# UNIT I

## UNIT 1 INRODUCTION

**An Overview of Object Oriented System and Development**
**Object Basic**
**Object Oriented Systems Development Life Cycle**

### An Overview of Object Oriented System and

### Development AIMS AND OBJECTIVES

The main objective of this unit is to define and understand the
- The object oriented philosophy and why it is needed
- The unified approach , methodology used to study the object oriented concepts.

### INTRODUCTION

Software development is dynamic and always undergoing major change. The methods and tools will differ significantly from those currently in use.

Today a vast number of tools and methodologies are available for systems development.

***Systems development*** refers to all activities that go into producing an information systems solution.

Systems development activities consists of
- systems analysis
- modeling,
- design
- implementation,
- testing, and
- maintenance.

A *software development methodology* is a series of processes that, if followed leads to the development of an application. The software processes describe how the work is to be carried out to achieve the original goal based on the system requirements. The software development process will continue to exist as long as the development system is in operation.

The original goal based on the system requirements. Further we study about the unified approach, which is the methodology used for learning about object oriented system development.Object-Oriented (OO) systems development is a way to develop software by building self-contained modules that can be more easily:

- Replaced
- Modified
- and Reused.

**Orthogonal View of the Software:**

Object-oriented systems development methods differ from traditional development techniques in that the traditional techniques view software as a collection of programs (or functions) and isolated data.

A program can be defined as
*Algorithms + Data Structures = Programs:*

"A software system is a set of mechanisms for performing certain action on certain data."

The main distinction between traditional system development methodologies and newer object-oriented methodologies depends on their primary focus

- traditional approach - focuses on the functions of the system
- object-oriented systems development - centers on the object, which combines data and functionality.

## OB.JECT-ORIENTED SYSTEMS DEVELOPMENT METHODOLOGY

Object-oriented development offers a different model from the traditional software development approach, which is based on functions and procedures.

In simplified terms, *object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified, and reused.*

In an object-oriented environment,

- software is a collection of discrete objects that encapsulate their data as well as the functionality to model real-world "objects."

- An object orientation yields important benefits to the practice of software construction
- Each object has attributes (data) and methods (functions).
- Objects are grouped into classes; in object-oriented terms, we discover and describe the classes involved in the problem domain.
- Everything is an object and each object is responsible for itself.

Example

Consider the Windows application needs Windows objects A Windows object is responsible for things like opening, sizing, and closing itself. Frequently, when a window displays something, that something also is an object (a chart, for example). A chart object is responsible for things like maintaining its data and labels and even for drawing itself.

Why an Oject 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.
- It adapts to
1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse

Importance of Object Orientation.

***. Higher level of abstraction***

The object-oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction. The development can proceed at the object level and ignore the rest of the system for as long as necessary. This makes designing, coding, testing, and maintaining the system much simpler.

***. Seamless transition among different phases of software development.***

The traditional approach to software development requires different styles and methodologies for each step of the process. Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds. This transition not only can slow the development process but also increases the size of the project and the chance for errors introduced in moving from one language to another.

The object-oriented approach, on the other hand, essentially uses the same language to talk about analysis, design, programming, and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.

***. Encouragement of good programming techniques.***

A class in an object-oriented system carefully delineates between its interfaces the routines and attributes within a class are held together tightly. In a properly designed system, the classes will be grouped into subsystems but remain independent; therefore, changing one class has no impact on other classes, and so, the impact is minimized. However, the object-oriented approach is not a panacea; nothing is magical here that will promote perfect design or perfect code.

***. Promotion of reusability.***

Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers (other objects). Within this framework, the class does not concern itself with the rest of the system or how it is going to be used within a particular system.

## OVERVIEW OF THE UNIFIED APPROACH

The *unified approach* (UA) is a methodology for software development that is proposed by the author, and used in this book. The UA, based on methodologies by Booch, Rumbaugh, and Jacobson, tries to combine the best practices, processes, and guidelines along with the Object Management Group's unified modeling language.

- The UA, based on methodologies by Booch, Rumbaugh, Jacobson, and others, tries to combine the best practices, processes, and guidelines.
- 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.
- UML is a set of notations and conventions used to describe and model an application.
- UA utilizes the unified modeling language (UML) which is a set of notations and conventions used to describe and model an application.

Figure 1-1 depicts the essence of the unified approach. The heart of the UA is Jacobson's use case. The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs.

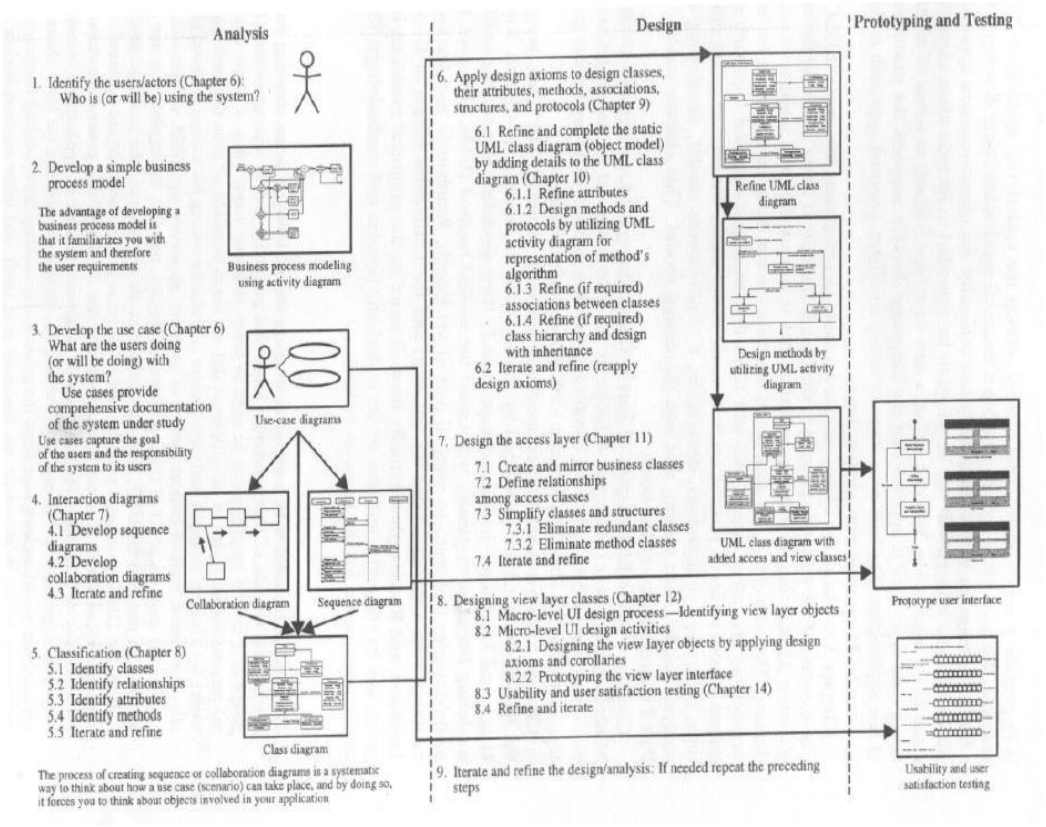



Fig 1.1 The unified approach road map.

The main advantage of an object-oriented system is that the class tree is dynamic and can grow. Your function as a developer in an object-oriented environment is to foster the growth of the class tree by defining new, more specialized classes to perform the tasks your applications require. After your first few projects, you will accumulate a repository or class library of your own, one that performs the operations your applications most often require. At that point, creating additional applications will require no more than assembling classes from the class library.

## OBJECT BASICS:

**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,
- Describe object persistence,
- Understand meta-classes.

**What is an object?**

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

• An object is an entity.

- It knows things (has attributes)
- It does things (provides services or has methods)

Example: *It Knows things (attributes)*

- I am an Employee.
- I know my name,
- social security number and
- my address.

*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 behavior.

**Object:**

- In an object-oriented system, everything is an object: numbers, arrays, records, fields, files, forms, an invoice, etc.
- An Object is anything, real or abstract, about which we store data and those methods that manipulate the data.
- Conceptually, each object is responsible for itself.
- A window object is responsible for things like opening, sizing, and closing itself.
- A chart object is responsible for things like maintaining its data and labels, and even for drawing itself.

***Two Basic Questions*** When developing an O-O application, two basic questions always arise.

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

***Traditional Approach***

• The traditional approach to software development tends toward writing a lot of code to do all the things that have to be done.

• You are the only active entity and the code is just basically a lot of building materials.

***Object-Oriented Approach*** OO approach is more like creating a lot of helpers that take on an active role, a spirit, that form a community whose interactions become the application.

***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, tow it, carry passenger in it.

***Objects are Grouped in Classes***

- A *class* is a set of objects that share a common structure and a common behavior; 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.
- The role of a class is to define the attributes and methods (the state and behavior) and applicability of its instances.
- The class car, for example, defines the property color.
- Each individual car (object) will have a value for this property, such as "maroon," "yellow" or "white."
- Each object is an instance of a class. There may be many different classes.

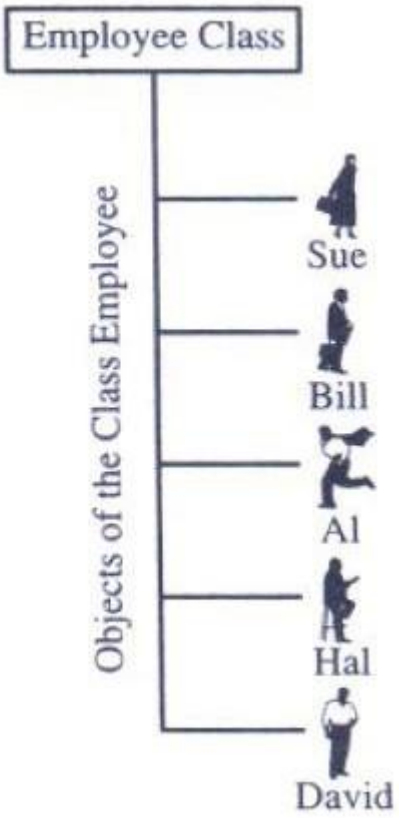



**FIGURE 1.2 Sue, Bill, AI, Hal, and David are instances or objects of the class Employee.**

**ATTRIBUTES: OBJECT STATE AND PROPERTIES**

Properties represent the state of an object. Often, we want to refer to the description of these properties rather than how they are represented in a particular programming language. In our example, the properties of a car, such as color, manufacturer, and cost, are abstract descriptions (Figure1.3).


**FIGURE 1.3 The attributes of a car object.**

We could represent each property in several ways in a programming language. For color, we could choose to use a sequence of characters such as

*red,* or the (stock) number for red paint, or a reference to a full-color video image that paints a red swatch on the screen when displayed. The importance distinction is that an object's abstract state can be independent of its physical representation.

**OB.JECTS BEHAVIOR AND METHODS**

- Behavior denotes the collection of methods that abstractly describes what an object is capable of doing.
- Each produre defines and describes a particular behavior of an object.
- The object , called the receiver, is that on which the method operates.
- Methods encapsulate the behavior of the object, provide interface to the object, and hide any of the internal strctures and states maintained by the object.
- Procedure provide us the means to communicate with an object and access its properties.
- Objects take responsibility for their own behavior.

## OBJECTS RESPOND TO MESSAGES

An object's capabilities are determined by the methods defined for it. Methods conceptually are equivalent to the function definitions used in Procedural languages. For example, a draw method would tell a chart how to draw itself.

However, to do an operation, a message is sent to an object. Objects perform operations in response to messages. For example, 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, since brakes have been designed with specialized parts such as brake pads and drums precisely to respond to that message. Sending the same *stop* message to a different object, such as a tree, however, would be meaningless an could result in an unanticipated response.

***Messages*** essentially are nonspecific function calls: We would send a *draw* message to a chart when we want the chart to draw itself. A message is different from a subroutine call, since different objects can respond to the same message in different ways. For example, cars, motorcycles, and bicycles will all respond to a *stop* message, but the actual operations performed are object specific.

In the top example, depicted in Figure 1.4, we send a *Brake* message to the *Car* object. In the middle example, we send a *multiplication* message to 5 object followed by the number by which we want to multiply 5. In the bottom example, a *Compute Payroll* message is sent to the *Employee* object, where the employee object knows how to respond to the *Payroll* message.

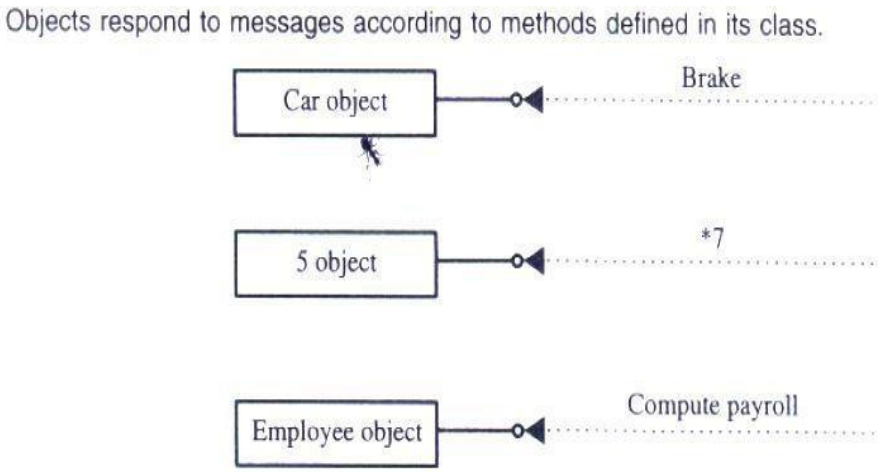



***FIG 1.4***

Polymorphism is the main difference between a message and a subroutine call. Methods are similar to functions, procedures, or subroutines in more traditional programming languages, such as COBOL, Basic, or C. The area where methods and functions differ, however, is in how they are invoked. In a Basic program, you call the subroutine (e.g., GOSUB 1000); in a C program, you call the function by name (e.g., draw chart). In an object-oriented system, you invoke a method of an object by sending an object a message. A message is much more general than a function call. It is important to understand the difference between methods and messages. Say you want to tell someone to make you French onion soup. Your instruction is the message, the way the French soup is prepared is the method and the French onion soup is the object.

**ENCAPSULATION AND INFORMATION HIDING**

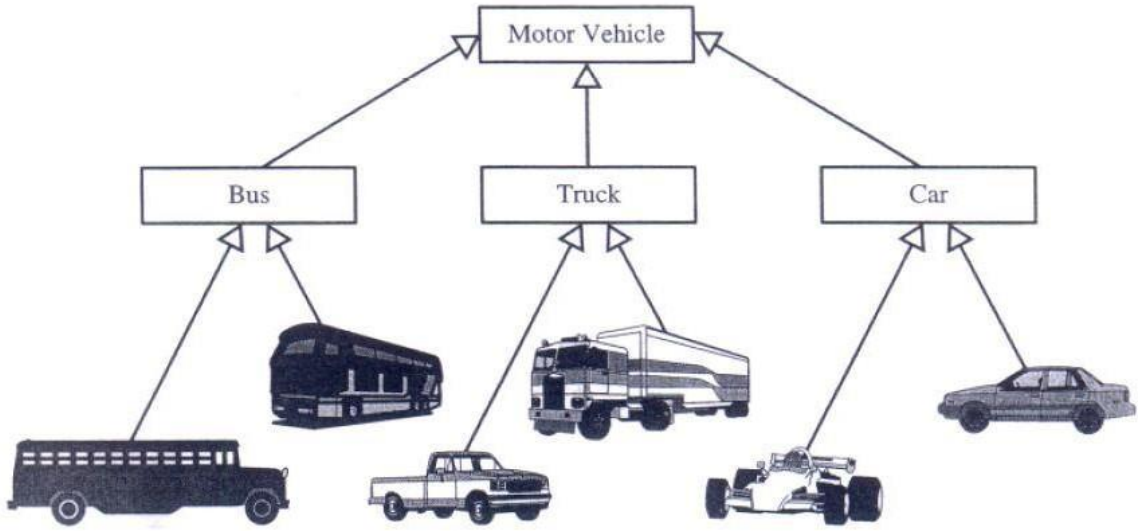
- Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings.
- As in conventional programming, some languages permit arbitrary access to objects and allow methods to be defined outside of a class.
- For example, Simula provides no protection, or information hiding, for objects, meaning that an object's data, or *instance variables*, may be accessed wherever visible.
- However, most object-oriented languages provide a well-defined interface to their objects through classes. For example, C++ has a very general *encapsulation* protection mechanism with public, private, and protected members.
- Public members (member data and member functions) may be accessed from anywhere. For instance, the *compute Payroll* method of an employee object will be public.

- Private members are accessible only from within a class. An object data representation, such as a list or an array, usually will be private.
- Protected members can be accessed only from subclasses.
- An important factor in achieving encapsulation is the design of different classes of objects that operate using a common *protocol*, or object's user interface. This means that many objects will respond to the same message, but each will perform the message using operations tailored to its class.
- Data abstraction is a benefit of the object-oriented concept that incorporates encapsulation and polymorphism. Data are abstracted when they are shielded by a full set of methods and only those methods can access the data portion of an object.

***Class Hierarchy***

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

Superclass/subclass hierarchy.




**The family car in Figure 1.5 is a subclass of car.**

A *subclass* inherits all of the properties and methods (procedures) defined in its *super class.* in this case, we can drive a family car just as we can drive any car or, indeed, almost any motor vehicle. Subclasses generally add new methods and properties specific to that class. Subclasses may refine or constrain the state and behavior inherited from its super class. In our example, race cars only have

one occupant, the driver. In this manner, subclasses modify the attribute (number of passengers) of its super class, Car.

***Inheritance (programming by extension )***

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

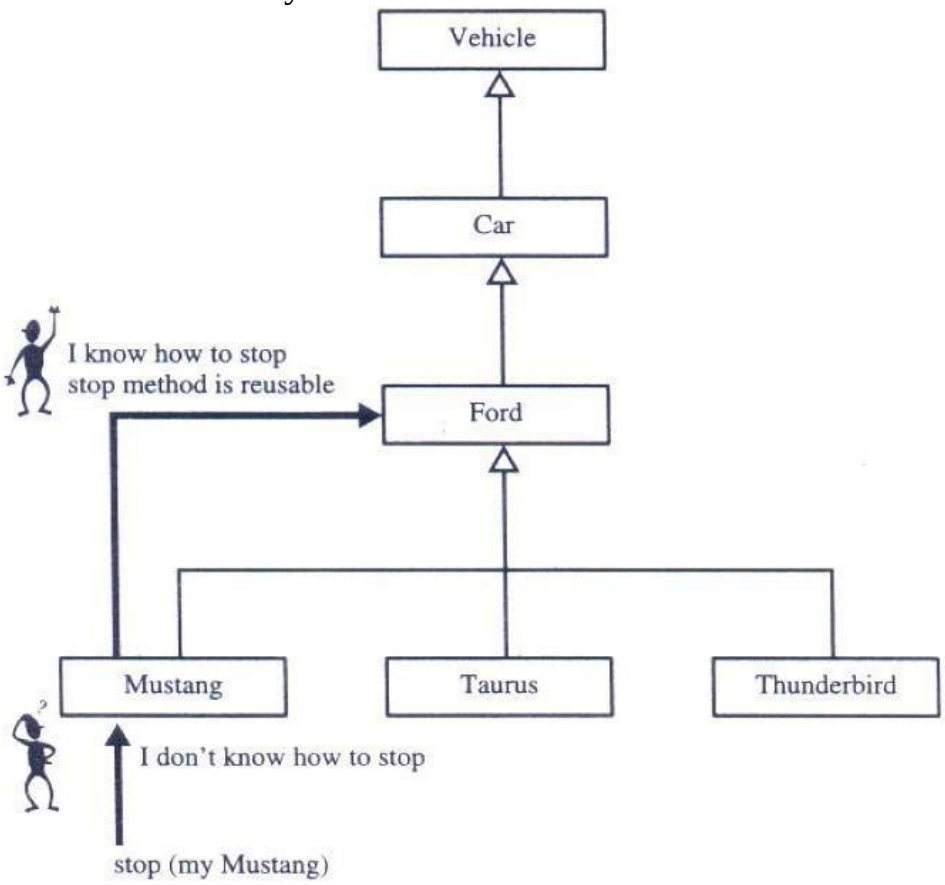



**FIGURE 1.6 INHERITANCE ALLOWS REUSABLITY**

**DYNAMIC INHERITANCE**

***Dynamic inheritance*** allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. A previous example was a Windows object changing into an icon and then back again, which involves changing a base class between a Windows class and an Icon class. More specifically, *dynamic inheritance* refers to the ability to add, delete, or change parents from objects (or classes) at run time.

In object-oriented programming languages, variables can be declared to hold or reference objects of a particular class. For example, a variable declared to reference a motor vehicle is capable of referencing a car or a truck or any subclass of motor vehicle.

**MULTIPLE INHERITANCE**

Some object-oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class. This kind of inheritance is referred to as *multiple inheritance*. For example, a utility vehicle inherits attributes from both the Car and Truck classes.

Multiple inheritance can pose some difficulties. For example, several distinct parent classes can declare a member within a multiple inheritance hierarchy. This then can become an issue of choice, particularly when several super classes define the same method. It also is more difficult to understand programs written in multiple inheritance systems. One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and then add an object of another class as an attribute.

**For example utility vehicle inherent from Car and Truck classes.**

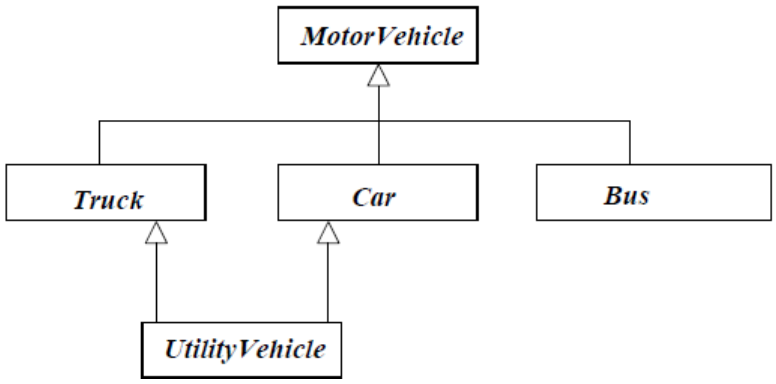



**Fig 1.7 Utility vehicle inherent from car and truck classses.**

**POLYMORPHISM**

*Poly* means "many" and *morph* means "form."

***Polymorphism*** means that the same operation may behave differently on different classes.

**Booch defines *polymorphism*** as the relationship of objects of many different classes by some common super class; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way. For example, consider how driving an automobile with a

manual transmission is different from driving a car with an automatic transmission. The manual transmission requires you to operate the clutch and the shift, so in addition to all other mechanical controls, you also need information on when to shift gears. Therefore, although driving is a behavior we perform with all cars (and all motor vehicles), the specific behavior can be different, and depending on the kind of car we are driving. A car with an automatic transmission might implement its *drive* method to use information such as current speed, engine RPM, and current gear.

Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation details to the objects involved. Since no assumption is made about the class of an object that receives a message, fewer dependencies are needed in the code and, therefore, maintenance is easier. For example, in a payroll system, manager, office worker, and production worker objects all will respond to the *compute payroll* message, but the actual operations performed are object specific.

**OBJECT RELATIONSHIPS AND ASSOCIATIONS**

**ASSOCIATIONS**

The concept of ***association*** **represents relationships between objects and classes.** For example a pilot *can fly* planes.

For example a pilot *can fly* planes

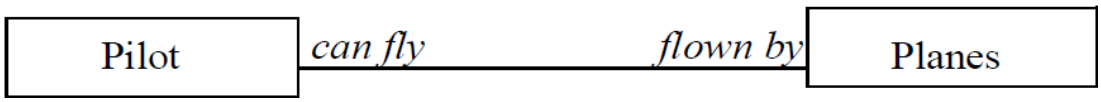



**FIGURE 1.8 Association represents the relationship among objects, which is bidirectional.**

Associations are **bidirectional**; that means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. For example, *can fly* connects a pilot to certain airplanes. The inverse of *can fly* could be called *is flown by.*

An important issue in association is ***cardinality***, which 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," Generally, the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of cylinders in an engine is four, six, or eight. Consider a client-account relationship where one client can have one or more accounts and vice versa (in case of joint accounts); here the cardinality of the client-account association is many to many.

### Consumer-Producer Association

A special form of association is a **consumer-producer** relationship, also known as a ***client-server association* or a *use relationship***. The ***consumer-producer relationship*** can be viewed as one-way interaction: One object requests the service of 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. For example, we have a print object that prints the consumer object. The print producer provides the ability to print other objects. Figure 1.9 depicts the consumerproducer association.


**FIGURE 1.9 The consumer/producer association.**

## AGGREGATIONS AND OBJECT CONTAINMENT

All objects, except the most basic ones, are composed of and may contain other objects. For example, a spreadsheet is an object composed of cells, and cells are objects that may contain text, mathematical formulas, video and so forth.

Breaking down objects into the objects from which they are composed is decomposition. This is possible because an object's attributes need not be somple 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, where an attribute can be an object itself. For example a car object is an aggregation of engine, seat, wheels and other objects.

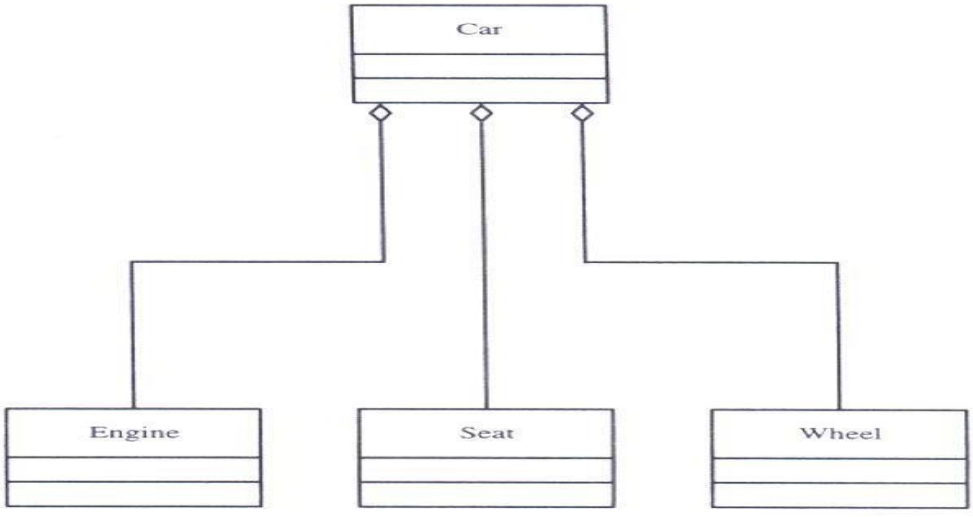



**Fig 1.10 – A car object is an aggregation of other objects such as engine, seat and wheel objects.**

***A Case Study - A Payroll Program***

Consider a payroll program that processes employee records at a small manufacturing firm. This company has three types of employees:

- *Managers*: Receive a regular salary.
- *Office Workers*: Receive an hourly wage and are eligible for overtime after 40 hours.
- *Production Workers*: Are paid according to a piece rate.

***Structured Approach***

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

**What if we add two new types of employees?**

Temporary office workers ineligible for overtime, junior production workers who receive an hourly wage plus a lower piece rate.

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

***An Object-Oriented Approach***

*What objects does the application need?*

• The goal of OO analysis is to identify objects and classes that support the problem domain and system's requirements.

• Some general candidate classes are:

• *Persons*

• *Places*

• *Things*

• ***Class Hierarchy***

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

***Class Hierarchy***

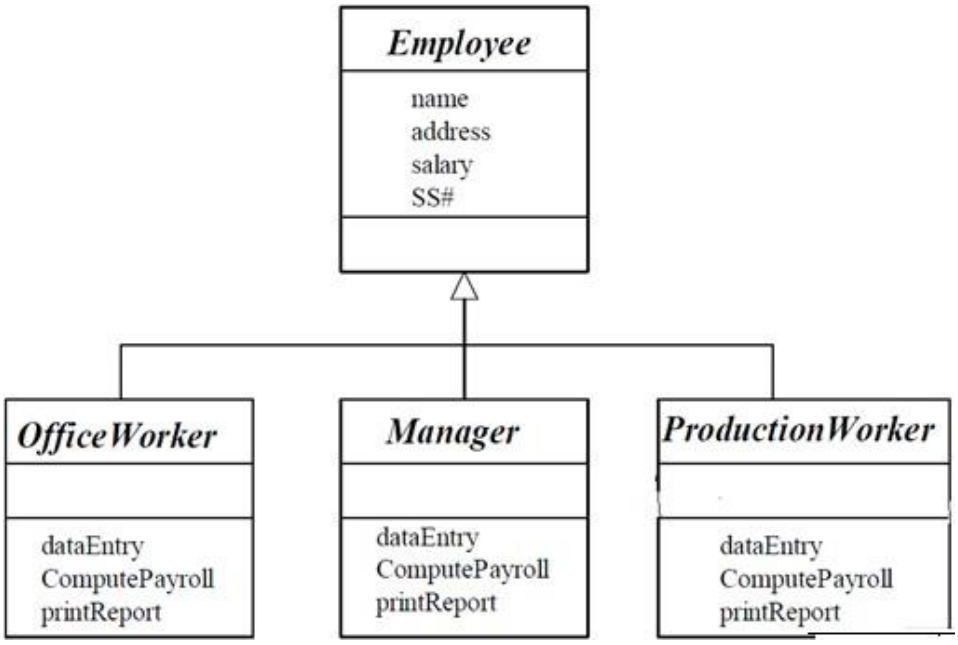



**Fig: 1.11 Class hierarchy for the payroll application**

***OO Approach***

FOR EVERY EMPLOYEE DO
BEGIN
employee computePayroll
END

## ADVANCE TOPICS
## DYNAMIC BINDING

The process of detennining (dynamically) at run time which function to invoke is termed ***dynamic binding***. Making this detennination earlier, at compile time, is called ***static binding***.

Static binding optimizes the calls; dynamic binding occurs when polymorphic calls are issued. Not all function invocations require dynamic binding.

Dynamic binding allows some method invocation decisions to be deferred until the information is known. A run-time selection of methods often is desired, and even required, in many applications, including databases and user interaction (e.g., GUIs). For example, a cut operation in an Edit submenu may pass the cut operation (along with parameters) to any object on the Desktop, each of which handles the message in its own way. If an (application) object can cut many kinds of objects, such as text and graphic objects, many overloaded cut methods, one per type of object to be cut, are available in the receiving object; the particular method being selected is based on the actual type of object being cut (which in the *GUI* case is not available until run time) .

## OBJECT PERSISTENCE

Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process in which they were *cre*ated. A file *or* a

database can provide support for objects having a longer lifeline longer than the duration of the process for which they were created.This characteristic is called O***bject Persistence.*** An object can persist beyond application session boundaries, during which the object is stored in a file or a database, in some file or database form.

***Meta-Classes***

- Everything is an object.
- How about a class?
- Is a class an object?
- Yes, a class is an object! So, if it is an object, it must belong to a class.
- Indeed, class belongs to a class called a Meta-Class or a class' class.
- Meta-class used by the compiler. For example, the meta-classes handle messages to classes, such as constructors and "new."

Rather than treat data and procedures separately, object-oriented programming packages them into "objects." O-O system provides you with the set of objects that closely reflects the underlying application. Advantages of object-oriented programming are:

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

## Object Oriented Systems Development Life Cycle

**Goals**

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

**SOFTWARE PROCESS**

The essence of the software process consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfies those needs.

**THE SOFTWARE DEVELOPMENT PROCESS**

System development can be viewed as a process. Furthermore, the development itself is a process of change, refinement, transformation, or addition to the existing product. Within the process, it is possible to replace one sub process with a new one, as long as the new sub process has the same interface as the old one, to allow it to fit into the process as a whole. With this method of change, it is possible to adapt the new process.

The process can be divided into small, interacting phases-sub processes. The sub processes must be defined in such a way that they are clearly spelled out, to allow each activity to be performed as independently of other sub processes as possible. Each sub process must have the following

- A description in terms of how it works
- Specification of the input required for the process
- Specification of the output to be produced

The software development process also can be divided into smaller, interacting sub processes. Generally, the software development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation (Figure 1.12):

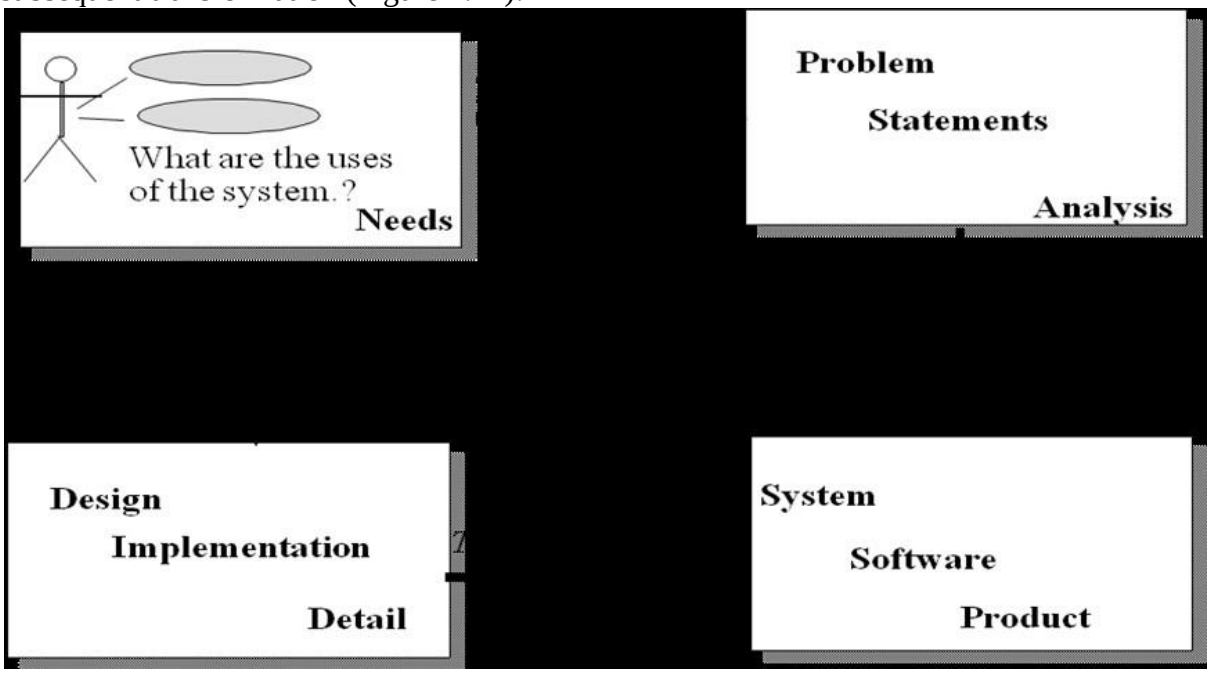



Fig 1.12: Software process reflecting transformation from needs to a software product that satisfies those needs.

***Transformation 1 (analysis)*** translates the users' needs into system requirements and responsibilities. The way they use the system can provide insight into the users' requirements. For example, one use of the system might be analyzing an incentive payroll system, which will tell us that this capacity must be included in the system requirements.

***Transformation* 2 *(design)*** begins with a problem statement and ends with a detailed design that can be transformed into an operational system. This transformation includes the bulk of the software development activity, including the definition of how to build the software, its development, and its testing. It also includes the design descriptions, the program, and the testing materials.

***Transformation* 3 *(implementation)*** refines the detailed design into the system deployment that will satisfy the users' needs. This takes into account the equipment, procedures, people, and the like. It represents embedding the software product within its operational environment. For example, the new compensation method is programmed, new forms are put to use, and new reports now can be printed.

An example of the software development process is the ***waterfall approach,*** which starts with deciding *what* is to be done (what is the problem). Once the requirements have been determined, we next must decide *how* to accomplish them. This is followed by a step in which we *do it*, whatever "it" has required us to do. We then must *test* the result to see if we have satisfied the users' requirements. Finally, we *use* what we have done (see Figure 1.13).

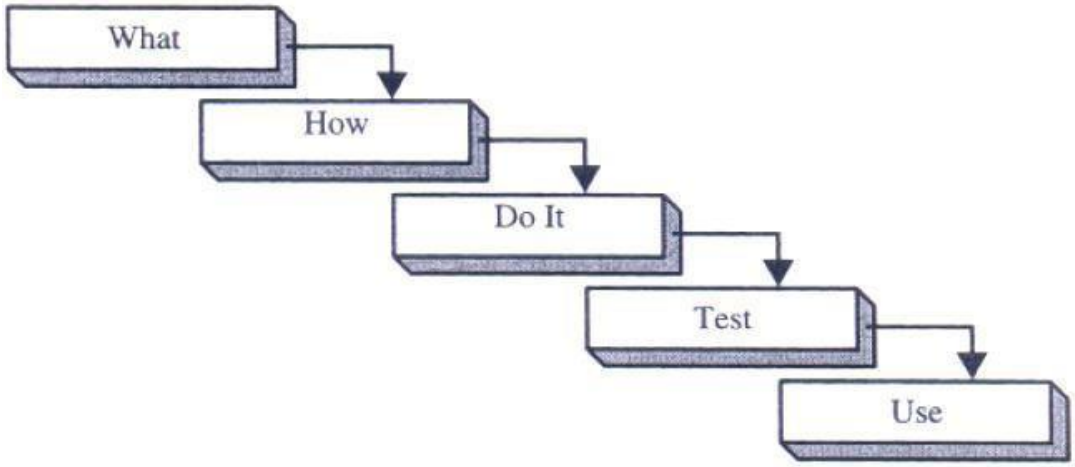



**FIGURE 1.13** The waterfall software development process.

In the real world, the problems are not always well-defined and that is why the waterfall model has limited utility. For example, if a company has experience in building accounting systems, then building another such product based on the existing design is best managed with the waterfall model, as it has been described. Where there is uncertainty regarding what is required or how it can be built, the waterfall model fails. This model assumes that the requirements are known before the design begins, but one may need experience with the product before the requirements can be fully understood. It also assumes that the requirements will remain static over the development cycle and that a product delivered months after it was specified will meet the delivery-time needs.

Finally, even when there is a clear specification, it assumes that sufficient design knowledge will be available to build the product. The waterfall model is the best way to manage a project with a well-understood product, especially very large projects. Clearly,it is based on well-established engineering principles. However, its failures can be tracedto its inability to accommodate software's special properties and its inappropriateness for resolving partially understood issues; furthermore, it neither emphasizes nor encourages software reusability.

After the system is installed in the real world, the environment frequently changes, altering the accuracy of the original problem statement and, consequently, generating revised software requirements. This can complicate the software development process even more. For example, a new class of employees or another shift of workers may be added or the standard workweek or the piece rate changed. By definition, any such changes also change the environment, requiring changes in the programs. As each such request is processed, system and programming changes make the process increasingly complex, since each request must be considered in regard to the original statement of needs as modified by other requests.

**BUILDING HIGH-QUALITY SOFTWARE**

The software process transforms the users' needs via the application domain to a software solution that satisfies those needs. Once the system (programs) exists, we must test it to see if it is free of bugs. High-quality products must meet users' needs and expectations.

Furthermore, the products should attain this with minimal or no defects, the focus being on improving products (or services) prior to delivery rather than correcting them after delivery.

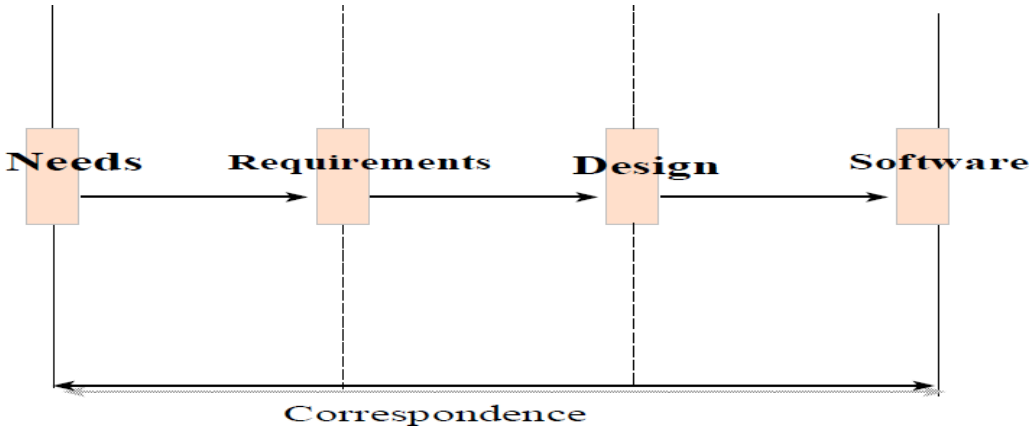
There are two basic approaches to systems testing.

- We can test a system according to how it has been built
- or, alternatively, we can test the system with respect to what it should do.

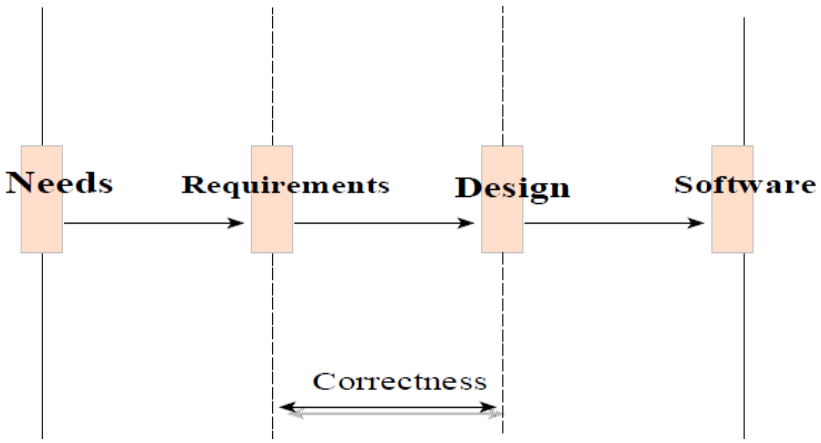
Blum describes a means of **system evaluation in terms of four quality measures:**

- **correspondence,**
- **correctness,**
- **verification,**
- **and validation.**

***Correspondence*** measures how well the delivered system matches the needs of the operational environment, as described in the original requirements statement. *It cannot be determined until the system is in place.*

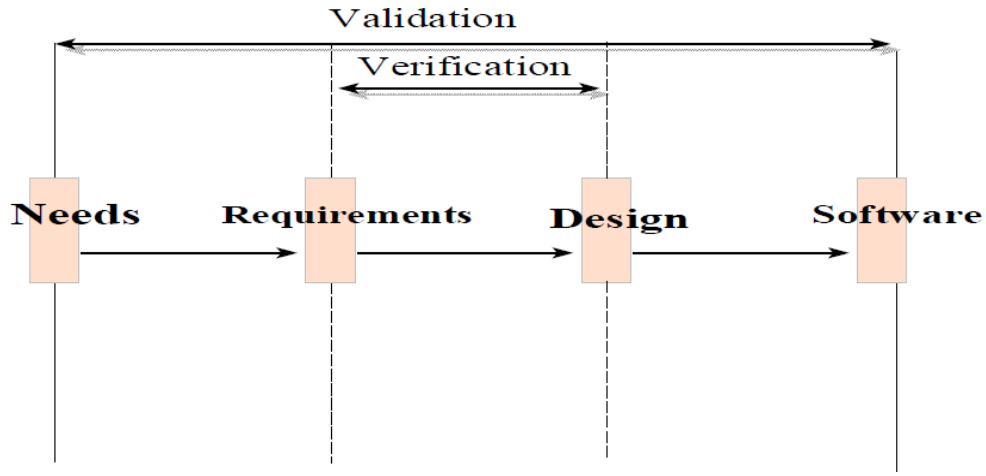



**Correctness** measures the consistency of the product requirements with respect to the design specification.




***Verification*** **is to predict the correctness.** However, correctness always is objective. Given a specification and a product, it should be possible to determine if the product precisely satisfies the requirements of the specification.

***Validation*** is to predict the correspondence. True correspondence cannot be determined until the system is in place.




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

**OB.JECT ORIENTED SYSTEMS DEVELOPMENT: A USE-CASE DRIVEN APPROACH**

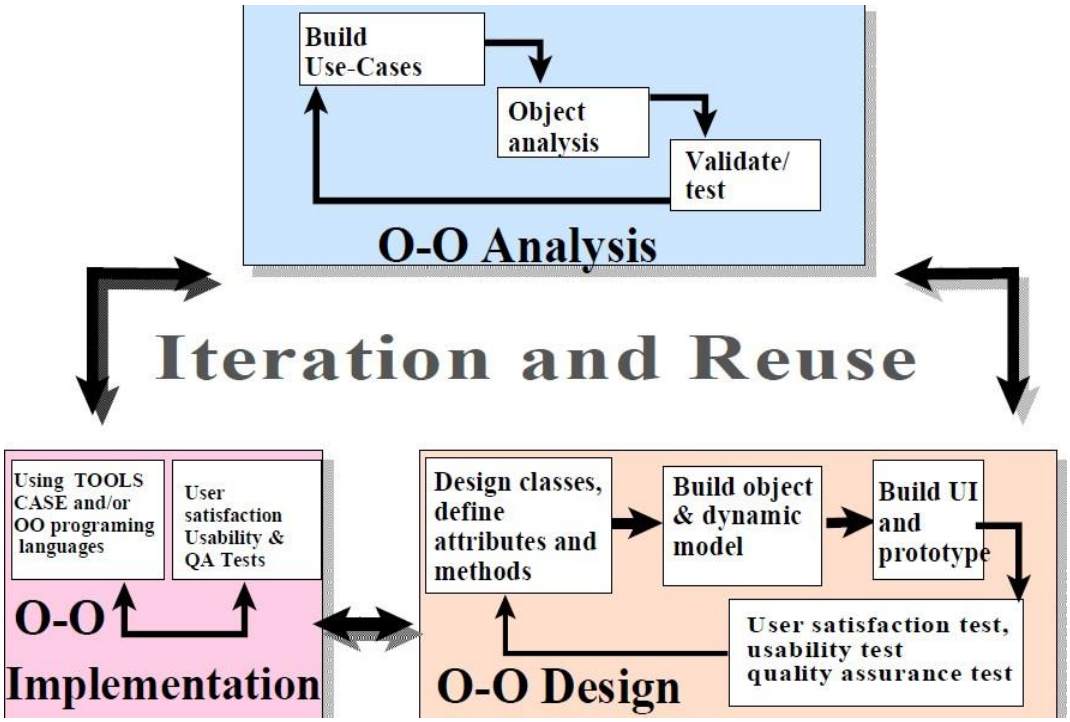



**FIGURE 1.14** The object-oriented systems development approach. Object- oriented analysis corresponds to transformation 1;design to transformation 2, and implementation to transformation 3 of Figure 1.13.

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

By following the life cycle model of Jacobson, Ericsson, and Jacobson one can produce designs that are traceable across requirements, analysis, design, implementation, and testing (as shown in Figure 1.15). The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can become test scenarios.

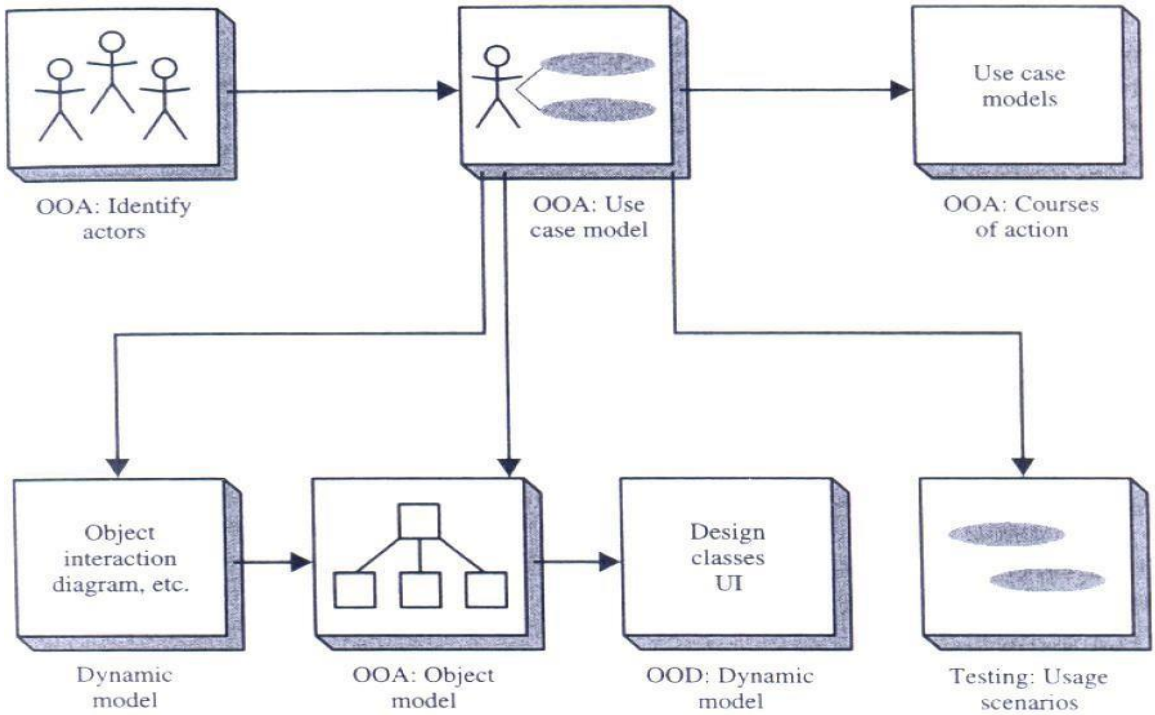



Fig 1.15: By following the life cycle model of Jacobson et aI., we produce designs that are traceable across requirements, analysis, implementation, and testing.

***Object-Oriented Systems Development activities***

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

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

*The object-oriented analysis* phase of software development is concerned with determining the system requirements and identifying classes and their re1ationship to other classes in the problem domain.

To understand the system requirements, we need to identify the users or the actors. Who are the actors and how do they use the system? In object-oriented as well as traditional development, scenarios are used to help analysts understand requirements.

Scenarios are a great way of examining who does what in the interactions among objects and what *role* they play; that is, their interrelationships. This intersection among objects' roles to achieve a given goal is called *collaboration.*

Expressing these high-level processes and interactions with customers in a scenario and analyzing it is referred to as *use-case modeling.* The use-case model represents the users' view of the system or users' needs.

This process of developing uses cases, like other object-oriented activities, is iterative-once your use-case model is better understood and developed you should start to identify classes and create their relationships.

***Object-Oriented Analysis***

OO analysis concerns with determining the system requirements and identifying classes and their relationships that make up an application.

***Object-Oriented Design***

The goal of object-oriented design (OOD) is to design

• The classes identified during the analysis phase,

• The user interface and

• Data access.

Object-oriented design and object-oriented analysis are distinct disciplines, but they can be intertwined. Object-oriented development is highly incremental; in other words, you start with object-oriented analysis, model it, create an object-oriented design, then do some more of each, again and again, gradually refining and completing models ofthe system. The activities and focus of object-oriented analysis and object-oriented design are intertwined-grown, not built.

First, build the object model based on objects and their relationships, then iterate and refine the model:

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

Here are a few guidelines to use in your design:

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

**PROTOTYPING**

- A prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes.
- A Prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system.
- It can also give users a chance to comment on the usability and usefulness of the design.
- The main idea here is to build a prototype with uses-case modeling to design systems that users like and need.

Prototyping provides the developer a means to test and refine the user interface and increase the usability of the system. As the underlying prototype design begins to become more consistent with the application requirements, more details can be added to the application, again with further testing, evaluation, and rebuilding, until all the application components work properly within the prototype framework

***Types of Prototypes***

- **A *horizontal prototype*** is a simulation of the interface. but contains no functionality. This has the advantages of being very quick to implement, providing a good overall feel of the system, and allowing users to evaluate the interface on the basis of their normal, expected perception of the system.
- A ***vertical prototype*** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth.
- An ***analysis prototype*** is an aid for exploring the problem domain. This class of prototype is used to inform the user and demonstrate the proof of a concept. It is not used as the basis of development, however, and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code.
- A ***domain prototype*** is an aid for the incremental development of the ultimate software solution. It often is used as a tool for the staged delivery of subsystems to the users or other members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product.

The purpose of this review is threefold:

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.
2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.
3. To give management and everyone connected with the project the first (or it could be second or third. . .) glimpse of what the technology can provide. The evaluation can be performed easily if the necessary supporting data is readily available. Testing considerations must be incorporated into the design and subsequent implementation of the system.

**IMPLEMENTATION: COMPONENT-BASED DEVELOPMENT**

Today, software components are built and tested in-house, using a wide range of technologies. For example, computer-aided software engineering (CASE) tools allow their users to rapidly develop information systems. The main goal of CASE technology is the automation of the entire information system's development life cycle process using a set of integrated software tools, such as modeling, methodology, and automatic code generation. However, most often, the code generated by CASE tools is only the skeleton of an application and a lot needs to be filled in by programming by hand.

A new generation of CASE tools is beginning to support component-based development.

***Component-based development*** **(CBD)** is an industrialized approach to the software development process. Application development moves from custom development to assembly of prebuilt, pretested, reusable software components that operate with each other. **Two basic ideas** underlie component-based development.

- First, the application development can be improved significantly if applications can be assembled quickly from prefabricated software components.
- Second, an increasingly large collection of interpretable software components could be made available to developers in both general and specialist catalogs.

Put together, these two ideas move application development from a craft activity to an industrial process fit to meet the needs of modern, highly dynamic, competitive, global

businesses. The industrialization of application development is akin to similar transformations that occurred in other human endeavors.

A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of prebuilt components or old-fashioned code written in a language such as C, assembler, or COBOL.

CBD will allow independently developed applications to work together and do so more efficiently and with less development effort.

Existing (legacy) applications support critical services within an organization and therefore cannot be thrown away. Massive rewriting from scratch is not a viable option, as most legacy applications are complex, massive, and often poorly documented. The CBD approach to legacy integration involves application wrapping, in particular component wrapping, technology.

The ***software components*** are the functional units of a program, building blocks offering a collection of reusable services. A software component can request a service from another component or deliver its own services on request. The delivery of services is independent, which means that components work together to accomplish a task. Of course, components may depend on one another without interfering with each other. Each component is unaware of the context or inner workings of the other components. In short, the object-oriented concept addresses analysis, design, and programming, whereas component-based development is concerned with the implementation and system integration aspects *of* software development.

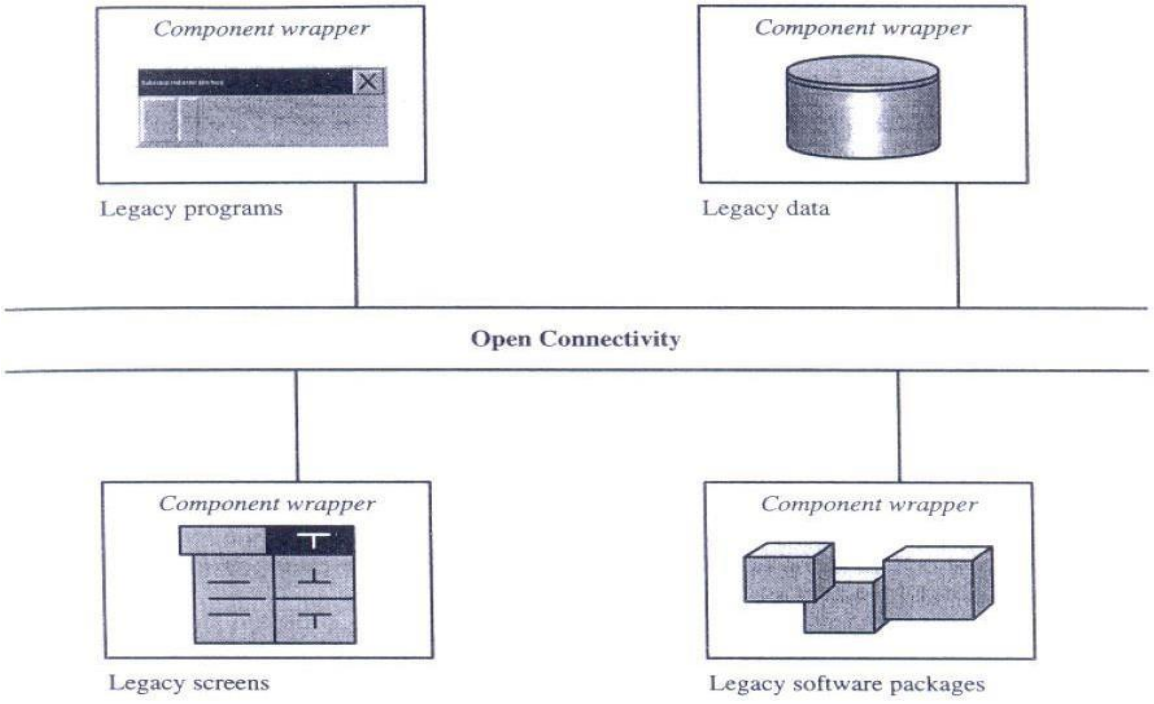



Fig. 1.16 Reusing legacy system via component wrapping technology.

***Rapid Application Development (RAD)***

RAD is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods. To achieve RAD, the developer sacrifices the quality *of* the product for a quicker delivery.

RAD is concerned primarily with reducing the "time to market," not exclusively the software development time. In fact, one successful RAD application achieved a substantial reduction in time to market but realized no significant reduction in the individual software cycles.

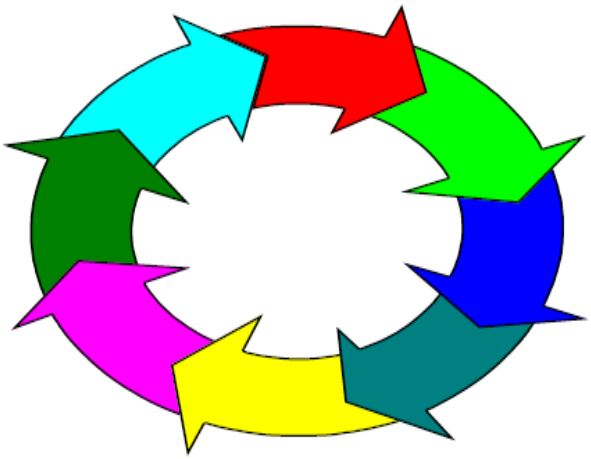
RAD does not replace SDLC but complements it, since it focuses more on process description and can be combined perfectly with the object-oriented approach. The task *of* RAD is to build the application quickly and incrementally implement the design and user requirements, through tools such as Delphi, VisualAge, Visual Basic, or PowerBuilder. After the overall design for an application has been completed, RAD begins.

The main objective *of* RAD is to build a version *of* an application rapidly to see whether we actually have understood the problem (analysis). Further, it determines whether the system does what it is supposed to do (design). RAD involves a number *of* iterations. Through each iteration we might understand the problem a little better make an improvement. RAD encourages the incremental development approach of "grow, do not Build" software.

***Incremental Testing***

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

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



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

***Reuse strategy***

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

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

**Important Questions**

1. What is system development methodology?
2. What are the two orthogonal views of software?
3. What is the object-oriented systems development methodology?
4. How does the O-O approach differ from the traditional top-down approach?
5. Define OOAD.
6. Describe the componetns of the unified approach.
7. What is an object?
8. Discuss object state and properties.
9. What the advantages are of object oriented development?
10. What is the difference between object's methods and properties?
11. How are classes organized in an object-oriented environment?
12. What is the difference between a method and a message?
13. What is polymorphism?
14. What is the difference between an object's methods and an object's attributes?
15. How are objects identified in an object-oriented system?
16. What is inheritance?
17. What is data abstraction?
18. Why is encapsulation important?
19. What is a protocol?
20. Why is polymorphism useful?
21. What is the lifetime of an object and how can you extend the lifetime of it?
22. What is an association?
23. Discuss about consumer producer relationship.
24. Define aggregation.
25. What is a formal class?
26. What is software development process?
27. What is waterfall SDLC?
28. What are the some of the advantages and disadvatages of the waterfall model?
29. Define verification, validation, correspondence, and correctness.
30. How is software verification different from validation?
31. What are the four quality measures for evaluating software?
32. What is prototyping? What are the various categories of prototyping?
33. Explain object oriented systems development lifecycle.
34. What are object oriented design processes?
35. What is RAD?
36. What is component based development?
37. Define object model. What are the various elements of object model?
38. What is an instance? Give an example.
39. Give a brief note on object behavior
40. Define object persistence.
41. Briefly explain about relationships and association of classes.
42. Define class hierarchy.
43. Write about static and dynamic binding?
44. Define meta-classes?
45. What do you mean by software development process?
46. Define Prototype. Give the types of prototype.
47. What is use case modelling?
48. What is object modelling?

49. Why is CBD important?
50. Why is reusability important? How does OO software development promote reusability?

PART – B

1. Describe the various Object oriented concepts?
2. Describe the Software Development process.
3. Explain the object oriented software development life cycle.
4. How can we build a high quality Software?
5. Explain briefly the waterfall approach.
6. Describe class hierarchy and inheritance
7. Discuss briefly about the prototype and its types
8. Write short notes on
    a. RAD
    b. CBD
9. Describe in detail about objects.
10. Compare and Contrast Traditional development methodologies and Object Oriented System.