OO Analysis: Structural View
Identification of Classes and Relationships

So far, we captured User's View by having Use Case Model (Use-Case Diagrams, and Use Case Descriptions) Use Case Model specifies, from the user's point of view, what the system must do. Moving from problem domain to solution domain, software to be build, to properly analyze we divide the system into separate parts. However, the software structure of an object-oriented system is based on objects. We must define objects that have the capability of delivering that functionality. We find the "real-world" objects involved in the use cases and creating groups of objects (class). Here we don't split it up according to use cases but by objects and groups of objects (classes).

To have a *Structural View*, we model the static structure of a system. Static structure depicts elements (objects/Classes) of the system and the relationships between them. Elements and the relationships between them do not change over time. For example, in student's office example in use case modeling, students have a name and a registration number and attend various courses. This sentence covers a small part of the university structure and does not lose any validity even over years. It is only the specific students and courses that change. To model structure of a system *class diagram* is the most widely used UML diagram. Class diagram is applied in various phases of the software development process. The level of detail or abstraction of the class diagram is different in each phase. In the early phases, a class diagram allows you to create a conceptual view of the system and to define the vocabulary to be used.

In object-oriented approach we identify objects which represent things in the problem domain and about which we need to store information. In the development process we add objects to provide the software structure we want to implement and objects that are to do with how the software will work. The original objects, although they may gather some extra features in the development process, will still be identifiable in the code.

Structure Analysis Process

- Examining the problem domain/statement to identify objects.
- Identify entities, relationships, and operations of the problem domain.
- Understanding the role of the objects by examining the relevant data and processes

To create *Basic Class Diagram*, also referred as *Structural Model* we perform following steps*:*

1. Identify Objects and Classes.
2. Identify Object attributes.
3. Identify operations on Objects and Classes.
4. Define Object and Class relationships.
5. Illustrate Objects/Class relationships.

A benefit of the OO approach is that the same concepts appear in all stages of development. OO techniques view software systems as systems of communicating objects, where each object is an instance of a class. All objects of a class share similar features(Attributes and Operations). Classes can be specialized by subclasses. Objects communicate by sending messages.

An ***object*** is a representation of something in the application area about which we need to store data to enable the system to do what the users want it to. In a library system: we will undoubtedly want to store data about books. Books, therefore, would be objects in our model of the library system and eventually in the code. In "Hire-n-Ride-Bike" System data to be stored about bikes, Bikes, would be objects in our model of the system. The data to store about bikes (type, daily hire rate, deposit) and is referred as *attributes*.

A ***class*** is the construction plan for a set of similar objects that appear in the system to be specified. Classes can characterize, persons (e.g., students), things (e.g., buildings), events (e.g., courses or exams), or even abstract concepts such as groups. Objects represent the concrete forms of classes and are referred to as Characteristics of their instances. The relevant characteristics of the instances of a class are described through the definition of structural characteristics (attributes) and behavior (operations). Operations enable objects to communicate with one another and to act and react. An attribute allows to store information that is known for all instances but that generally has different specific values for each instance. Operations specify how specific behavior can be triggered on individual objects.

Classes provide schemas for characterizing objects and objects are instances of classes. An object has a unique identity and several characteristics that describe it in more detail, Structural characteristics (attributes) and behavior (in form of operations). An object rarely appears in isolation in a system; instead, it usually interacts and communicates with other objects. Operations are identical for all objects of a class and are therefore usually described exclusively for the class.

Class Notation:
A class is represented by a rectangle that can be subdivided into multiple compartments. First compartment must contain the name of the class as shown below:

Name
attributes
operations

2nd compartment contains the attributes of the class, and 3rd compartment the operations of the class. class names are singular nouns or nouns phrases. The class name should describe the class using vocabulary typical for the application domain. Class name is positioned centered, whereas contents of other two compartments are positioned left-justified and are optional. In general, the level of detail in these compartments reflects the respective phase of the software development process in which the class is being examined.

An ***Object*** is an abstraction of a person, place, thing, or concept within the problem domain. We use objects both to model the real-world characteristics of the application area and to provide us with a basis for the computer implementation. Objects are "instantiated" means created, and the term "instance" is interchangeable with "object".

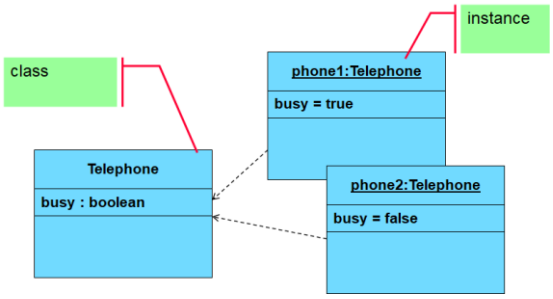
Objects are further defined as someone having following characteristics:

- What they know about themselves (Attributes).
- What they do (Operations).
- What they know about other objects (Relationships).

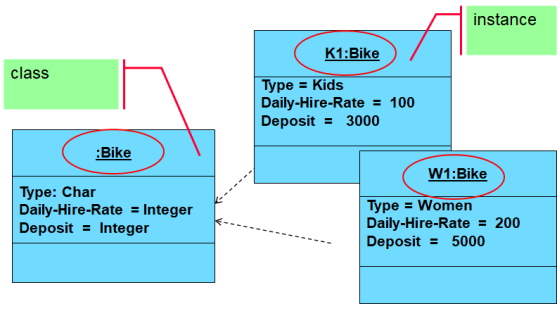
Identifying Objects and Classes

Class: A sets of objects with equivalent roles in the system. Every object belongs to a class. Objects may be tangible (book, copy, course), or roles (library member, student) or events (arrival, leaving, request) or interaction (meeting, intersection).

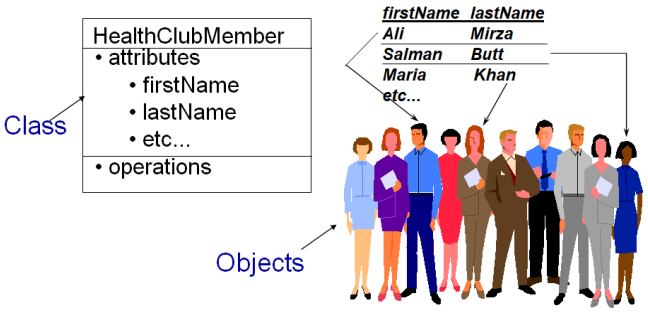
A Class is a template (specification, blueprint) for a collection of objects that share a common set of attributes and operations. A class of objects is a group of objects with the same set of attributes, the same relationships and the same behaviour. Although we start by identifying objects, it is the class that determines the structure and behaviour of its objects. Producing objects is a class's main role in life; a class is an object factory, it can produce hundreds of objects, all with exactly the same structure and behaviour. When we define a class we define the structure and the public interface for all objects of that class. Objects are instances of a Class. We Describe/Specify one or more distinct objects with a common form (structure and behavior) as shown below:



Class and Objects from "Hire-n-Ride-Bike" System of case study are depicted below:

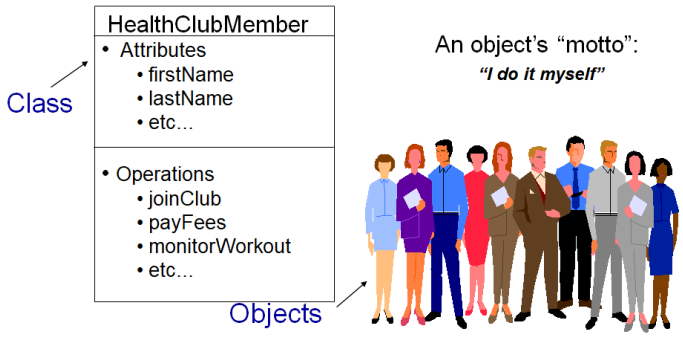


Next we define attributes of objects. An ***Attribute*** (properties, characteristics) is data that further describes an object instance. In the diagram shown below, we depict attributes and objects of a class "HealthClubMember"



Next, we identify operations of a class. Operations are used to interact with the objects and classes. Each class has a protocol that captures the messages or operations that could be

handled by the class or its instances (objects). Users or "clients" of classes and objects deal primarily with the external operations of the class. An ***Operation*** (methods, services, behavior) is a procedure that an object can perform. In the diagram shown below, we depict attributes and operations of a class "HealthClubMember"



OO Analysis (Structural)

To ensure that a model remains clear and understandable, we generally do not model all of the details of the content. Include the information that is relevant for the moment. This means that we abstract from reality to make the model less complex and to avoid an unnecessary flood of information.

Classes and Objects as the two building blocks of object-oriented systems. Structure of a software system is defined by the way in which these building blocks relate with one another. Behaviour of the system is defined by the manner in which the objects interact with one another. In order to construct a software system, we need mechanisms that create connections between these building blocks.

Relationships between Classes

To have a structural view, we must identify relationships between different parts of the system, and in OO context it means we identify relationships between classes. A relationship is what a class or an object knows about another class or object. Simplest and most general kind of relationships is ***association***, which simply indicates that the objects of the two classes are related in some non-hierarchical way. There are almost no other restrictions on how an association can be formed.

When two or more classes have a hierarchical relationship based on generalization, it is referred to as ***inheritance***. Classes connected by inheritance share some commonalities and is referred as Superclass-Subclass relationship as shown in the following:

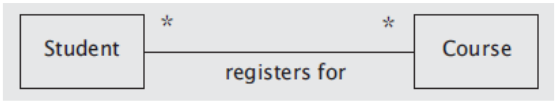
- Person - FacultyPerson, StudentPerson, Staff (Person a superclass, and FacultyPerson, StudentPerson, and Staff as subclasses)
- ModesOfTravel - Airplane, Train, Auto, Cycle, Boat (ModesOfTravel a superclass, and Airplane, Train, Auto, Cycle, and Boat as subclasses)

An ***Association*** is formally defined as a relation among two or more classes describing a group of links with common structure and semantics. An association implies that an object of one class is making use of an object of another class and is indicated simply by a solid line connecting the two class icons.

Associations represent conceptual relationships between classes as well as physical relationships between the objects of these classes as shown below.

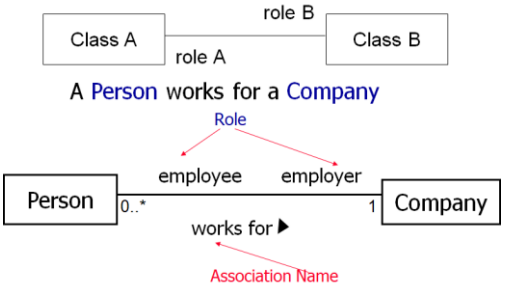
- FacultyInformation – CourseInformation
- StudentInformation - CourseInformation

The association usually has a descriptive name. The name often implies a direction, but in most cases, this can be inverted. Associations correspond to verbs in the problem statement as shown below:



The following diagram depicts that Student objects may make use of Course objects when transcripts are generated, when tuition is computed or when a course is dropped:

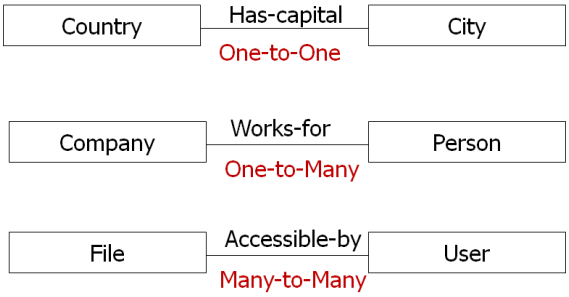
The mechanism to make this connection may simply be that the Student object stores a reference to the Section object. Each section is associated with a unique course. When a student registers for a course, he/she actually enrolls in a specific section of the course. Our diagram shown above says student enrolls in a section, which belongs to a course, but this could be stated as a course has sections that enroll students. The diagram is usually drawn to read the link or association from left to right or top to bottom. Association basically depicts a role being played by a class as shown below:



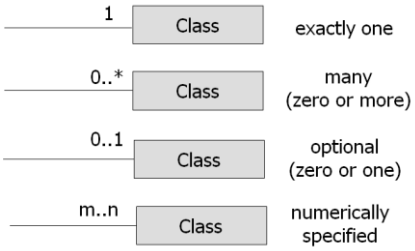
Characteristics of Associations

Since associations represent very general relationships, they allow for variation in the nature of the connection. The common variation is the arity of the connection, i.e., how many objects of class A can be linked to one object of class B and vice-versa. Another variation involves whether there is some form of containment involved in the relationship. In other cases there is some specific kind of information that is added to the system whenever a link is made between objects. These characteristics are usually represented in UML by annotating the connection between classes. Some of these are discussed below.

Arity of Relationships could be one–one, one–many, or many–many as shown below:



An association can also be represented as an attribute with the appropriate multiplicity, whereby the type of the attribute is the class of the corresponding partner objects. Association multiplicities are summarized as shown below:

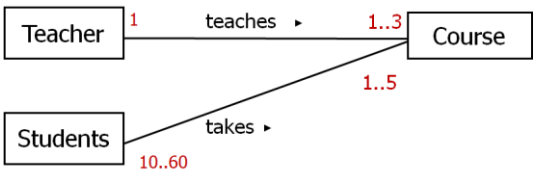


Association labels and multiplicities will be clear by working through following simple examples:

In university system we have defined classes (Blue color) from the following working description, which we use to draw associations between classes:

A Teacher teaches 1 to 3 Courses.
A Student can take up to five Courses.
Student has to be enrolled in at least one course
Up to 60 students can enroll in a course.
A class should have at least 10 students.

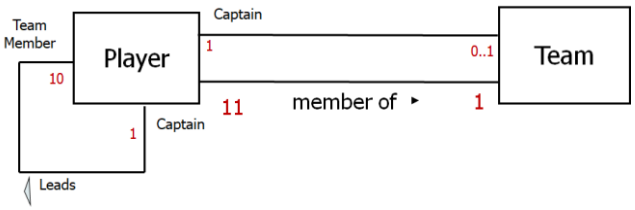
Associations between classes is depicted as shown below:



A members-team-example:

A cricket Team has 11 Players.
One of them is the captain.
A Player can play only for one Team
The captain leads the team members

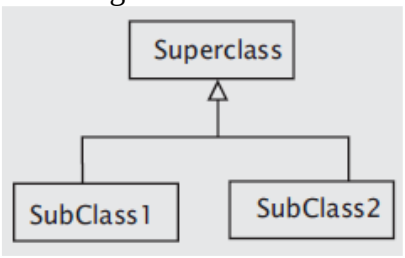
Associations between classes is depicted as shown below:



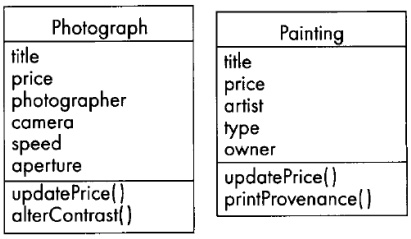
Inheritance

There are situations when two classes have not only a great deal of similarity, but also significant differences. The classes may be similar enough that association does not capture the similarity, and differ too much so that the idea of genericity cannot be profitably employed. Suppose that C1 and C2 are two such classes. We then extract the common aspects of C1 and C2 and create a class, say, B, to implement that functionality. The classes C1 and C2 could then be smaller, containing only properties and methods that are unique to them. This idea is called inheritance - C1 and C2 are said to inherit from B. B is said to be the superclass, and C1 and C2 are termed subclasses. The superclasses are generalisations

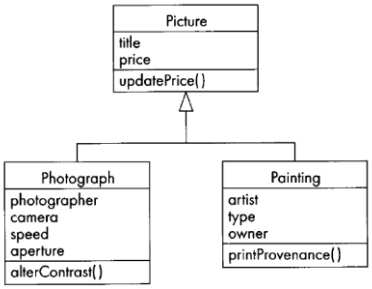
or abstractions: we move toward a more general type, an 'upward' movement, and subclasses denote specialization toward a more specific class – a 'downward' movement. The class structure then provides a hierarchy. Formally, an inheritance is a relationship characterized by an ancestor and a descendant represented using UML notation as below:



To further clarify Inheritance (Generalization) association with the help of following example.
In a system for an art gallery, we have two classes namely Photograph and Painting and are shown below along with attributes and operations:



Photograph and Painting have 2 attributes (title and price) and one operation updatePrice() in common. We can introduce a new general class, Picture, in which we can place these common features i.e. Attribute (title and price) and Operation (updatePrice()). Inheritance association between Picture (superclass) with Photograph and Painting as subclasses is shown below:



Containment Relationships

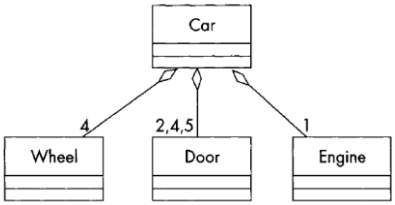
Aggregation: a specialized form of Association in which a whole is related to its part(s). Aggregation is a kind of association where the object of class A is 'made up of objects of class B. This suggests some kind of a whole–part relationship between A and B. UML notation, aggregation is shown as a line joining the two classes with a diamond next to the whole class. Aggregation is known as a "part of" or *containment relationship* and follows "has a" heuristic. For example, in the following "has a" is implicit in class association:

Assembly – Parts
Group – Member
Container – Contents

For example, a car and its parts (wheel, door and engine). If we have to depict a car and its parts with the attributes:

Car has 4 wheels, Car has 2, or 4, or 5 doors, and Car has an engine

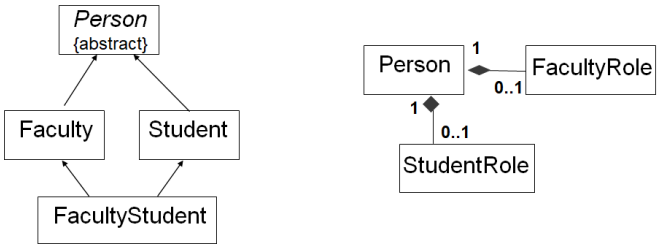
The aggregation association of car example is shown below:



Composite aggregation, also known as composition, has been recognised as being significant. Composition implies that each instance of the part belongs to only one instance of the whole, and that the part cannot exist except as part of the whole. Composition is indicated with a filled-in diamond and is usually not named since some form of whole–part relationship is assumed. For example, a vertex cannot exist unless it is a part of a triangle as shown below:

If the triangle object is destroyed, so are the individual vertices.

Composition is often used in place of Generalization (inheritance) to avoid “transmuting” (adding, copying, and deleting of objects) as shown in the following diagram.



Instead of Generalization (inheritance) association on the left side, composition on the right is more representative of the situation.