Introduction to UML 2.0

Mrityunjoy Saha
27th August, 2007

Table of Contents:

1 Introduction

The Unified Modelling Language or UML is a graphical modelling language that is used to express software designs. Components of a software system can be specified using UML. UML describes a notation and not a process. But a process can be designed using this language.

2 Diagrams

UML 2 defines thirteen basic diagram types, divided into two general sets:

- Structural Modelling Diagrams
- Behavioral Modelling Diagrams

In this document we will focus on only most commonly used diagrams in each of the above two categories. These are:

Structural Modelling Diagrams

- Package diagram
- Class diagram

Behavioral Modelling Diagrams

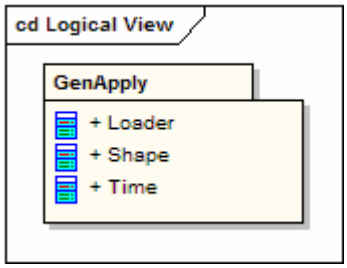
- Use Case diagram
- Sequence diagram
- Activity and State diagram

2.1 Structural Modelling Diagrams

Structure diagrams define the static architecture of a model. We can use structure diagrams to model parts such as – classes, objects, interfaces and physical components. Also structure diagrams are used to model the relationship and dependencies between elements.

2.1.1 Package diagram

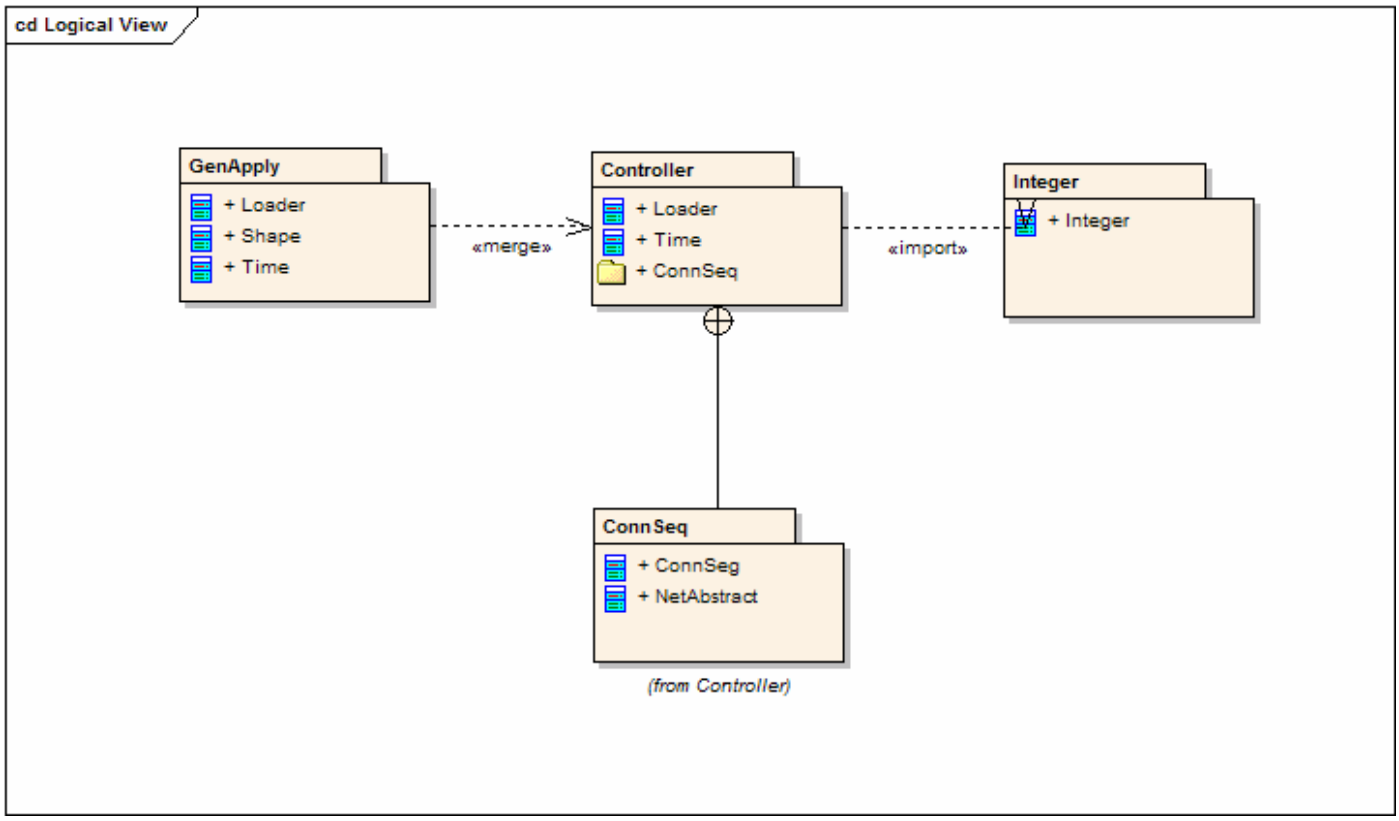
Package diagrams represent the structure of packages and their elements. Package diagrams are most commonly used to organize use case diagrams and class diagrams. Let's take a look at following basic and simplest package diagram:



The above package diagram shows a package named 'GenApply' and its elements – Loader, Shape and Time. Packages are represented as folders and contain the elements that share a namespace; all elements

within a package must be identifiable, and so has a unique name or type. The package must show the package name and can optionally show the elements within the package.

Now we would learn merge connector, import connector and nested connector with reference to following package diagram:



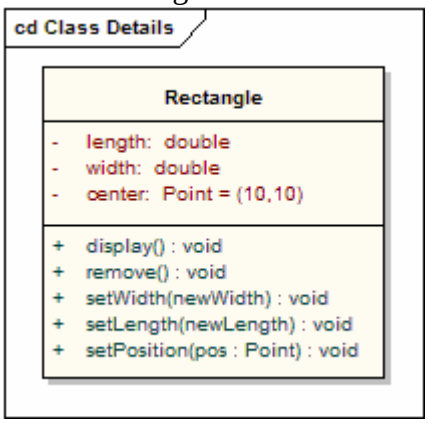
Package Merge:
A merge connector between two packages indicates that element definitions in the source package will be expanded when there is a corresponding matching element found (by name) in target package. Target package element definitions remain unaffected. Also unmatched elements in the source package will carry their original definition. In the above diagram a merge connector exists between package GenApply and Controller. Definition of Loader, Shape in the source package will be expanded since there is a corresponding element match found in target package.

Package Import:
The import connector denotes that target package elements can be referred by name (fully qualified name not required) when used in source package. In the above diagram target package Integer's element Integer can be referred by name from within source package Controller's elements.

Nested Connector:
Nested connector indicates that target package contains the entire source package. In the above diagram target package Controller contains the source package ConnSeq.

2.1.2 Class diagram

The class diagram is core to object-oriented design. It describes the types of objects in the system and the static relationships between them. The core element of the class diagram is the class. The below diagram shows a class named Rectangle:



The top part contains the name of the class, centre section contains attributes and bottom section contains operations that can be performed on the class.

Since class diagram is one of the most important parts we will explore this in detail.

Attributes and Operations:

An attribute is a property of the class and an operation is a task that can be performed on data in the class. The format of attribute is:

visibility name: type = default value

And the format of operation is:

visibility name (parameters): type

The format of a parameter is:

direction name: type = default value

Direction can be one of *in*, *out*, *inout* or it can be unspecified.

Visibilities can be one of these:

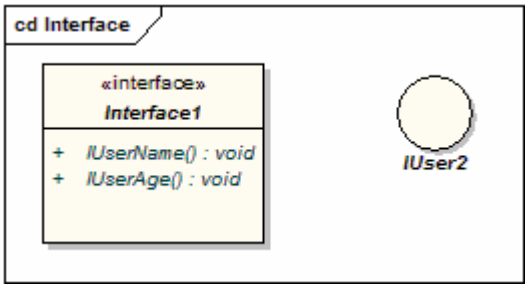
- private
+ Public
# Protected
~ Package

So in the above class diagram:

- length is a private attribute of type double.
- center is a private attribute (member) of type point and default value is (10, 10).
- remove () is a public operation (method) which returns of type void.

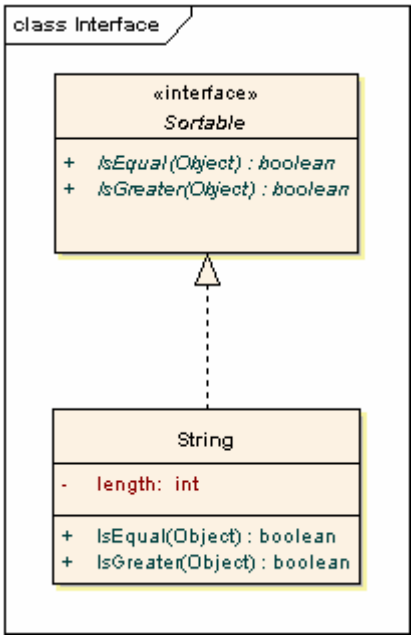
Interfaces:

Interfaces are very similar to abstract classes with the exception that they do not have any attributes. Interfaces may be drawn in a similar style to a class, with operations specified, as shown below:



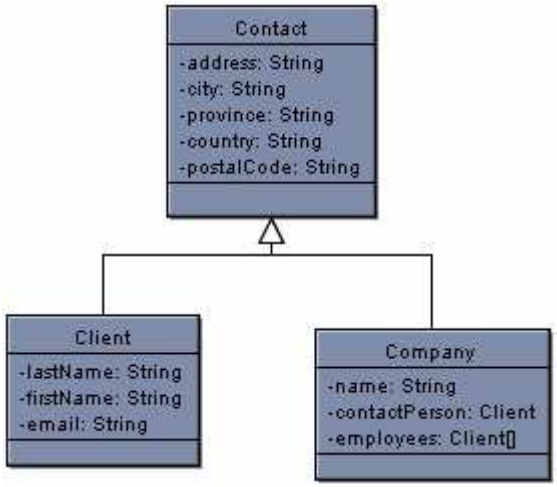
We all know that interface is a contract of behaviours which implementing classes must agree to meet. That is by realizing an interface classes are guaranteed to support specified behaviours. Are we talking too much OO concept? Let me state this in java: An implementation class of an interface must provide definition of all the methods declared in the interface.

Realization:
The source object realizes or implements the destination object. For example a business process is realized by use cases, which are in turn realized by classes etc. In general high level (abstract) components are realized by more specific components. Now if we want to show this in UML it will look like:



Sortable is an interface and String is a class which implements (realizes) the Sortable interface. The realization link (dashed line) shows the implementation. We can see, as agreed String class defines all the methods declared in Sortable interface.

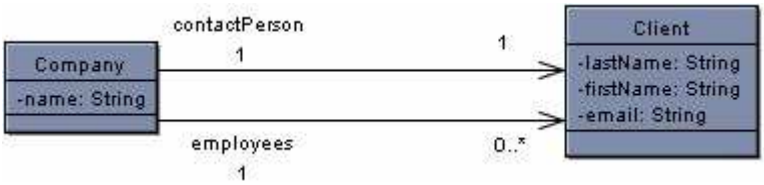
Generalization:
A generalization link is used to indicate inheritance.



From the above diagram we can say that Client and Company generalize or extend Contact (see the link). In Client and Company all the attributes and operations are available (depends on visibility) and also they can add more attributes and operations.

Association:
A class can contain reference to some other class. For example a Company (class) has a contact person (Client class) and a group of employees (multiple Clients). This can be shown by following class diagram:

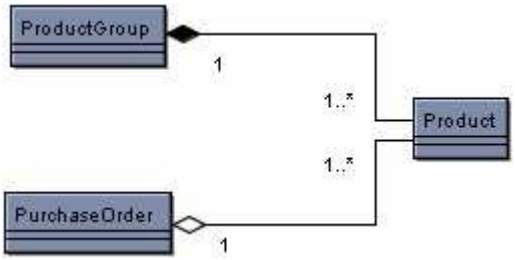
But the above becomes more expressive when attributes are shown as association:



The above diagram tells in a Company there is only one contact person. The multiplicity of association is one to one. Also it tells in the Company there are zero many employees. Multiplicities can be anything you specify:

- **0** zero
- **1** one
- **1..*** one or many

Aggregation and Composition:
Suppose we have many products in our company. Now with multiple products we can have a product group and also someone can place an order for multiple products. So both product group and purchase order are associated with product, but there is a difference. To understand that lets look at the following diagram:



Our general knowledge says:

- If a product group is destroyed then products within the group are also destroyed. The reason is Product Group is composed of Products. The composition association is represented by the solid diamond.
- If a purchase order is destroyed then products mentioned in the purchase order remain unaffected. The reason is Purchase Order is an aggregate of Products. The aggregation association is represented by the hollow diamond.

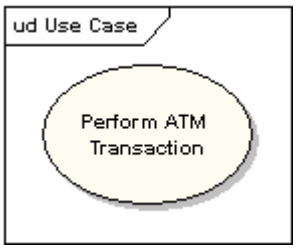
2.2 Behavioral Modelling Diagrams

Behavior diagrams show the interaction and states within a model as it executes over time.

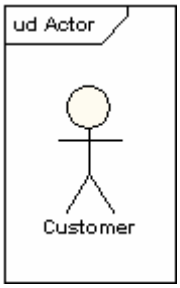
2.2.1 Use case diagram

In many design processes, the use case diagram is the first that designers will work with when starting a project. The use case model captures the requirements of a system. This diagram allows for the specification of high level user goals that the system must carry out.

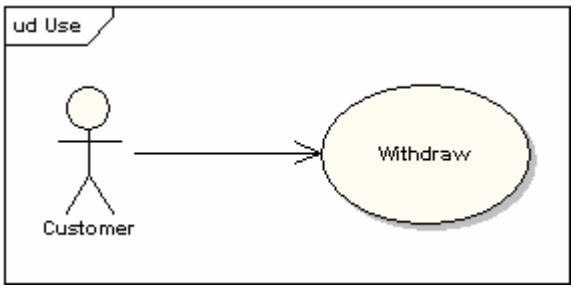
Use Case:
A use case is made up of a set of scenarios and each scenario is a sequence of steps. More specifically a use case is a single unit of meaningful work. The following diagram shows a use case (ATM transaction):



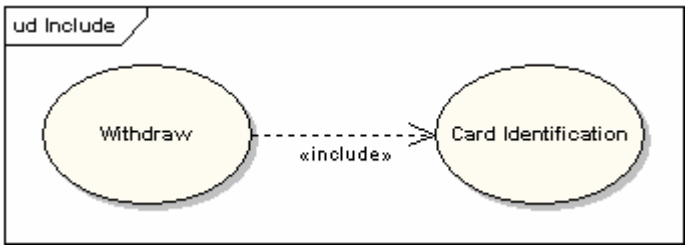
Actor:
The use case diagram allows us to graphically show use cases and the actors that use them. An actor is a role that a user plays in the system. An actor is usually drawn as a named stick figure:



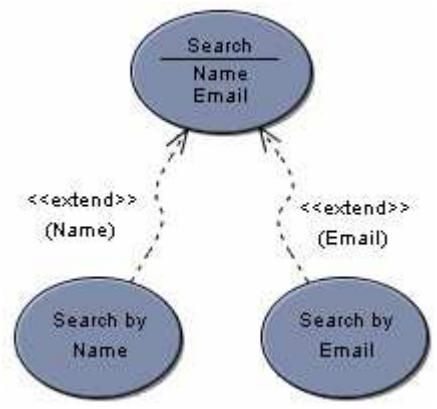
Let us say actor "Customer" uses the “Money Withdraw" use case. Now in UML it can be represented as:



Includes:
Use cases may contain the functionality of another use case as part of their normal processing. While withdrawing money at ATM using a card, card verification is a compulsory process. If money withdraw is a use case and card verification is a separate use case then we can say money withdraw use case includes card verification use case. In UML it can be shown as:



Use Case extending:
Suppose we have developed an employee search use case. Recently requirement team says now they need a new functionality which contains everything of employee search use case plus something additional. If we give a name to this new functionality as search by name then we can say search by name use case extends search use case. Note that functionalities can only be added at the extension point. With reference to this example let’s look at the following diagram:

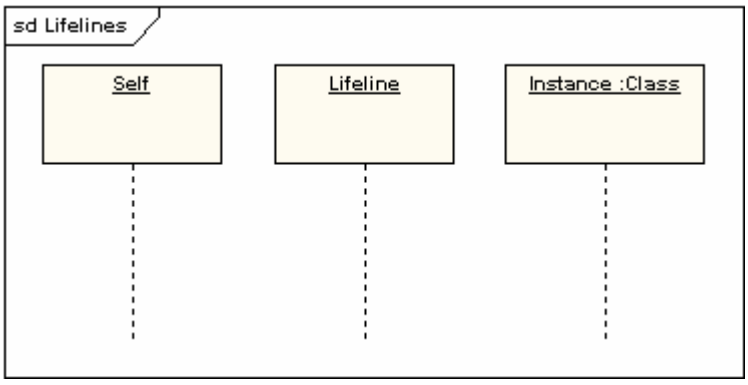


2.2.2 Sequence diagram

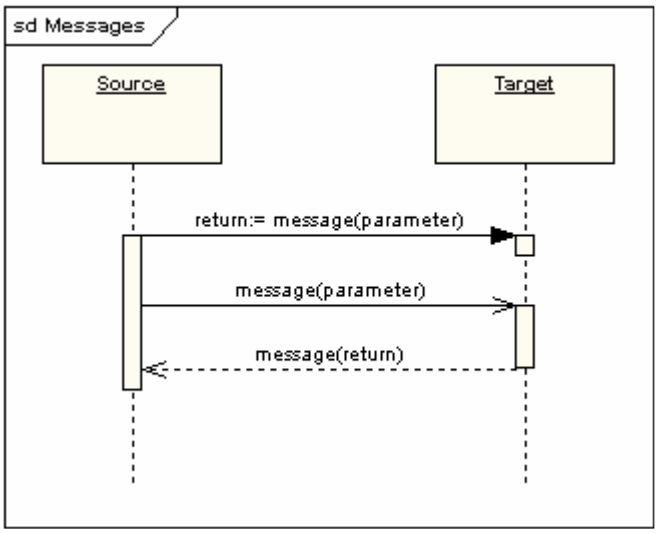
Sequence diagram represents the dynamic behaviour of a system. It shows interaction between objects via messages (operations, but we are familiar with term method) over time.

Lifelines:

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. An actor also can have its lifeline.

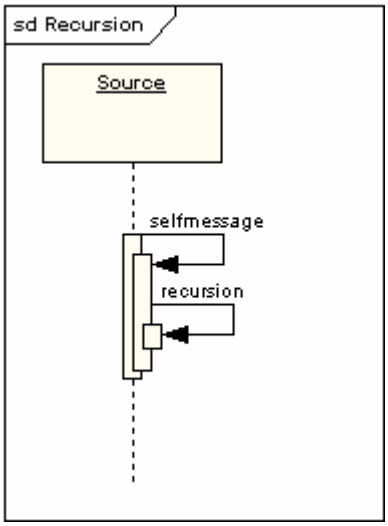


Messages:
Messages represent operation calls and displayed as arrows. Let us look at the following diagram:



The thin rectangle running down the lifeline (dotted line in each object) denotes the execution occurrence. The first message (method call) is a regular call. The second message is asynchronous (denoted by line arrowhead), and the third is the asynchronous return message (denoted by the dashed line).

Suppose we have a class named Source. When add () method of Source gets called internally it calls validate () method on the same object (of same Source class). Below diagram shows this:



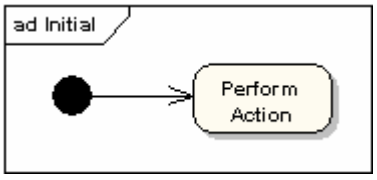
Note that recursive call (usually calls same method multiple times) also is being shown in the above diagram.

2.2.3 Activity and State diagram

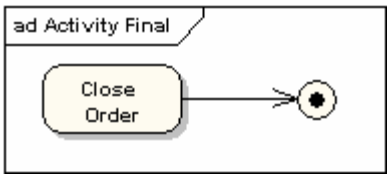
Like sequence diagram, activity diagram also shows the interaction between objects. It is recommended to use activity and state diagram when we need to show the sequence of events on a broader scale. Activity diagrams are useful for business modelling where they are used for detailing the processes involved in business activities.

Activity:
An activity is the execution of a task whether it is a physical activity or the execution of code. An activity is shown as a round-cornered rectangle:

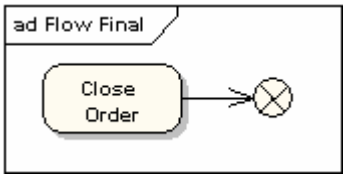
Start Point:
Each activity diagram has one start at which the sequence of actions begins.



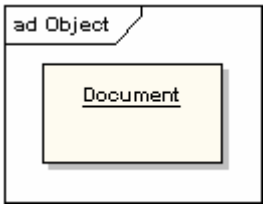
End Point:
Each activity diagram has one finish at which the sequence of actions ends. There are two types of final node: activity and flow final nodes. The activity final node is depicted as a circle with a dot inside.



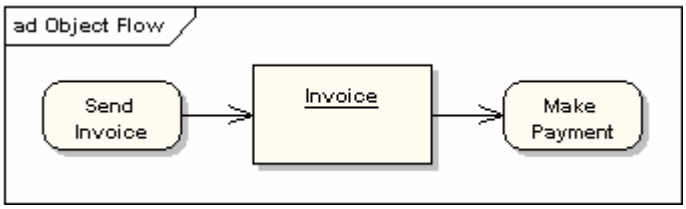
The flow final node is depicted as a circle with a cross inside.



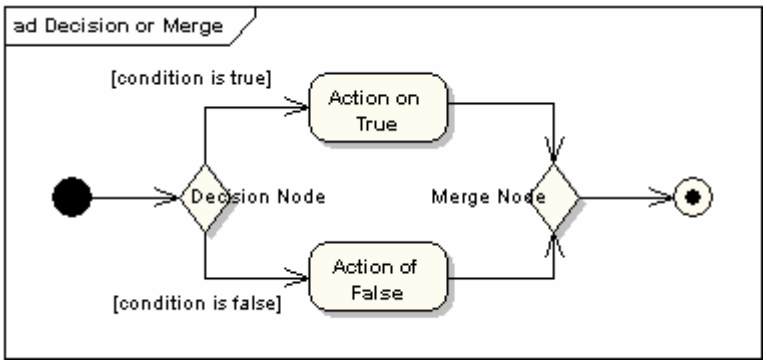
Objects and Object Flows:
An object is shown as a rectangle:



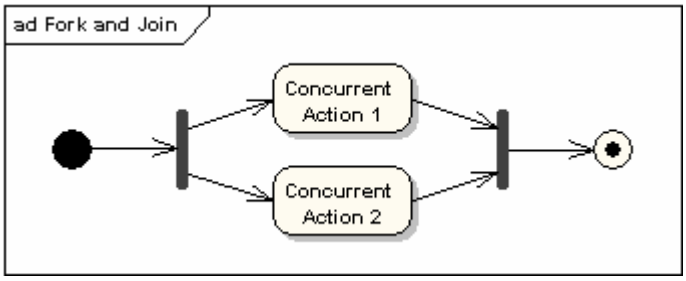
An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed:



Decision and Merge Nodes:
Decision nodes and merge nodes have the same notation: a diamond shape. They can both be named. The control flows coming away from a decision node will have guard conditions which will allow control to flow if the guard condition is met. The following diagram shows use of a decision node and a merge node:



Fork and Join Nodes:
Forks and joins have the same notation: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. The following diagram shows an example of their use:



Difference between Merge and Join:
A join is different from a merge in that the join synchronizes two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received. A merge passes any control flows straight through it. If two or more inflows are received by a merge symbol, the action pointed to by its outflow is executed two or more times.

3 Case Study

In this section we will state the steps of a very simple use case – Logon. We will draw class diagrams and then the sequence diagram for the same. The main flow of Logon use case can be specified by the following steps:

1. Logon dialog is shown
2. User enters user name and password
3. User clicks on OK or presses the enter key
4. The user name and password are checked and approved
5. The user is allowed into the system

Alternative flow: Logon Failed - if at step 4 the user name and password are not approved, allow the user to try again.

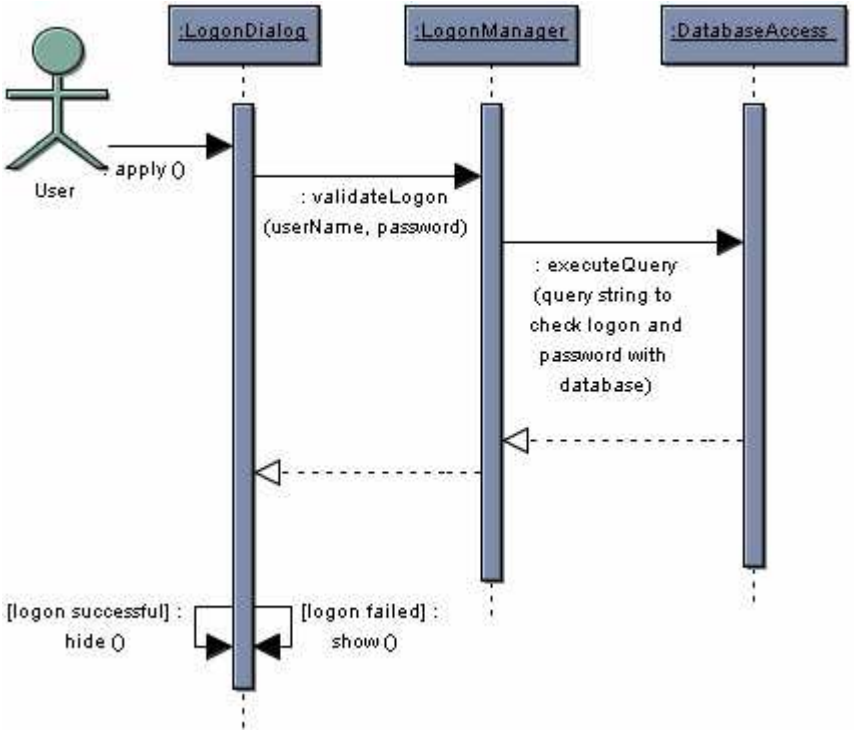
With above requirement we can think of three classes (class diagrams are also shown):

1. LogonDialog: Represents the logon screen where user enters user id and password. We can show or hide the dialog.

2. LogonManager: Validates user id and password and returns the Boolean value which indicates successful login or failure.

3. DatabaseAccess: The class which allows us to access database.

Now let's draw the complete sequence diagram:



The dotted lines in the opposite indicates return path. Though above diagram is self descriptive still we will go through steps of execution:

- Actor User enters user id and password and clicks on the submit button.
- apply () method of LogonDialog is being called, which in turn calls validateLogon () operation of LogonManager and this internally calls executeQuery () method of DatabaseAccess.
- executeQuery () method returns to validateLogon () method.
- validateLogon () method returns to apply () method.
- Based on the Boolean value (indicates success or failure) a decision is being taken. If successful then calls hide () and if failure then calls show () in order to hide or display the same screen to the user. Note that, both hide () and show () are self messages (method call on the same object).

4 Conclusion

In this document we have covered most useful and important part of UML. With this knowledge now we should be able to draw use case diagram, class diagrams and sequence diagram for a given requirement. In the next version of this document we will look into other diagrams and also we will focus on how best UML diagrams can be used in SOA based designs.