Introduction to Object Oriented Analysis and Design

Basic Concepts of Object Oriented Programming
It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:
• Objects
• Classes
• Data abstraction and encapsulation
• Inheritance
• Polymorphism
• Dynamic binding
• Message passing

Objects
Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analysed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

Classes
We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. **A class is thus a collection of objects similar types.** For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language.

Data Abstraction and Encapsulation
The wrapping up of data and function into a single unit (called class) is known as *encapsulation.* Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function*.

Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real- world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent.

An Introduction to the UML

The *Unified Modelling Language*, or the *UML*, is a graphical modelling language that provides us with a syntax for describing the major elements (called *artifacts* in the UML) of software systems.

Other industries have languages and notations, which are understood by every member of that particular field.

The UML is a graphical language for capturing the artifacts of software developments.

- The language provides us with the notations to produce models.
- The UML is gaining adoption as a single, industry wide language.
- The UML was originally designed by the Three Amigos at Rational Corp.
- The language is very rich, and carries with it many aspects of Software Engineering best practice.

A software development will have many stakeholders playing a part. for example:

- Analysts
- Designers
- Coders
- Testers
- QA
- The Customer
- Technical Authors

All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behaviour of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram.

COMMON DIAGRAMS USED IN UML

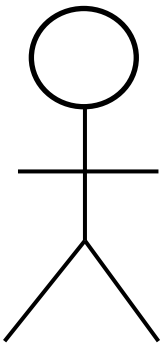
1. **Use Cases** - .How will our system interact with the outside world?
2. **Class Diagram** - .What objects do we need? How will they be related?
3. **Collaboration Diagram** - .How will the objects interact?
4. **Sequence Diagram** - .How will the objects interact?
5. **State Diagram** - .What states should our objects be in?
6. **Package Diagram** - .How are we going to modularise our development?
7. **Component Diagram** - .How will our software components be related?
8. **Deployment Diagram** - .How will the software be deployed?

Use Case diagram

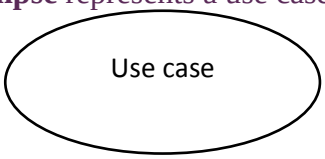
A use case is a description of a system's behavior from a user's standpoint. For system developers, this is a valuable tool: it's a tried-and-true technique for gathering system requirements from a user's point of view. That's important if the goal is to build a system that real people can use

An actor initiates a use case, and an actor (possibly the initiator, but not necessarily) receives something of value from the use case.

The actor is represented by the symbol below. The actor's name appears just below the actor.



An **ellipse** represents a use case

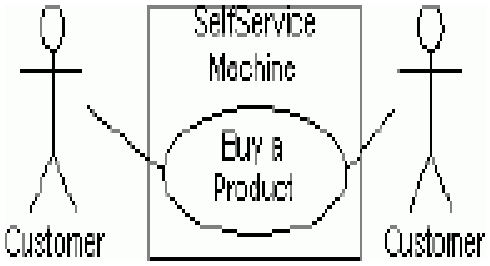


The name of the use case appears inside. An association line connects an actor to the use case, and represents communication between the actor and the use case. The association line is solid, like the line that connects associated classes.

Use case diagrams add more power to the requirements gathering. They visualize use cases. They also facilitate communication between analysts and users and between analysts and clients.

Use case diagram example1

The actor in this use case is customer. This customer wants to buy some of the products offered by the self-service machine. First of all he/she inserts money into the machine, selects one or more products, and machine presents a selected product(s) to the customer. Use case diagram for this scenario can be represented as:



The Buy a Product use case diagram

Use cases use active verbs. Eg. Buy product, register for a course, display courses etc. The rectangle is showing the system boundary. The actors are normally outside the system boundary.

UML Use Case Include

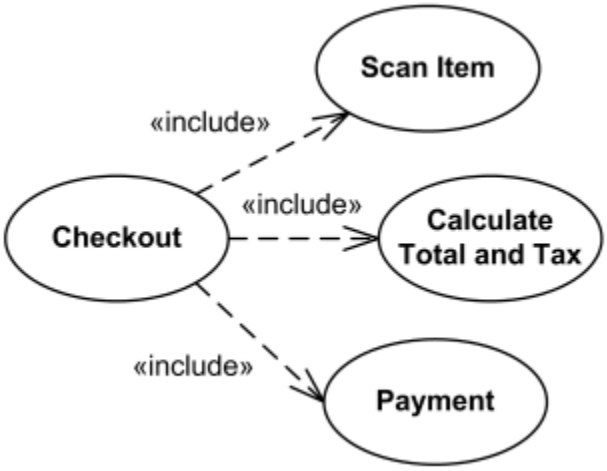
Use case include is a **directed relationship** between two **use cases** which is used to show that behavior of the **included** use case (the addition) is inserted into the **behavior** of the **including** (the base) use case.

The **include** relationship could be used:

- to simplify large use case by splitting it into several use cases,
- to extract **common parts** of the behaviors of two or more use cases.

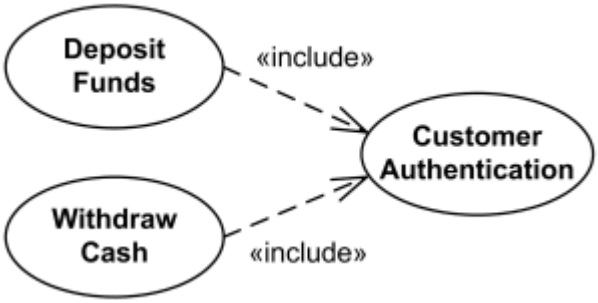
A large use case could have some behaviors which might be detached into distinct smaller use cases to be included back into the base use case using the UML**include** relationship. The purpose of this action is modularization of behaviors, making them more manageable.

Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword «include».



Checkout use case includes several use cases - Scan Item, Calculate Total and Tax, and Payment

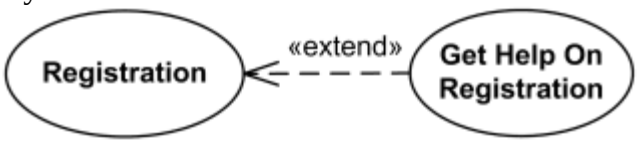
Large and complex Checkout use case has several use cases extracted, each smaller use case describing some logical unit of behavior. Note, that including Checkout use case becomes incomplete by itself and requires included use cases to be complete.



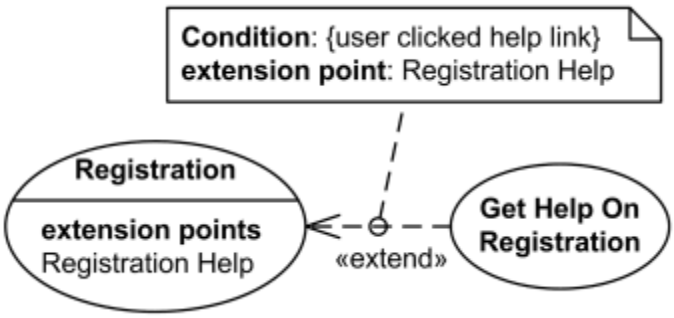
***Deposit Funds** and **Withdraw Cash** use cases include **Customer Authentication** use case.*

UML Use Case Extend

Extend is a directed relationship that specifies how and when the behavior defined in usually supplementary (optional) **extending use case** can be inserted into the behavior defined in the **extended use case**.
Extended use case is meaningful on its own, it is **independent** of the extending use case. **Extending** use case typically defines **optional** behavior that is not necessarily meaningful by itself. The extend relationship is **owned** by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.
The extension takes place at one or more extension points defined in the **extended use case**.
Extend relationship is shown as a dashed line with an open arrowhead directed from the **extending use case** to the **extended (base) use case**. The arrow is labeled with the keyword **«extend»**.



Registration *use case is complete and meaningful on its own. It could be extended with optional* ***Get Help On Registration*** *use case.*
The **condition** of the extend relationship as well as the references to the extension points are optionally shown in a comment note attached to the corresponding extend relationship.



Class Diagrams

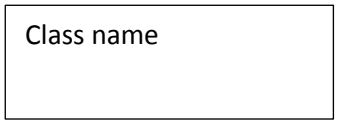
Classes

An object is any person, place, thing, concept, event, screen, or report applicable to your system. Objects both know things (they have attributes) and they do things (they have methods). A class is a representation of an object and, in many ways, it is simply a template from which objects are created. Classes form the main building blocks of an object-oriented application.

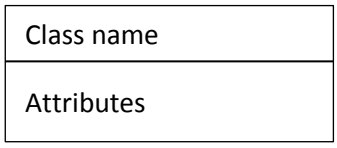
In software engineering, a **class diagram** in the **Unified Modeling Language** (**UML**) is a type of static structure **diagram** that describes the structure of a system by showing the system's **classes**, their attributes, operations (or methods), and the relationships among objects.

Drawing a class diagram

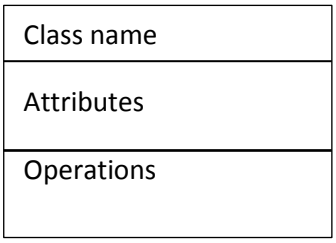
Class name in top of box – write <> on top of interfaces' names – use italics for an abstract class name.



Attributes (optional) – should include all fields of the object



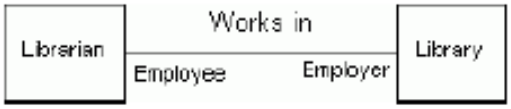
Operations / methods (optional) – may omit trivial (get/set) methods • but don't omit any methods from an interface! – should not include inherited methods



Although thousands of students attend the university, you would only model one class, called *Student*, which would represent the entire collection of students.

Associations

When classes are connected together conceptually, the connection is called an association.
You visualize the association as a line connecting the two classes, with **the name of the association just above the line.**



When one class associates with another, each one usually plays a role within that association. You can show those roles on the diagram by writing them near the line next to the class that plays the role.

Association may be more complex than just one class connected to another. Several classes can connect to one class.
Sometimes an association between two classes has to follow a rule. You indicate that rule by putting a constraint near the association line.

Notation for associations.

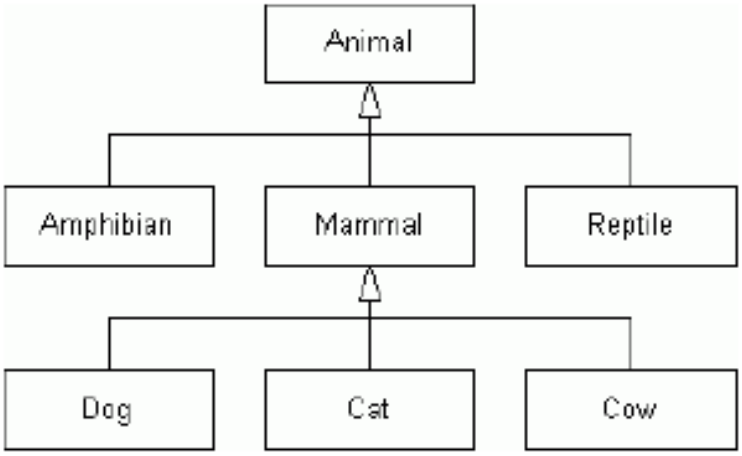
Figure 6. Notation for associations.

It is not enough simply to know professors instruct seminars. How many seminars do professors instruct? None, one, or several? Furthermore, associations are often two-way streets: not only do professors instruct seminars, but also seminars are instructed by professors. This leads to questions like: how many professors can instruct any given seminar and is it possible to have a seminar with no one instructing it? The implication is you also need to identify the multiplicity of an association. The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction (Table 1 summarizes the potential multiplicity indicators you can use).

Table 1. Multiplicity Indicators.

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

Inheritance & Generalization
If you know something about a category of things, you automatically know some things you can transfer to other categories.
If you know something is an animal, you take for granted that it eats, sleeps, has a way of being born, and has a way of getting from one place to another... But imagine that mammals, amphibians and reptiles are all animals. Also cows, dogs, cats... are grouped in category mammals. Object-orientation refers to this as **inheritance**.



An inheritance hierarchy in the animal kingdom

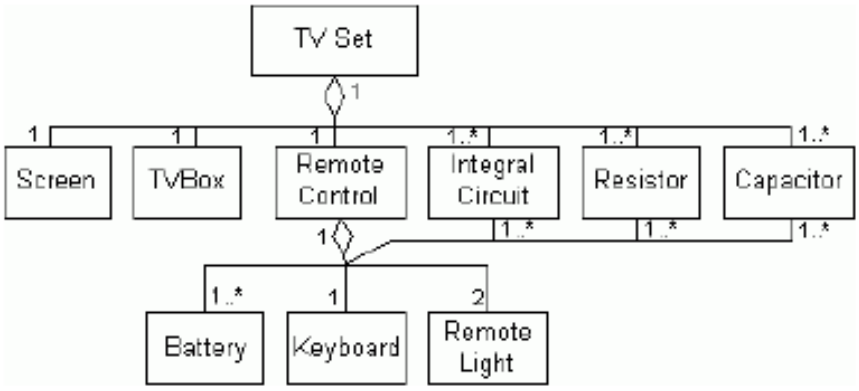
One class (the child class or subclass) can inherit attributes and operations from another (the parent class or superclass). The parent class is more general than the child class. In generalization, a child is substitutable for a parent. That is, anywhere the parent appears, the child may appear. The reverse isn't true, however.

Aggregations

Sometimes a class consists of a number of component classes. This is a special type of relationship called aggregation. **The components and the class they constitute are in a part-whole association.**

Aggregation is represented as a hierarchy with the "whole" class at the top, and the component below. A line joins a whole to a component with an open diamond on the line near the whole.

Let's take a look at the parts of a TV set. Every TV has a TV box, screen, speaker(s), resistors, capacitors, transistors, ICs... and possibly a remote control. Remote control can have these parts: resistors, capacitors, transistors, ICs, battery, keyboard and remote lights.



An aggregation association in the TV Set system