemented. The name object was chosen because "everyone knows what is an object is ". The real question is "what do objects have to do with system development" rather that "what is an object?"

Object:

A car is an object a real-world entity, identifiably separate from its surroundings. A car has a well-defined set of attributes in relation to other object.



Attributes:

- Data of an object.
- Properties of an object.

Methods:

- Procedures of an object.or
- Behaviour of an object.

The term object was for formal utilized in the similar language. The term object means a combination or data and logic that represent some real-world entity.

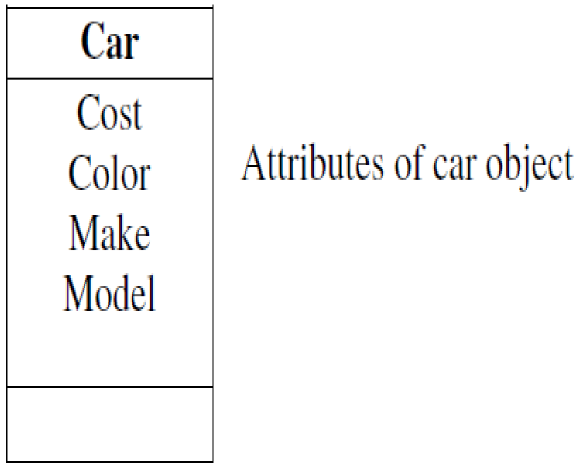
When developing an object oriented applications, two basic questions arise

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

Programming in an object-oriented system consists of adding new kind of objects to the system and defining how they behave. The new object classes can be built from the objects supplied by the object-oriented system.

Object state and properties (Attributes):

Properties represent the state of an object. In an object oriented methods we want to refer to the description of these properties rather than how they are represented in a particular programming language.



We could represent each property in several ways in a programming languages.

For example:

Color → 1. Can be declared as character to store sequence or character [ex: red, blue, ..]
2. Can declared as number to store the stock number of paint [ex: red paint, blue paint, ..]
3. Can be declared as image (or) video file to refer a full color video image.

The importance of this distinction is that an object abstract state can be independent of its physical representation.

Object Behaviour and Methods:

We can describe the set of things that an object can do on its own (or) we can
do with it.

For example:

Consider an object car,
→ We can drive the car.
→ We can stop the car.

Each of the above statements is a description of the objects behaviour. The objects behaviour is described in methods or procedures. A method is a function or procedures that is defined in a class and typically can access to perform some operation. Behaviour denotes the collection of methods that abstractly describes what an object is capable of doing. The object which operates on the method is called receiver. Methods encapsulate the behaviour or the object, provide interface to the object and hide any of the internal structures and states maintained by the object. The procedures provide us the means to communicate with an object and access it properties.

For example:

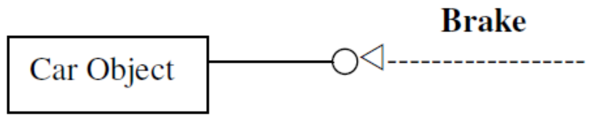
An employee object knows how to compute salary. To compute an employee salary, all that is required is to send the compute payroll message to the employee object.So the simplification of code simplifies application development and maintenance.

Objects Respond to Messages:

The capability of an object's is determined by the methods defined for it. To do an operation, a message is sent to an object. Objects represented to messages according to the methods defined in its class.

For example:

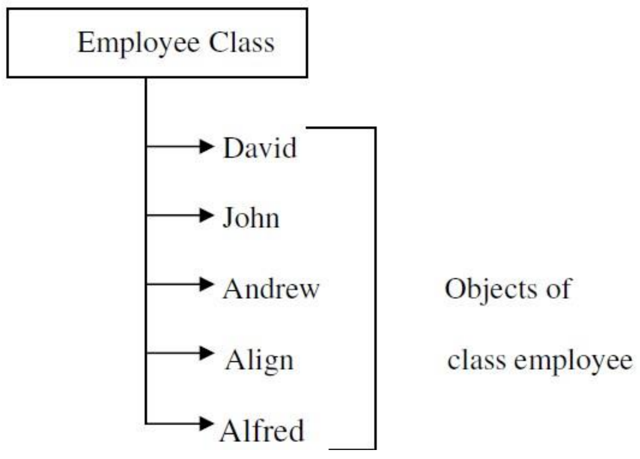
When we press on the brake pedal of a car, we send a stop message to the car object. The car object knows how to respond to the stop message since brake have been designed with specialized parts such as break pads and drums precisely respond to that message.



Different object can respond to the same message in different ways. The car, motorcycle and bicycle will all respond to a stop message, but the actual operations performed are object specific.It is the receiver's responsibility to respond to a message in an appropriate manner. This gives the great deal or flexibility, since different object can respond to the same message in different ways. This is known as polymorphism.

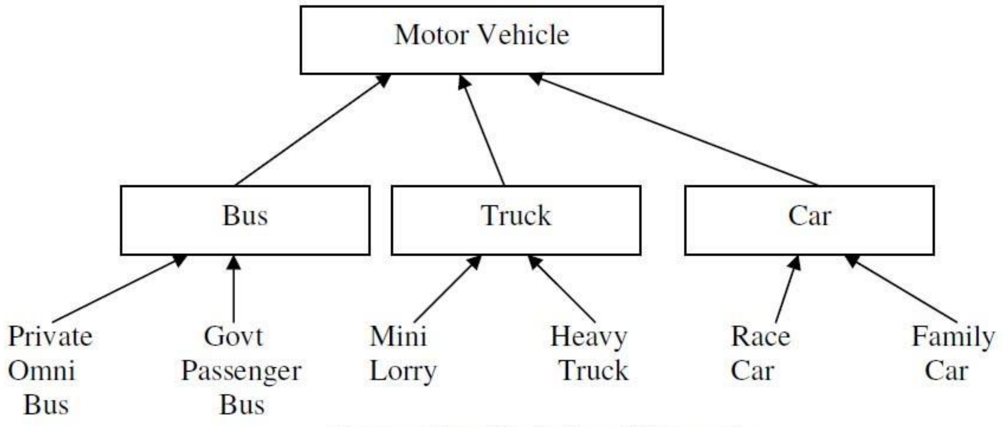
Objects are grouped in classes:The classification of objects into various classes is based its properties (states) and behaviour (methods). Classes are used to distinguish are type of object from another. An object is an instance of structures, behaviour and inheritance for objects. The chief rules are the class is to define the properties and procedures and applicability to its instances.

For example:



Class Hierarchy:

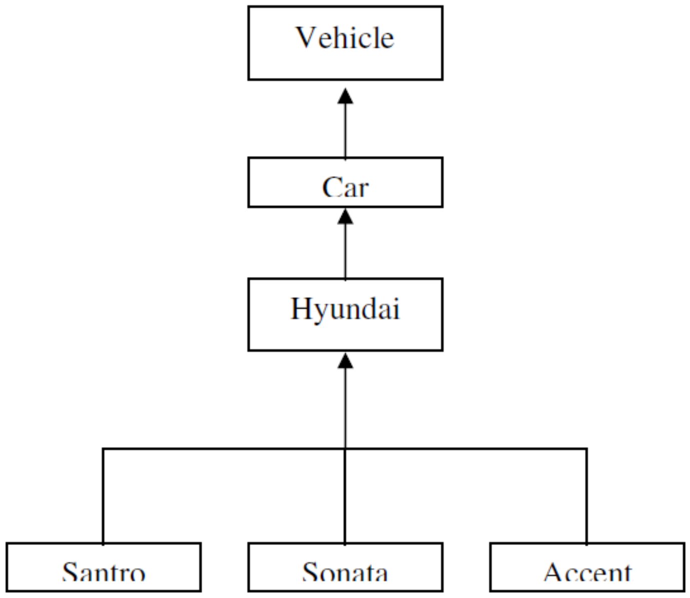
An object-oriented system organizes classes into a subclass super class hierarchy. The properties and behaviours are used as the basis for making distinctions between classes are at the top and more specific are at the bottom of the class hierarchy. The family car is the subclass of car. A subclass inherits all the properties and methods defined in its super class.



Super class/Subclass Hierarchy

Inheritance:

It is the property of object-oriented systems that allow objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The derived class holds the properties and behaviour of base class in addition to the properties and behaviour of derived class.



Inheritance allows reusability.

Dynamic Inheritance:

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, hanging base classes changes the properties and attributes of a class.

Example:

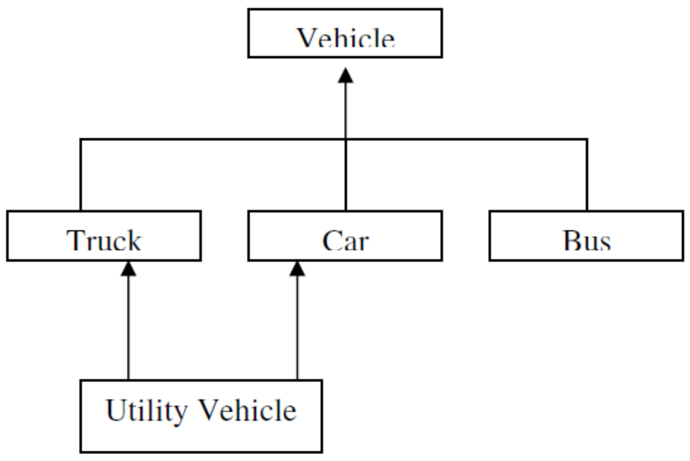
A window objects change to icon and basic again. When we double click the folder the contents will be displayed in a window and when close it, changes back to icon. It involves changing a base class between a windows class and icon class.

Multiple Inheritances:

Some object-oriented systems permit a class to inherit its state (attributes) and behaviour from more than one super class. This kind or inheritance is referred to as multiple inheritances.

For example:

Utility vehicle inherits the attributes from the Car and Truck classes.



Encapsulation and Information Hiding:

Information hiding is the principle of concealing the internal data and procedures of an object. In C++ , encapsulation protection mechanism with private, public and protected members.

In per-class protection:

Class methods can access any objects of that class and not just the receiver.

In per-object protection:

Methods can access only the receiver. An important factor in achieving encapsulation is the design at different classes of objects that operate using a common protocol. This means that many objects will respond to the message using operations tailored to its class. A car engine is an example of ncapsulation. Although engines may differ in implementation, the interface between the driver and car is through a common protocol.

Polymorphism:

Poly _ ”many”
Morph _ “form”

It means objects that can take on or assume many different forms.Polymorphism means that the same operations may behave differently on different classes. Booch defines polymorphism as the relationship of objects many different classes by some common super class. Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation detail to the objects involved.

POLYMORPHISM

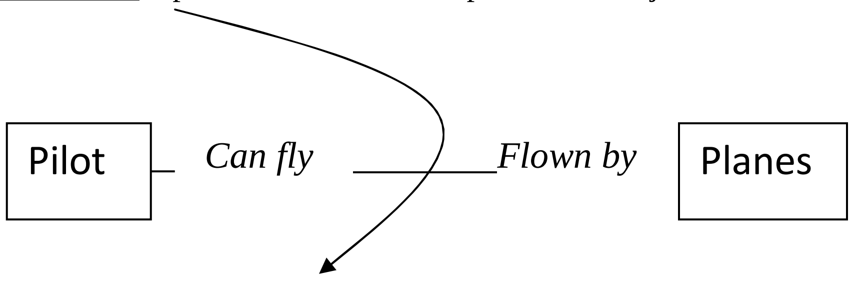
- *Poly* means “many” and *morph* means “form”, *Polymorphism* = many forms
- The same operation may behave differently on different classes
- Mechanism by which several methods can have the same name and implement the same abstract operation

*Ex : Adding two matrices is a different process from adding two integers, BUT it is still **Addition** (the message/method invoked)*

OBJECT RELATIONSHIPS & ASSOCIATIONS

- **Association** represents the relationships between objects and classes

Ex :



- associations are bidirectional ==> traversed in both directions
- important issue in association is *cardinality* ==> how many instances of one class may relate to a single instance of an associated class

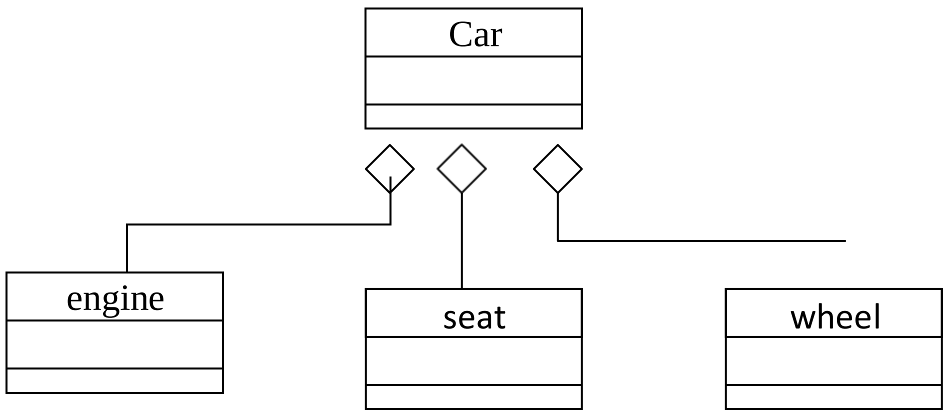
AGGREGATIONS

- All objects except the most basic ones, are composed of and may contain other objects

Ex : a spreadsheet is an object composed of cells, and cells are objects that may contain text, mathematical formulas, ect.,

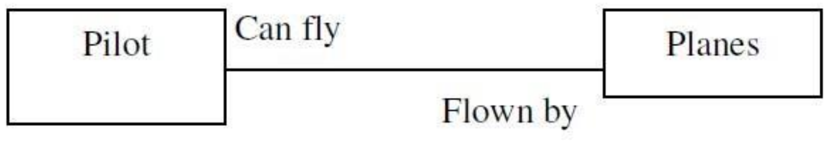
- Breaking down objects into the obejcts from which they are composed is decomposition
- Aggregation □ where an attribute can be an object itself

Ex : A car object is an agrregation of engine, seat, wheels and other objects



Object Relationship and associations:

Association represents the relationships between objects and classes. Associations are bidirectional. The directions implied by the name are the forward direction and the opposite is the inverse direction.



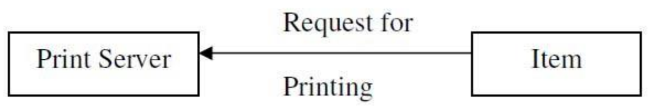
A pilot “can fly” planes. The inverse of can fly is “is flown by “. Plane “is flown by” pilot

Cardinality:
It specifies how many instances of one class may relate to a single instance of an associated class. Cardinality constrains the number of related objects and often is described as being “one” or “many”.

Consumer-producer association:

A special form or association is a consumer-producer relationship, also known as a client-server association (or) a use relationship. It can be viewed as one-way interaction. One object requests the service or another object. The object that makes the request is the consumer or client and the object that receives the request and provides the service is the producer (or) server

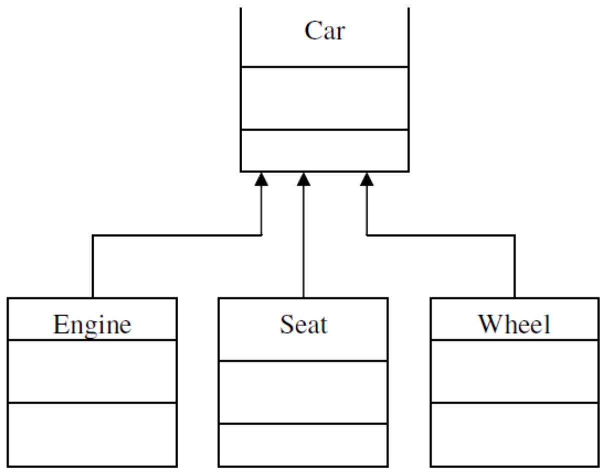
Example:



The consumer-producer association we have a print object that prints the consumer object. The print producer provides the ability to print other objects.

Aggregations:

All objects, except the most basic ones, are composed of and may contain other objects. Breaking down objects in to the objects from which they are composed is de composition. This is possible because an objects attributes need not be simple data fields, attributes can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as aggregation. The car object is an aggregation of other objects such as engine, seat and wheel objects.



Static and Dynamic Binding:

Determining which function has to be involved at compile time is called static binding. Static binding optimized the calls. (Ex) function call. The process of determining at run time which functions to involve is termed dynamic binding. Dynamic binding occurs when polymorphic call is issued. It allows some method invocation decision to be deferred until the information is known.

Example:

Cut operation in a edit submenu. It pass the cut operation to any object on the desktop, each or which handles the message in its own way.

Object Persistence:

Objects have a lifetime. They are explicitly created and can exist for a period of time that has been the duration of the process in which they were created. A file or database can provide support for objects having a longer lifeline, longer than the duration of the process for which they are created. This characteristic is called object persistence.

Meta-Classes:

In an object-oriented system every thing is an object, what about a class? Is a class an object?. Yes, a class is an object. So, If it is an object, it must belong to a class, such a class belong to a class called a meta-class (or) class or classes.

CASE STUDY: A PAYROLL PROGRAM

Consider a payroll program that processes employee records at a small manufacturing firm. The company has several classes of employees with particular payroll requirements and rules for processing each. This company has three types of employees:

1. *Managers* receive a regular salary.
2. *Office Workers* receive an hourly wage and are eligible for overtime after 40 hours.
3. *Production Workers* are paid according to a piece rate.

We will walk through traditional and object-oriented system development approaches to highlight their similarities and differences with an eye on object-oriented concepts.[1] The main focus of this exercise is to better understand the object-oriented approach. To keep the discussion simple, many issues (such as data flow diagrams, entity-relationship diagrams, feasibility analysis, and documentation) will not be addressed here.

Structured Approach

The traditional structured analysis/structured design (SA/SD) approach relies on modeling the processes that manipulate the given input data to produce the desired output. The first few steps in SA/SD involve creation of preliminary data flow diagrams and data modeling. Data modeling is a systems development methodology concerned with the system's entities, their associations, and their activities. Data modeling is accomplished through the use of entity-relationship diagrams. The SA/SD approach encourages the top-down design (also known as *top-down decomposition* or *stepwise refinement),* characterized by moving from a general statement about the process involved in solving a problem down toward more and more detailed statements about each specific task in the process. Top-down design works well because it lets us focus on fewer details at once. It is a logical technique that encourages orderly system development and reduces the level of complexity at each stage of the design. For obvious reasons, top-down design works best when applied to problems that clearly have a hierarchical nature. Unfortunately, many real-world problems are not hierarchical. Top-down function-based design has other limitations that become apparent when developing and maintaining large systems.

Top-down design works by continually refining a problem into simpler and simpler chunks. Each chunk is analyzed and specified by itself, with little regard (if any) for the rest of the system. This, after all, is one reason why top-down design is so effective at analyzing a problem. The method works well for the initial design of a system and helps ensure that the specifications for the problem are met and solved. However, each program element is designed with only a limited set of requirements in mind. Since it is unlikely that this exact set of requirements will return in the next problem, the program's design and code are not general and reusable. Top-down design does not preclude the creation of general routines that are shared among many programs, but it does not encourage it. Indeed, the idea of combining reusable programs into a system is a bottom-up approach, quite the opposite of the top-down style [9].

Once the system modeling and analysis have been completed, we can proceed to design. During the design phase, many issues must be studied. These include user interface design (input and output), hardware and software issues such as system platform and operating systems, and data or database management issues. In addition, people and procedural issues, such as training and documentation, must be addressed.

Finally, we proceed to implementing the system using a procedural language. Most current programming languages, such as FORTRAN, COBOL, and C, are based on procedural programming. That is, the programmer tells the computer exactly how to process each piece of data, presents selections from which the user can choose, and codes an appropriate response for each choice. Today's applications are much more sophisticated and developed with more demanding requirements than in the past, which makes systems development using these tools much more difficult.

In a procedural approach such as C or COBOL, the payroll program would include conditional logic to check the employee code and compute the payroll accordingly:

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

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

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

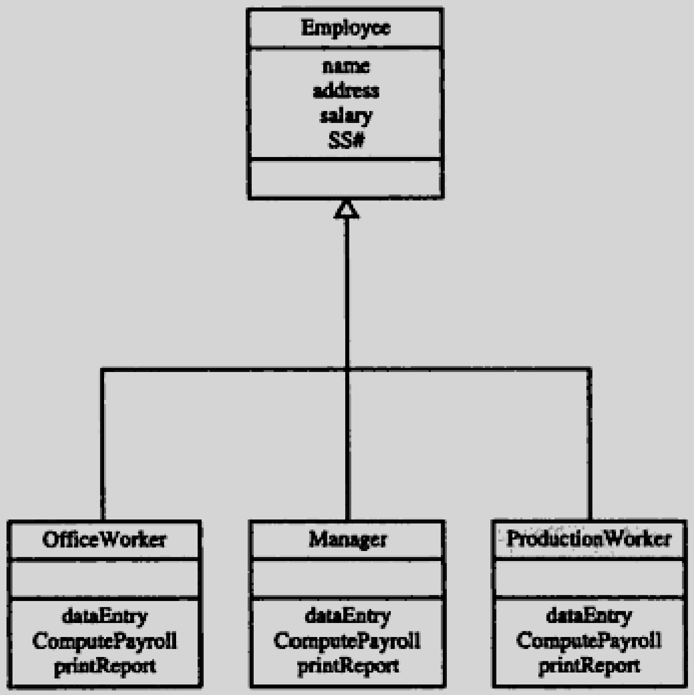


FIGURE 2-11
Class hierarchy for the payroll application.

The main program would be written in a general way that looped through all of the employees and sent a message to each employee to calculate its payroll:

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

If a new class of employee were added, a class for that type of employee would have to be created (see Figure 2-12). This class would know how to calculate its payroll. Unlike the procedural approach, the main program and other related parts of the program would not have to be modified; changes would be limited to the addition of a new class.

Object-oriented Systems Development Life Cycle:

Software development

- Analysis, design, implementation, testing & refinement to transform users' need into software solution that satisfies those needs
- Object-oriented approach
  - more rigorous process to do things right

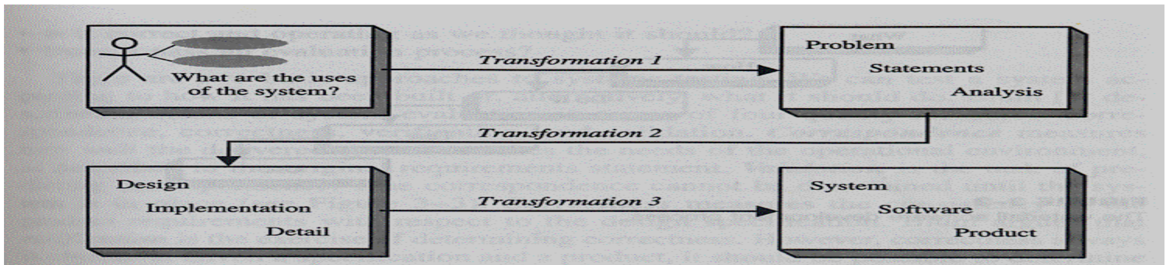

What are the uses of the system?
Transformation 1
Problem Statements Analysis
Transformation 2
Design Implementation Detail
Transformation 3
System Software Product

FIGURE 3–1
Software process reflecting transformation from needs to a software product that satisfies those needs.

- more time spent on gathering requirements, developing requirements model & analysis model, then turn into design model
- need not see code until after 25% development time

SOFTWARE DEVELOPMENT PROCESS

- Process to change, refine, transform & add to existing product
- **transformation 1(analysis)** - translates user's need into system's requirements & responsibilities.how they use system can give insight into requirements, eg: analyzing incentive payroll - capacity must be included in requirements
- **transformation 2 (design)** - begins with problem statement, ends with detailed design that can be transformed into operational system
  - bulk of development activity, include definition on how to build software, its development, its testing, design description + program + testing material
- **transformation 3 (implementation)** - refines detailed design into system deployment that will satisfy users'needs
  - takes account of equipment, procedures, resources, people, etc - how to embed software product within its operational environment, eg: new compensation method prg needs new form, gives new report.

Software process – transforming needs to software product

Waterfall Model – from 'what' to 'use'

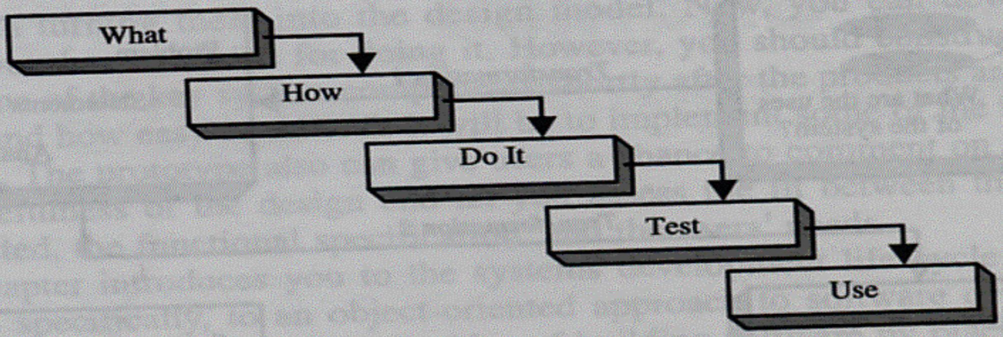


FIGURE 3–2
The waterfall software development process.

Why waterfall model fails

- When there is uncertainity regarding what's required or how it can be built
- Assumes requirements are known before design begins
  - sometimes needs experience with product before requirements can be fully understood
- Assumes requirements remain static over development cycle
  - product delivered meets delivery-time needs
- Assumes sufficient design knowledge to build product
  - best for well-understood product
  - inable to cater software special properties or partially understood issues
  - doesn't emphasize or encourage software reuse
- Problem if environment changes
  - request changes in programs

Building high quality software

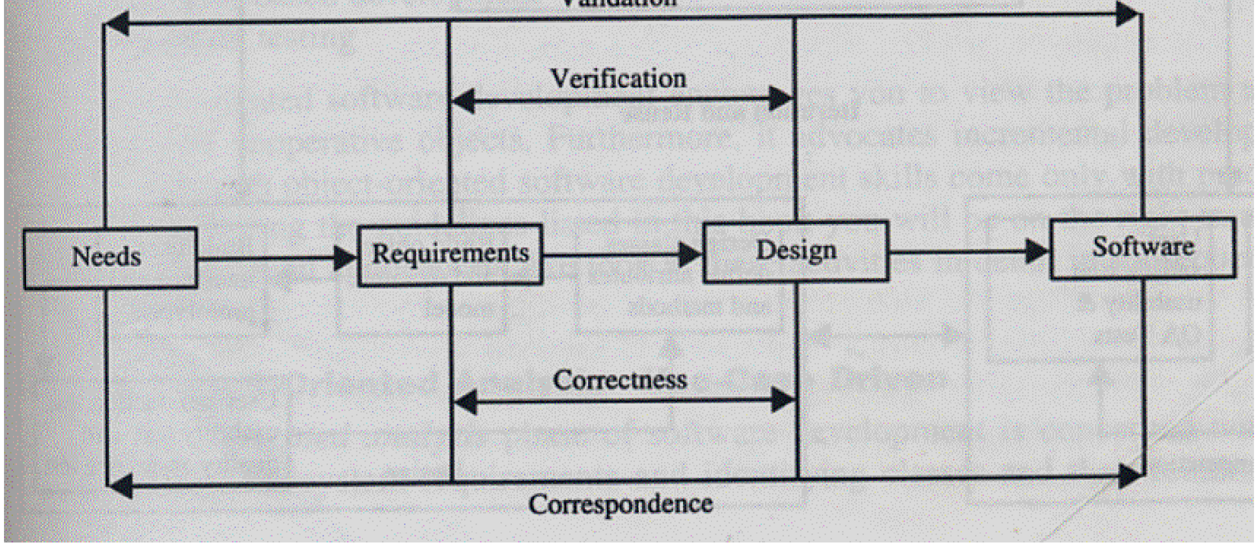
- Goal is user satisfaction
  - how do we determine system is ready for delivery
  - is it now an operational system that satisfies users'needs
  - is it correct and operating as we thought it should ?
  - Does it pass an evaluation process ?

Approaches to systems testing

- Test according to
  - how it has been built
  - what it should do
- **4 quality measures**
  - **correspondence**
    - measures how well delivered system matches needs of operational environment, as described in original requirements statement
  - **validation**
    - task of predicting correspondence (true correspondence only determined after system is in place)
  - **correctness**
    - measures consistency of product requirements with respect to design specification
  - v**erification**
    - exercise of determining correctness (correctness objective => always possible to determine if product precisely satisfies requirements of specification)

Quality Measures

FIGURE 3-3
Four quality measures (correspondence, correctness, validation, and verification) for software evaluation.



Verification vs Validation

- Verification
  - am I building the product right ?
  - Begin after specification accepted
- Validation
  - am I building the right product ?
  - Subjective - is specification appropriate ? Uncover true users' needs , therefore establish proper design ?
  - Begins as soon as project starts

- Verification & validation independent of each other
  - even if product follows spec, it may be a wrong product if specification is wrong
  - eg: report missing, initial design no longer reflect current needs
  - If specification informal, difficult to separate verification and validation

Object-oriented approach: A use-case driven approach

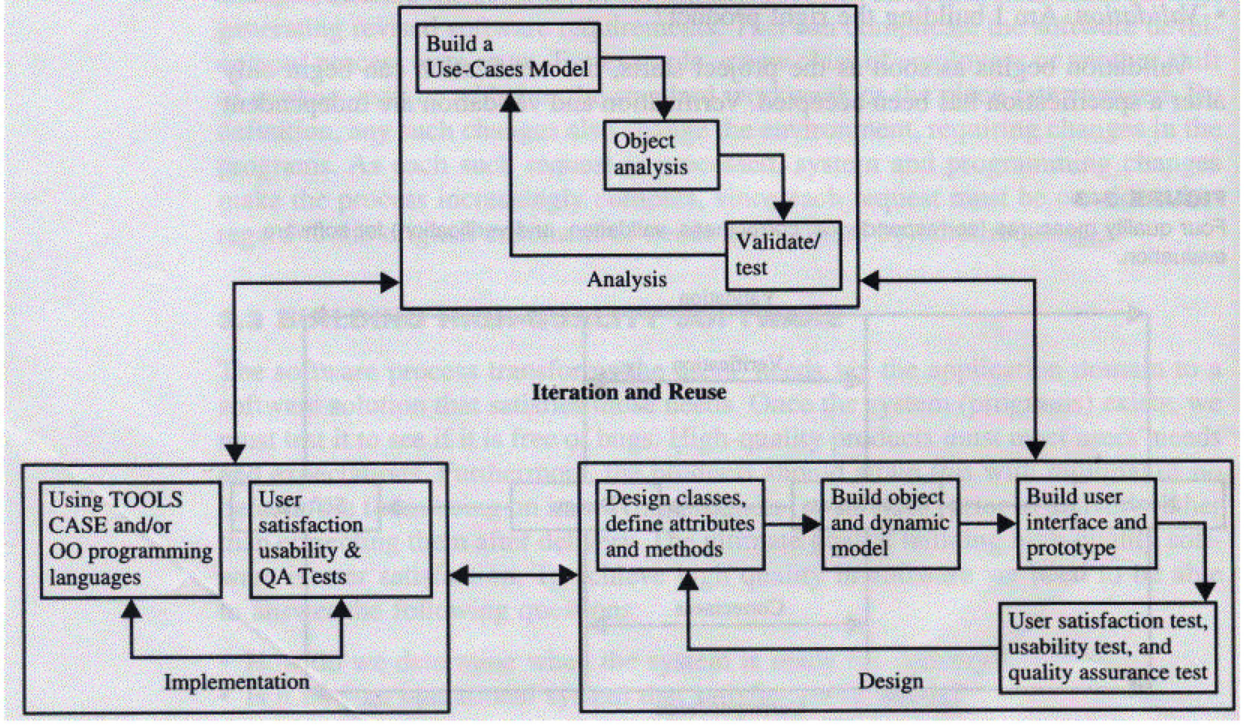
- Object-oriented software development life cycle consists of
  - Object-oriented analysis
  - Object-oriented design
  - Object-oriented implementation
- Use-case model can be employed throughout most activities of software development

- designs traceable across requirements, analysis, design, implementation & testing can be produced
- all design decisions can be traced back directly to user requirements
- usage scenarios can be test scenarios

Object-oriented Systems Development Approach

FIGURE 3–4

The object-oriented systems development approach. Object-oriented analysis corresponds to transformation 1; design to transformation 2, and implementation to transformation 3 of Figure 3–1.


Build a Use-Cases Model
Object analysis
Validate/ test
Analysis
Iteration and Reuse
Using TOOLS CASE and/or OO programming languages
User satisfaction usability & QA Tests
Implementation
Design classes, define attributes and methods
Build object and dynamic model
Build user interface and prototype
User satisfaction test, usability test, and quality assurance test
Design

Using Jacobson et al. life cycle model – traceable design across development

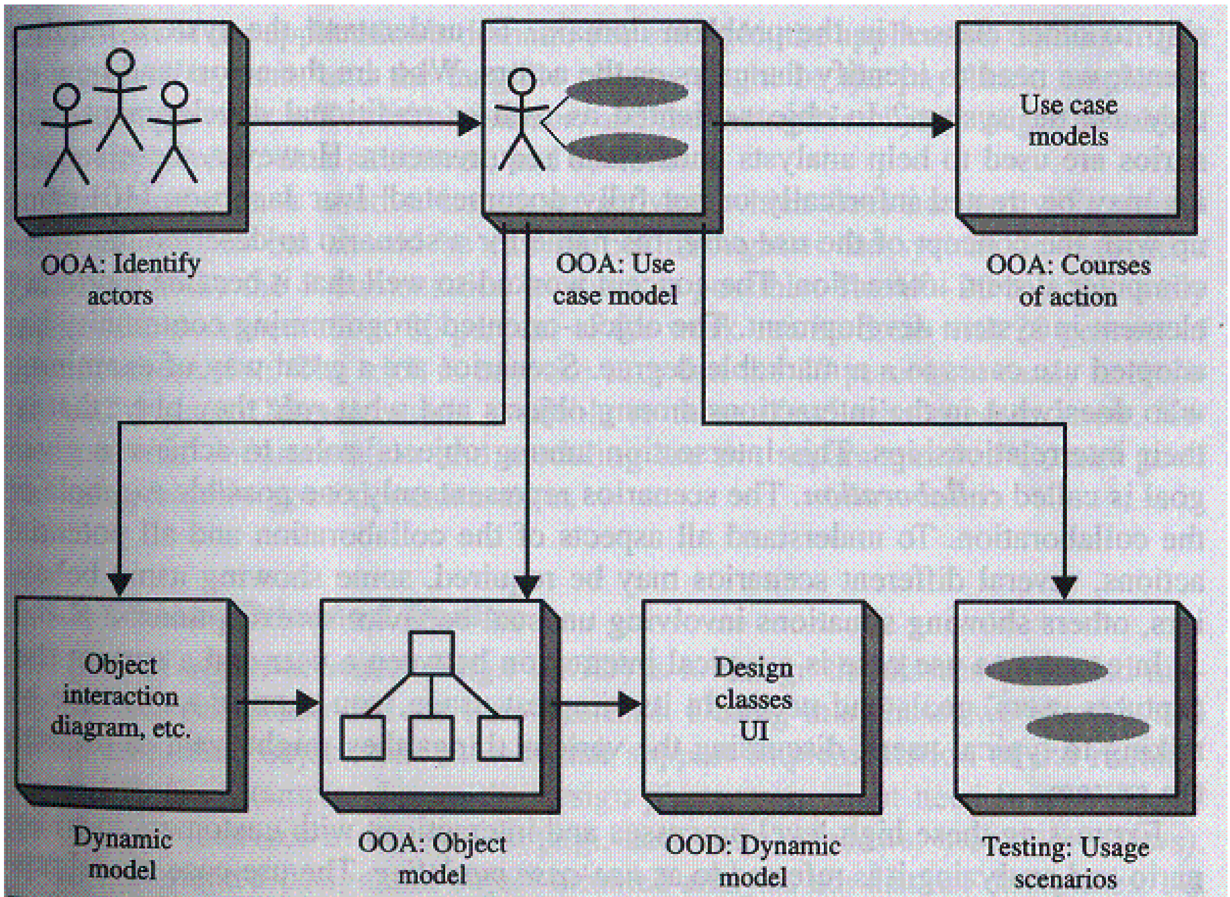


FIGURE 3–5
By following the life cycle model of Jacobson et al., we produce designs that are traceable across requirements, analysis, implementation, and testing.

Object-oriented software development

- **Activities**
    - Object-oriented analysis - use case driven
    - Object-oriented design
    - Prototyping
    - Component-based development
    - Incremental testing
- **Encourages**
    - viewing of system as a system of cooperative objects
    - incremental development

Object-oriented analysis - use-case driven

- *Use Case*, is a name for a scenario to describe the user–computer system interaction.
- Determine system requirements, identify classes & their relationship to other classes in domain
- To understand system requirements
    - need to identify the users or actors
        - who are the actors ? How do they use system ?
    - Scenarios can help (in traditional development, it is treated informally, not fully documented)
        - Jacobson introduces concept of use case - scenario to describe user-computer system interaction

Use case

- Typical interaction between user & system that captures users' goal & needs
  - In simple usage, capture use case by talking to typical users, discussing various things they might want to do with system
  - can be used to examine who does what in interactions among objects, what role they play, intersection among objects' role to achieve given goal is called collaboration
  - several scenarios (usual & unusual behaviour, exceptions) needed to understand all aspects of collaboration & all potential actions

use cased modeling

- expressing high level processes & interactions with customers in a scenario & analyzing it
- gives system uses, system responsibilities

developing use case is iterative

- when use case model better understood & developed, start identifying classes & create their relationship

Identifying objects

- What are physical objects in system ?
  - Individuals,organizations, machines, units of information, pictures, whatever makes up application/ make sense in context of real world
- objects help establish workable system
  - work iteratively between use-case & object models

    - incentive payroll - employee, supervisor, office administrator, paycheck, product made, process used to make product
- Intangible objects ?
    - Data entry screens, data structures

Documentation

- **80-20 rule**
- 80% work can be done with 20% documentation
- 20% easily accessible, 80% availbel to few who needs to know
- modeling & documentation inseparatable
    - good modeling implies good documentation

Object-oriented Design

- **Goal :** to design classes identified during analysis phase & user interface
- Identify additional objects & classes that support implementation of requirements
    - Eg. add objects for user interface to system (data entry windows, browse windows)
- Can be intertwined with analysis phase
    - Highly incremental, eg. can start with object-oriented analysis, model it, create object-oriented design, then do some more of each again & again, gradually refining & completing models of system
    - Activities & focus of oo analysis & oo design are intertwined, grown not built

Object-oriented Design

- First, build object model based on objects & relationship
- Then iterate & refine model

- Design & refine classes
- Design & refine attributes
- Design & refine methods
- Design & refine structures
- Design & refine associations

Guidelines in Object-oriented Design

- Reuse rather than build new classes
  - Know existing classes
- Design large number of simple classes rather than small number of complex classes
- Design methods
- Critique what has been proposed

– Go back & refine classes

Prototyping

- Prototype – version of software product developed in early stages of product's life cycle for specific, experimental purposes
  - Enables us to fully understand how easy/difficult it will be to implement some features of system
  - Gives users chance to comment on usability & usefulness of user interface design
  - Can assess fit between software tools selected, functional specification & user needs
  - Can further define use cases, makes use case modeling easier
    - prototype that satisfies user + documentation -> define basic courses of action for use cases covered by prototype

- Important to construct prototype of key system components shortly after products are selected
    - Pictures worth a thousand words
    - Build prototype with use-case modeling to design systems that users like & need

Prototyping: old & new

- Before: prototype thrown away when industrial strength version developed
- New trend: (eg. rapid application development) prototype refined into final product
    - Prototype used as means to test & refine user interface & increase usability of system
    - As underlying prototype design becomes more consistent with application requirements, more detail can be added to application
    - Test, evaluate & build further till all components work properly

Categories of Prototypes

Horizontal prototype

- Simulation of interface (entire interface in full-featured system)
- Contain no functionality
- Quick to implement, provide good overall feel of system

Vertical prototype

- Subset of system features with complete functionality
- Few implemented functions can be tested in great depth

Hybrid prototypes

- Major portions of interface established, features having high degree of risk are prototyped with more functionality

Analysis prototype

- Aid in exploring problem domain, used to inform user & demonstrate proof of concept
- Not used as basis of development, discarded when it has serve purpose
- Final product use prototype concepts, not code

Domain prototype

- Aid for incremental development of the ultimate software solution
- Often used as tool for staged delivery of subsystems to users/other members of development team
- Demonstrate the feasibility of implementation
- Eventually evolve into deliverable product

Developing prototypes

- Typical time from few days to few weeks
- Should be done parallel with preparation of functional spec
  - Can result in modification of spec (some problems/features only obvious after prototype built)
- Should involve representation from all user groups that will be affected by project
  - To ascertain all that the general structure of the prototype meets requirements established for overall design
- Purpose of review
  - Demo that prototype has been developed according to spec & that final spec is appropriate
  - Collect info about errors & problems in systems, eg user interface problems

    - Give management & everyone connected with project glimpse of what technology can provide
- Evaluation easier if supporting data readily available
- Testing considerations must be incorporated in design & implementation of systems

Implementation: Component-based development

- No more custom development, now assemble from prefabricated components
    - No more cars, computers, etc custom designed & built for each customer
    - Can produce large markets, low cost, high quality products
    - Cost & time reduced by building from pre-built, ready-tested components
    - Value & differentiation gained by rapid customization to targeted customers

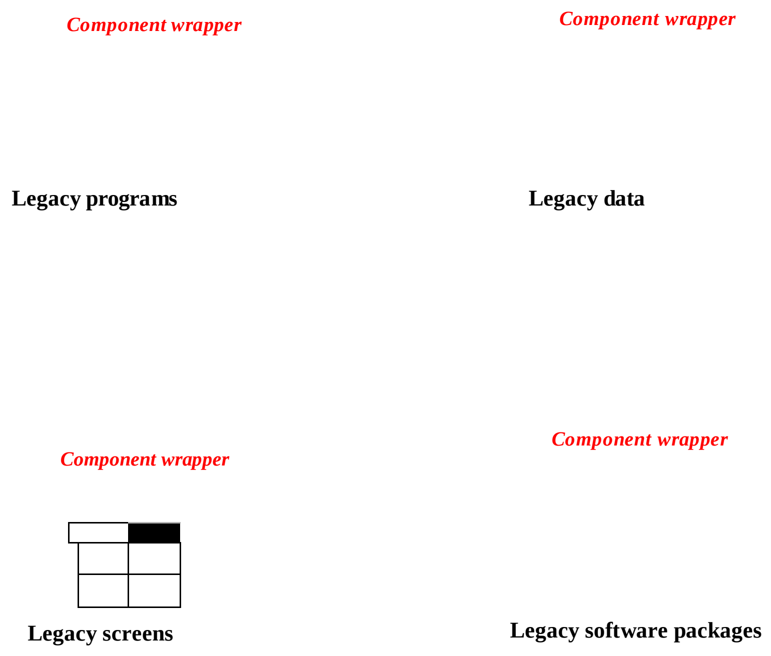
Component-based development

- Industrialised approach to system development, move form custom development to assembly of pre-built, pre-tested, reusable software components that operate with each other
    - Application development improved significantly if applications assembled quickly from prefabricated software components
    - Increasingly large collection of interpretable software components could be made available to developers in both general & specialist catalogs
- Components themselves can be constructed from other components, down to prebuilt components/old-fashioned code written in prg languages like C
- Visual tools/actual code can be used to glue together components
    - Visual glue – Digitalk’s Smalltalk PARTS, IBM VisualAge
- Less development effort, faster, increase flexibility

Application Wrapping

- Application/component wrapper
  - surrounds complete system, both code & data
  - Provides interface to interact with both new & legacy software systems
  - Off the shelf not widely available, mostly home-grown within organization
- Software component
  - Functional units of prgs, building block offering collection of reusable services
  - Can request service form another component or deliver its own services on request
  - Delivery of services independent, component work together to accomplish task
  - Components can depend on one another without interfering one another
  - Each component unaware of context/inner workings of other components
- Aspects of software development
  - OO concept – analysis, design, programming
  - Component-based – implementation, system intergration

Component wrapping technology



Rapid Application Development (RAD)

- Set of tools & techniques to build application faster than typically possible with traditional methods
- Often used with software prototyping
- Iterational development
  - Implement design & user requirements incrementally with tools like Delphi, VisualAge, Visual Basic, or PowerBuilder
  - Begins when design completed
    - Do we actually understood problem (analysis) ?
    - Does the system do what it is supposed to do (design) ?
    - Make improvement in each iteration

Incremental testing

- Software development and all of its activities including testing are an iterative process.
- Waiting until after development waste money & time
- Turning over applications to quality assurance group not helping since they are not included in initial plan

Reusability

- Major benefit of Object-oriented approach
- For objects to be reusable, much effort must be spent of designing it
    - Design reusability
- Effectively evaluate existing software components
    - Has my problem been solved ?
    - Has my problem been partially solved ?
    - What has been done before to solve problem similar to this one ?
- Need
    - detailed summary info about existing software components
    - Some kind of search mechanism
        - Define candidate object simply
        - Generate broadly/narrowly defined query

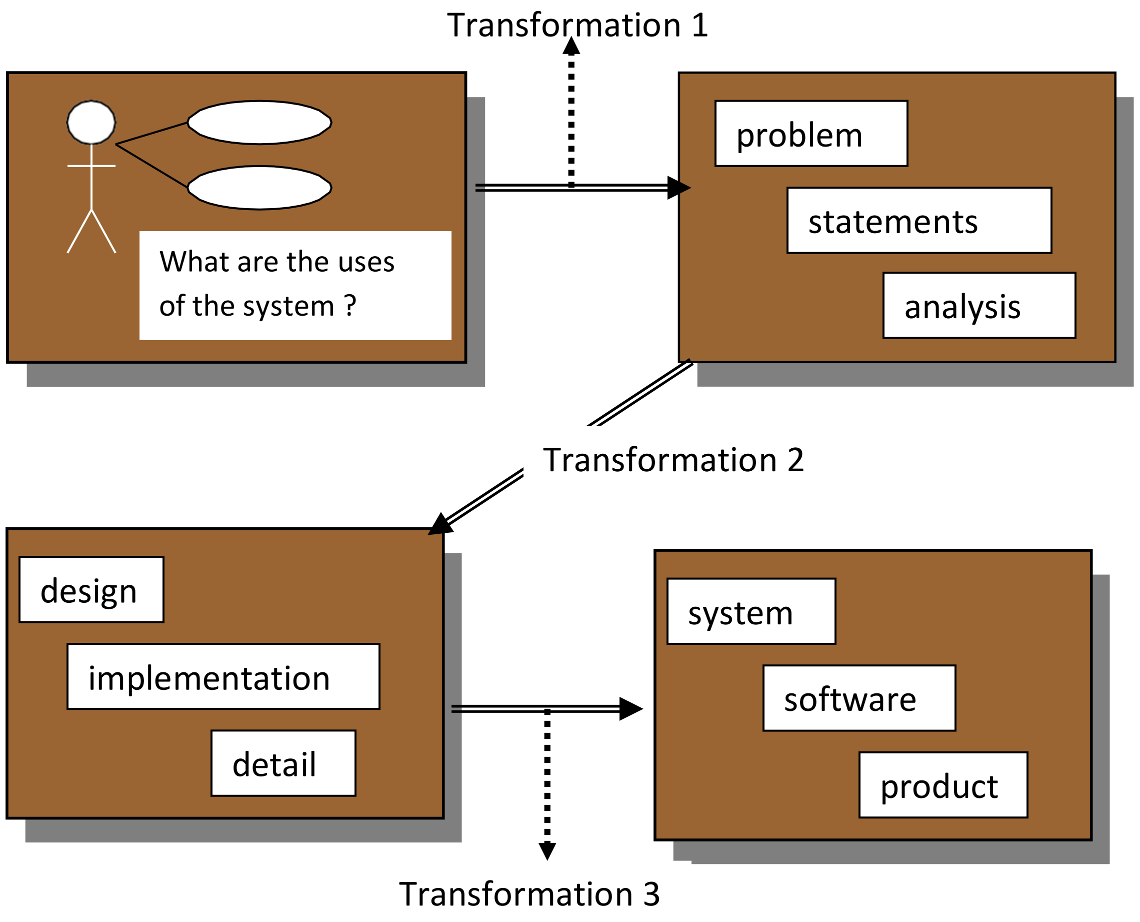
Reuse Strategy

- Information hiding
- Conformance to naming standards
- Creation & administration of an object repository
- Encouragement by strategic management of reuse as opposed to constant redevelopment
- Establish target for % of object in project to be reuse

❒ **<u>The SW Development Process</u>**

- System development can be viewed as a process
- Development = is a process of change, refinement, transformation, or addition to the existing product
- Generally, the SW development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation. Refer figure 1 :

Figure 1 : SW process reflecting transformation from needs to a SW product that satisfies those needs



    - <u>Transformation 1 (analysis)</u> – translates the user's needs into system requirements & responsibilities
    - <u>Transformation 2 (design)</u> – begins with a problem statement and ends with a detailed design that can be transformed into an operational system
    - <u>Transformation 3 (implementation)</u> – refines the detailed design into the system deployment that will satisfy the user's needs
- The emphasis on the analysis & design aspects of the SW life cycle, is intended to promote <u>*building high-quality SW*</u> (meeting the specifications and being adaptable for change)