# Object Orientation and Its Concepts

Object Orientation is about viewing and modeling systems as a set of interacting and interrelated objects. Object-oriented systems focus on capturing the structure and behavior of information systems in little modules that encompass both data and process. These little modules are known as objects. The basic concepts of object-orientation include.

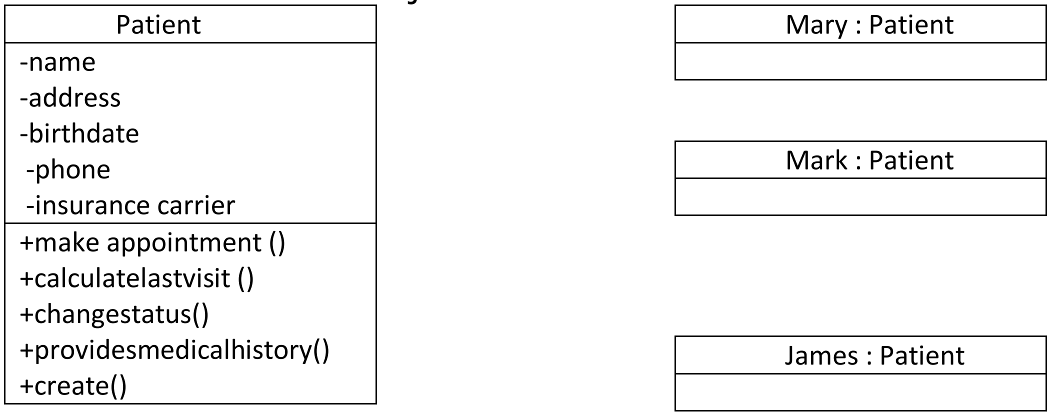
## Classes and Objects

A class is the general template we use to define and create specific instances, or objects. Every object is associated with a class. For example, all the objects that capture information about patients could fall into a class called **Patien**t, because there are attributes (e.g., name, address, birth date, phone, and insurance carrier) and methods (e.g., make appointment, calculate last visit, change status, and provide medical history) that all patients share.

An object is an instantiation of a class. In other words, an object is a person, place, or thing about which we want to capture information. If we were building an appointment system for a doctor's office, classes might include Doctor, Patient, and Appointment. The specific patients, such as Mary, James, and Marks are considered ***instances, or objects***, of the patient class

*Thus a class is defined a set of objects that share the same attributes, operations, relationships and semantics*

In UML a class and an object are shown as follows.



**class** **objects**

Each object has ***attributes*** that describe information about the object, such as a patient's name, birth date, address, and phone number. Attributes are also used to represent *relationships* between objects; for example, there could

be a department attribute in an employee object with a value of a department object that captures in which department the employee object works.
The state of an object is defined by the *value of its attributes* and its relationships with other objects at a particular point in time. For example, a patient might have a state of new or current or former.

Each object also has ***behaviors*.** The behaviors *specify what the object can do (observable effects of an operation/method)*. For example, an appointment object can probably schedule a new appointment, delete an appointment, and locate the next available appointment. In object-oriented programming, ***behaviors are implemented as methods.***

***NB: Visibility***
To specify the visibility of a class member (i.e. any attribute or method), these notations must be placed before the member's name.

| | |
|---|---|
| + | Public |
| - | Private |
| # | Protected |

## Methods/Operations and Messages

Methods implement an **object's behavior**. An operation is sometimes called a method although strictly speaking the method is limited to the implementation of the operation and does not include the signature of the operation.
The ***signature*** of the operation provides the name, parameters and the return type. *The operation in a sense is the sum of the signature and the method, while the method is the code part that that describes logic required to achieve the behavior promised by the operation.* Messages are information sent to objects to trigger operations.
A **message i**s essentially a function or procedure call from one object to another object. (Request for an execution of a procedure)
*For example, if a patient is new to the doctor's office, the receptionist sends a create message to the application. The patient class receives the create message and executes its* ***create ()*** *method which then creates a new object: aPatient*
***NB***
*The UML uses the term operation to mean the specification of an action; the term method is used to refer to the implementation of an operation*

**Object Identity**

*Property of an object that makes it distinct even if its state is the same as that of another object*. Each Patient object or Customer object needs to be uniquely identified. Two patients might have the same name thus the patient name is unlikely to distinguish the patients. an information system needs to observe that each object has a distinctive identity.

In some implementations this is accomplished by using an attribute that serve to distinguish the objects. In others a system generated identifier may be used.

E.g. in OODB Each time an object is created, a unique OID (object identifier) is added to the OODBMS (object-oriented database management system) *identifier table*. The OID is independent of the value of an object or any data contained in the object. The value of OID is not visible to the external user but used internally by the system to uniquely identify the object

**Encapsulation and Information Hiding**

Encapsulation is simply the combination of process and data into a single entity. Information hiding is the protection against unauthorized access to the internal state of an object via a uniquely defined interface
The principle of information hiding suggests that only the information required to use a software module be published to the user of the module. Typically, this implies that the information required to be passed to the module and the information returned from the module are published. Exactly how the module implements the required functionality is not relevant. We really do not care how the object performs its functions, as long as the functions occur.

Notice how the message ***create* ()** (in the previous section) is sent to an object, yet the internal algorithms needed to respond to the message are hidden from other parts of the system. The only information that an object needs to know is the set of operations, or methods, that other objects can perform and what messages need to be sent to trigger them.

**Generalization/Inheritance**
-Generalization is a relationship between a more general (parent, super) class and a more specific (child, subclass) class; the more specific class has additional attributes and operations
.
-Inheritance is the mechanism by which the more specific class acquires the attributes and operations of the more general class

The data modeling literature suggests using inheritance to identify higher-level, or more general, classes of objects. Common sets of attributes and methods can be organized into superclasses. Typically, classes are arranged in a hierarchy whereby the superclasses, or general classes, are at the top and the subclasses, or specific classes, are at the bottom. In Figure below, Person is a superclass to the classes Doctor and Patient.

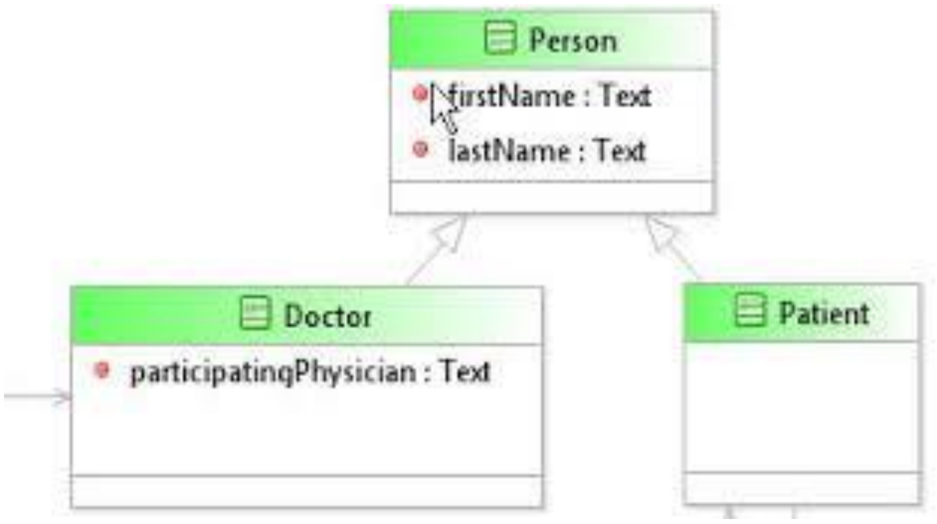
The relationship between the class and its superclass is known as the ***a-kind-of*** OR ***is a-type-of*** relationship. For example, a Doctor, which is a-kind-of Person.

Subclasses inherit the appropriate attributes and methods from the superclasses above them. That is, each subclass contains attributes and methods from its parent superclass.

Inheritance makes it simpler to define classes. Instead of repeating the attributes and methods in the Doctor and Patient classes separately, the attributes and methods that are common to both are placed in the Person class and inherited by the classes below it.

It indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and the superclass is considered a Generalization of the subclass.

The ***UML graphical representation of a Generalization*** is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes.



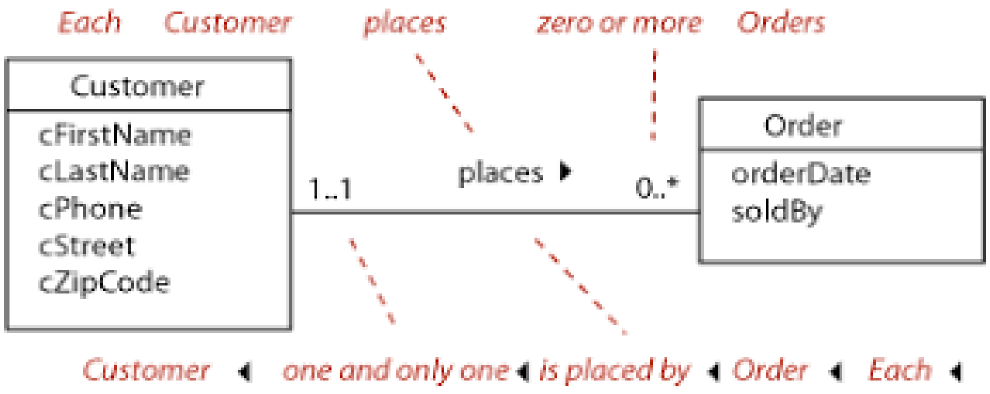
Most classes throughout a hierarchy lead to instances; any class that has instances is called a **concrete class**. Some classes do not produce instances because they are used merely as templates for other, more-specific classes (especially classes located high up in a hierarchy). The classes are referred to as **abstract classes**. Person is an example of an abstract class. Instead of creating objects from Person, we create instances representing the more-specific classes doctor and Patient, both types of Person

**Association**

This is a relationship or link between instances (or objects) of classes. An association is a naturally occurring relationship between specific things, such as an order is placed by a customer and an employee works in a department. **Is placed by** and **works in** are two associations that naturally occur between specific things.

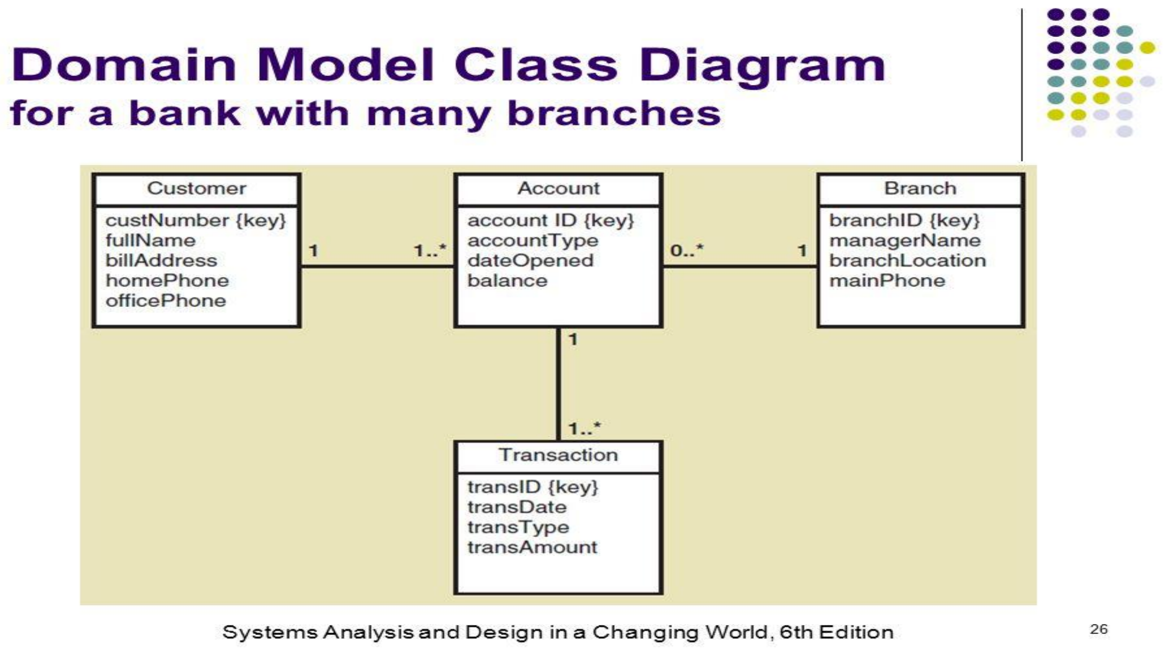
It is also important to understand the nature of each association in terms of the ***number of links for each thing and in which direction***. For example, a customer might place many different orders, but an order is placed by only one customer. In database management, the number of links that occur is referred to as the **cardinality** of the association. Cardinality can be one-to-one or one to-many. The term **multiplicity i**s used to refer to the number of links in UML and should be used when discussing UML models. I.e. **It is the number of objects that can be involved in that relationship.**

The multiplicity of customer is **1** and that of sales order is **0…*** this means that a customer object can relate to several sales order transactions or may not relate to any sales order, however an order must relate to one and only one customer



More on multiplicity:

| Adornment | Semantics |
|---|---|
| 0..1 | Zero or 1 |
| 1 | Exactly 1 |
| 0..* | Zero or more |
| * | Zero or more |
| 1..* | 1 or more |
| 1..6 | 1 to 6 |
| 1..3,7..10,15, 19..* | 1 to 3 *or* 7 to 10 *or* 15 exactly *or* 19 to many |


**Domain Model Class Diagram**
**for a bank with many branches**

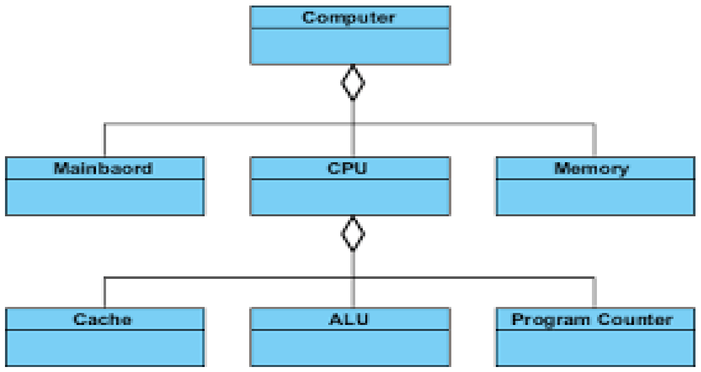
Systems Analysis and Design in a Changing World, 6th Edition 26

**Whole-Part Relationships** (**aggregation and composition)**

Whole-part relationships are used to show an association between one class and other classes that are parts of that class.
For example, learning about **a computer system** might involve recognizing that the computer is actually a collection of parts: processor, main memory, keyboard, disk storage, and monitor. A keyboard is not a special type of computer; it is part of a computer, but it is also something separate.

There are two types of whole-part relationships:
***Aggregation***: refers to a type of whole-part relationship between the aggregate (whole) and its components (parts), where the parts can exist

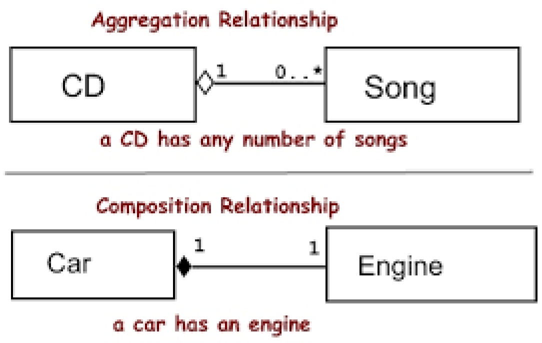
separately. Example aggregation in a computer system, in UML diamond symbol represents aggregation.



***Composition***: refers to whole-part relationships that are even stronger (Composition is a stronger variant of aggregation), where the parts, once associated, can no longer exist separately. The UML diamond symbol is filled in solid to represent composition.

For example, think of a house that is made up of bathrooms, bedrooms, stairwell, and so forth. The bathrooms or bedrooms never exist apart from the house; therefore, this whole-part relationship is a composition and uses solid diamond connectors.

Composition usually has a strong life cycle dependency between instances of the container class and instances of the contained class(es). If the container is destroyed, normally every instance that it contains is destroyed as well.



Others composition include; a department and an agency, a ward and hospital

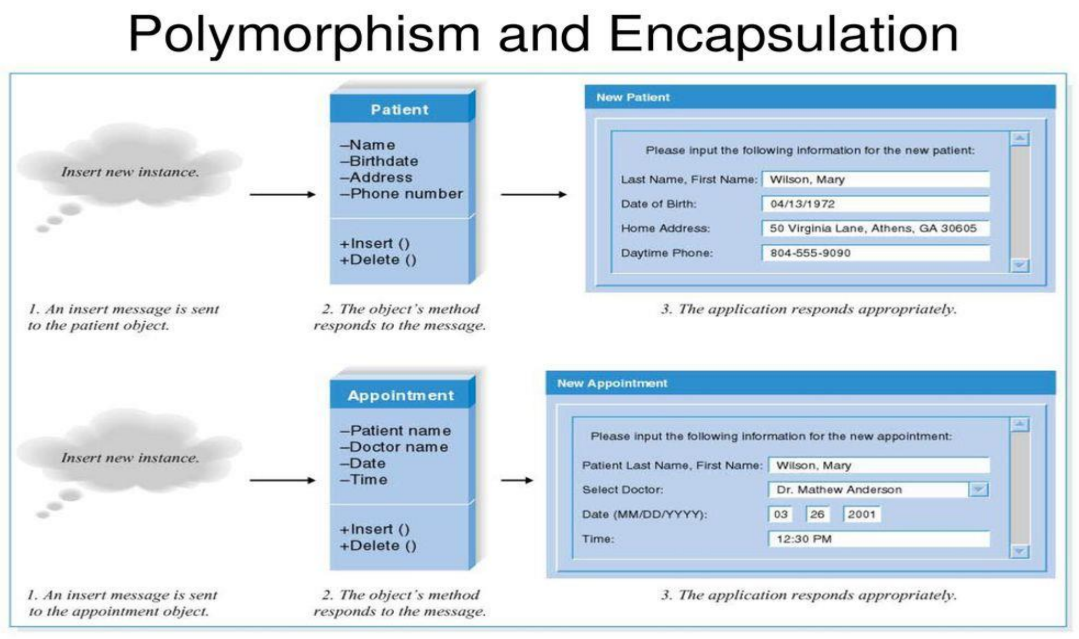
## Polymorphism

In the context of OOSAD Polymorphism means that the same message can be interpreted differently by different classes of objects. For example, **inserting a patient** means something different than **inserting an appointment**. As such, different pieces of information need to be collected and stored.

Luckily, we do not have to be concerned with how something is done when using objects. We can simply send a message to an object, and that object will be responsible for interpreting the message appropriately.

Polymorphism is made possible through **dynamic binding**. Dynamic, or late, binding is a technique that delays identifying the type of object until run-time. As such, the specific method that is actually called is not chosen by the object-oriented system until the system is running. This is in contrast to **static binding**. In a statically bound system, the type of object would be determined at compile time. Therefore, the developer would have to choose which method should be called, instead of allowing the system to do it.

***This ability facilitates easier development and reduces software design complexity. In OOSAD polymorphism has another significant use – it provides ability to hide different implementations behind an interface for component based development***


Polymorphism and Encapsulation
Insert new instance.
Patient
–Name
–Birthdate
–Address
–Phone number
+Insert ()
+Delete ()
1. An insert message is sent to the patient object.
2. The object's method responds to the message.
3. The application responds appropriately.
Insert new instance.
Appointment
–Patient name
–Doctor name
–Date
–Time
+Insert ()
+Delete ()
1. An insert message is sent to the appointment object.
2. The object's method responds to the message.
3. The application responds appropriately.

New Patient

Please input the following information for the new patient:

| | |
|---|---|
| Last Name, First Name: | Wilson, Mary |
| Date of Birth: | 04/13/1972 |
| Home Address: | 50 Virginia Lane, Athens, GA 30605 |
| Daytime Phone: | 804-555-9090 |

New Appointment

Please input the following information for the new appointment:

| | |
|---|---|
| Patient Last Name, First Name: | Wilson, Mary |
| Select Doctor: | Dr. Mathew Anderson |
| Date (MM/DD/YYYY): | 03 26 2001 |
| Time: | 12:30 PM |

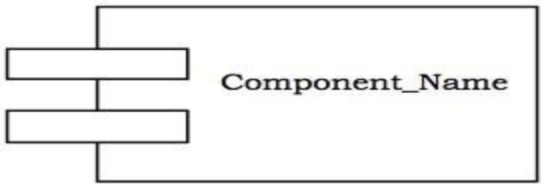
## Component

A component is a physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces. It represents the physical packaging of elements like classes.

A software component provides a set of interfaces.

An e-commerce application can have the following components: shopping cart, credit card, user account manager components

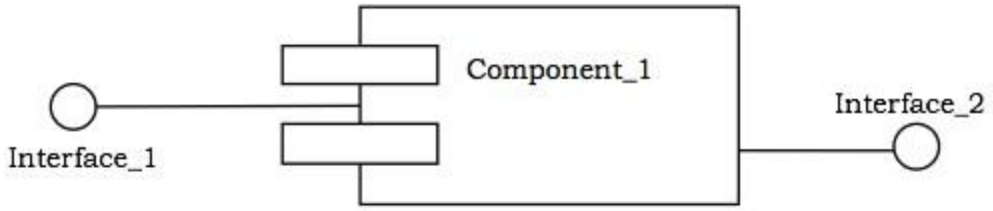
**Notation** – In UML diagrams, a component is represented by a rectangle with tabs as shown in the figure below.



## Interface

Interface is a collection of methods of a class or component. It specifies the set of services that may be provided by the class or component. The ***signatures*** of this operations/methods forms the interface of the class.

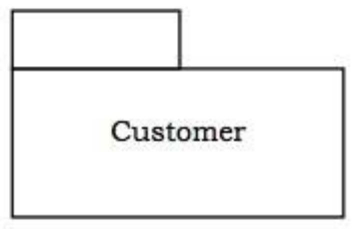
**Notation** – Generally, an interface is drawn as a circle together with its name. An interface is almost always attached to the class or component that realizes it. The following figure gives the notation of an interface.



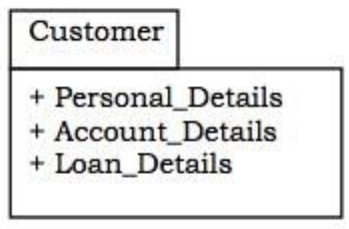
## Package

A package is simply an organizing unit. A package is an organized group of related elements. A package may contain structural things like classes, components, and other packages in it. A package is ***different*** from a component which has clearly defined function. A package clusters related components and classes in a named mechanism

**Notation** – Graphically, a package is represented by a tabbed folder. A package is generally drawn with only its name. However, it may have additional details about the contents of the package. See the following figures.



(a)



(b)

## <u>SYSTEMS MODELING</u>

### What is modeling?

A model is an abstraction of system for the purpose of understanding it before building it. Because, real systems that we want to study are generally very complex. In order to understand the real system, we have to simplify the system. So a model is an **abstraction** that hides the non-essential characteristics of a system and highlights those characteristics, which are pertinent to understand it.

Efraim Turban describes a model as a simplified **representation of reality.**

A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus modeling enables us to cope with the complexity of a system.
Most modeling techniques used for analysis and design involve **graphic languages**. These graphic languages are made up of sets of symbols. As you know one small line is worth thousand words. So, the symbols are used according to certain rules of methodology for communicating the complex relationships of information more clearly than descriptive text.

The information system needs to be modeled from various aspect /perspective such as structural and dynamic. UML has emerged as a popular language for developing systems based on OOSAD

Modeling is used frequently, during many of the phases of the software life cycle such as analysis, design and implementation. Modeling like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added to

### Why do we model?

Before constructing anything, a designer first builds a model. The main reasons for constructing models include:

-To test a physical entity before actually building it.

-To set the stage for communication between customers and developers

- *to demonstrate, or clarify, understanding of the existing system and/or to obtain feedback from users/clients;*
- *to describe unambiguously the proposed computer system to users/clients and to the programming team.*

-For visualization i.e. for finding alternative representations.

-For reduction of complexity in order to understand it.

-serves as aids to documentation

-helps to detail and construct the system