Lecture Notes: Object Oriented Analysis And Design

1.1 Introduction to OOAD and UML

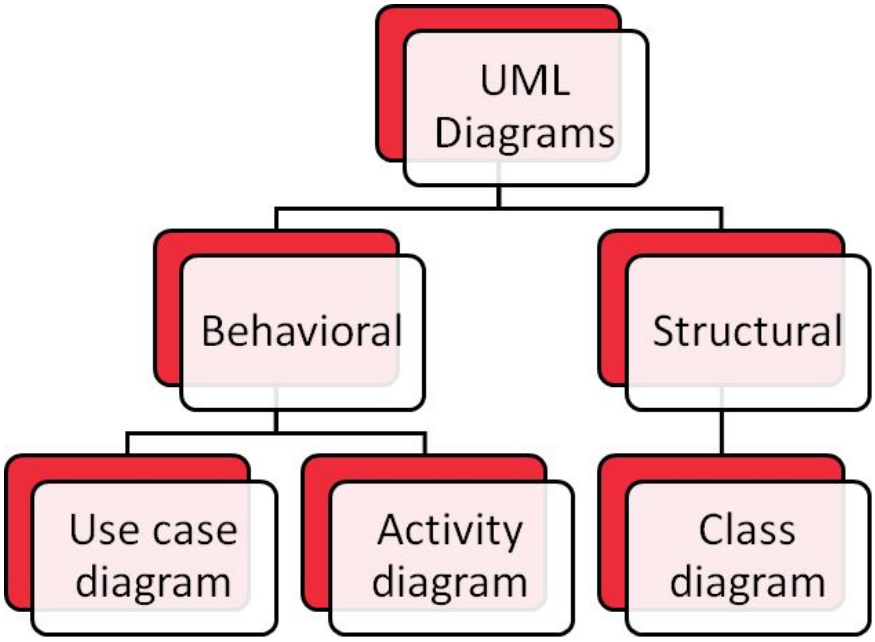
UML: Unified modeling language is a common language for all software developers to visualize the system and communicate with their peers for developmental and business purposes.

OOAD: Object oriented analysis and design is a methodology which has two parts: The analysis and the design. The analysis is done during the requirement gathering part, where you work with the users of the system or the software to define the functionality of the system. Post analysis, you design the system by refining the analysis models created during the analysis phase.

Why OOAD:

1. To keep all stakeholders on the same page
2. Help the product evolve
3. Transfer knowledge (documentation)

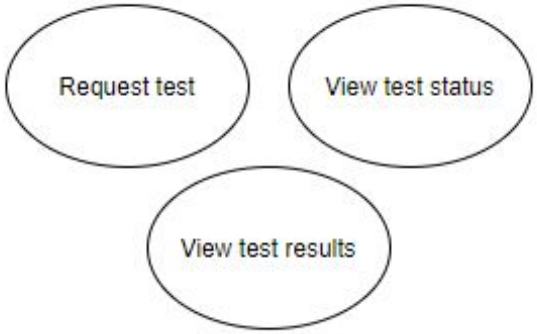
Object oriented programming: Attributes are properties and methods define the actions of an object.



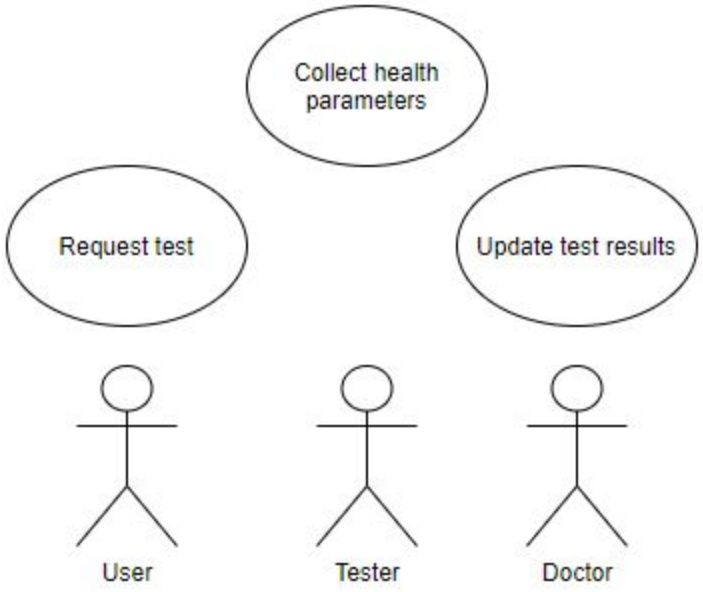
1.2 Use case diagram

Use case diagram is a type of behavioral diagram and is used for requirement gathering.
The following are the different elements of the use case diagram:

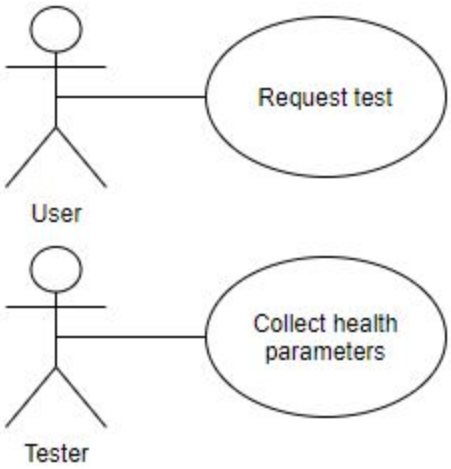
1. **Use cases**: Use cases are functionalities or features that a software will provide. It can be thought of as the answer to the question: *What will my software be used for?*
   Use cases have no particular order.



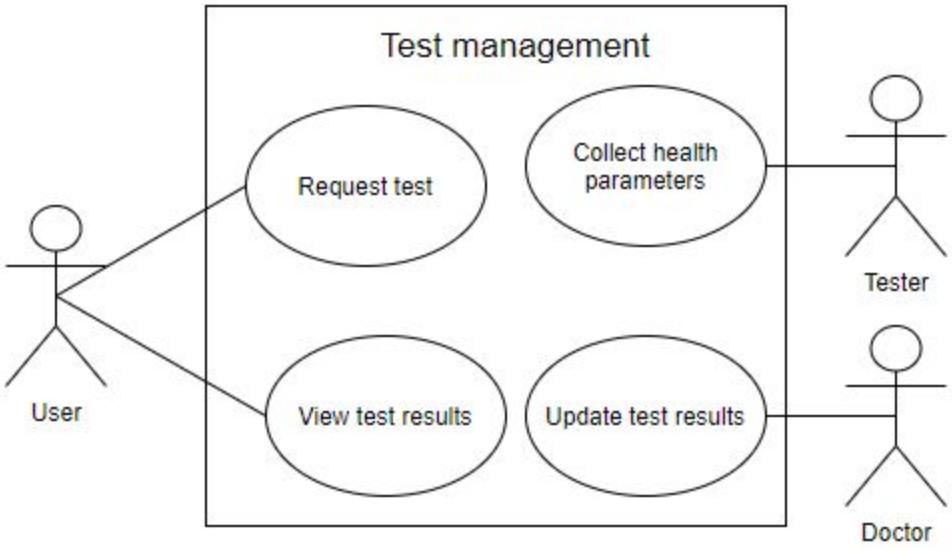
2. **Actor:** Actors are the external entities which invoke the use cases. Actors may or may not be humans. They can also be a 3rd party service (eg. Notification system). Actors are represented by stick figures.



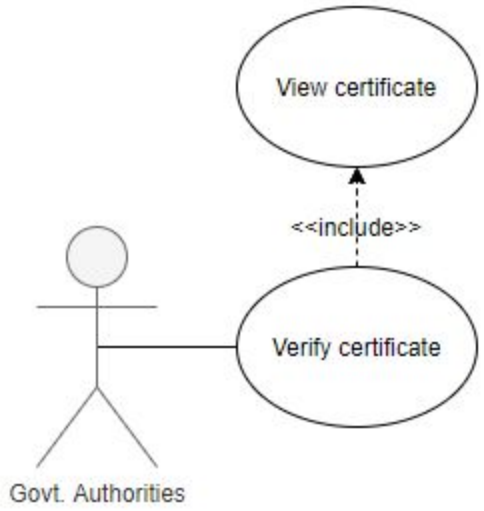
3. **Associations:** Actors are related to use cases by using associations. The associations are depicted by drawing lines between actors and use cases. One use case can be associated with multiple actors and one actor can be associated with more than one use case.



4. **System boundary:** A system boundary is drawn around the use cases, to separate the use cases from the actors. This indicates that use cases are internal to the system and actors are external to the system.

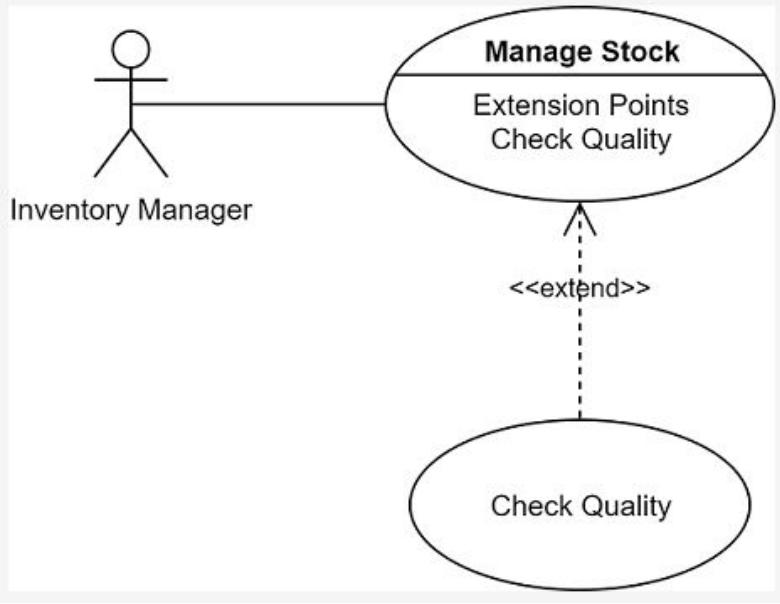


5. **Include:** Include relationships can be used to represent use cases which are connected with each other. In an include relationship, the arrow starts from the including use case and points towards the included use case. Before an actor can execute the including use case, they have to first execute the included use case.



In the above example, the Government authority has to view the certificate before he/she can verify it. So before executing *verify certificate*, which is the including use case, the government authority has to execute *view certificate*, which is the included use case.

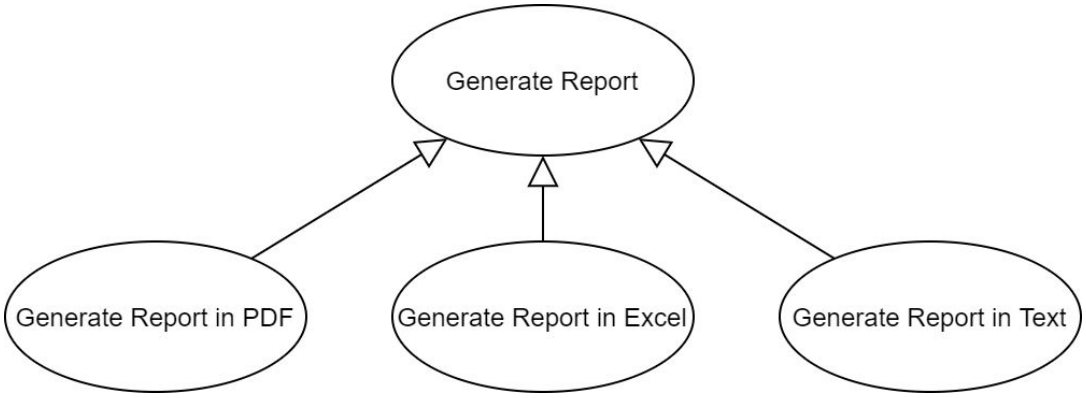
6. **Extend relationship:** In an extend relationship, the arrow starts from the extending use case and points towards the extended use case. Before an actor can execute the extended use case, they may or may not first execute the extending use case.



In the above example, whenever the inventory manager is managing the stock using the *Manage Stock* use case, they can first check the quality using the *Check Quality* use case, but it is not mandatory.

7. **Generalisation relationship:** In a generalisation relationship, the common functionalities of different use cases can be clubbed under a parent use case.

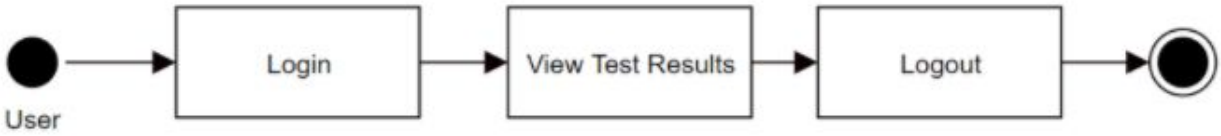
Other use cases that modify the behaviour from the parent use case are called children use cases.



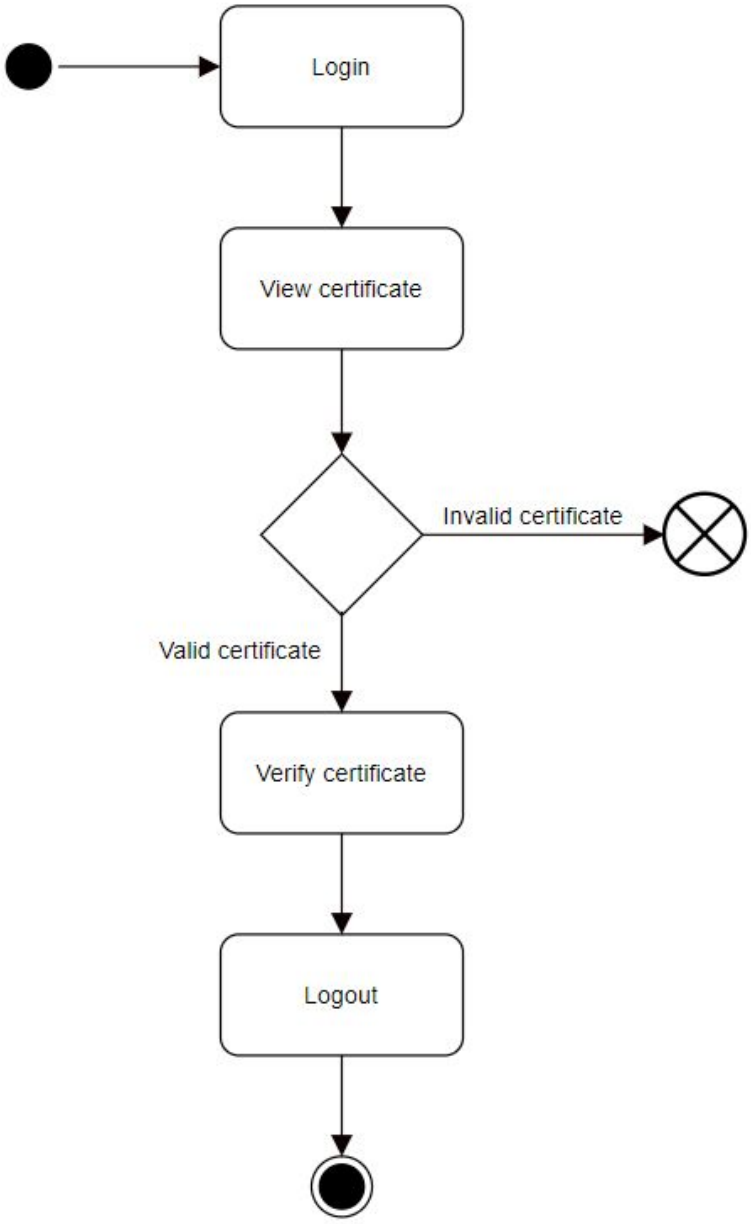
1.3 Activity diagram

Activity diagrams are pictorial representations of use case diagrams. They contain additional information, like the order of execution of use cases and also certain constraints on these use cases. The following are the elements of the use case diagram:

1. **Initial node, actions and activity final node**: The initial node is where the flow begins. It is represented by a solid circle. Actions are single units of behaviour performed by the system. The final node is where the flow ends and is represented by a solid circle inside a hollow circle.

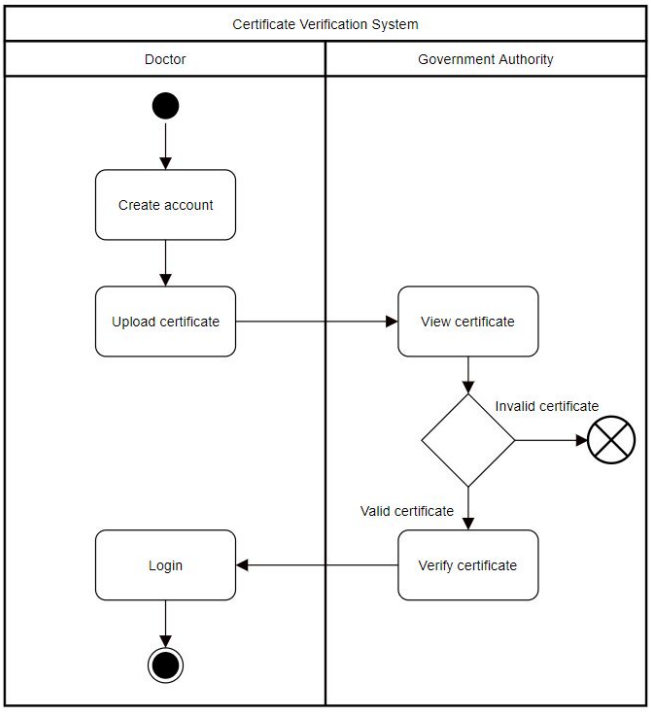


2. **Decision and merge node:** This is similar to an if-else condition, where the activity flow takes different routes for different conditions. The decision node is represented by a diamond, which has a single incoming flow and multiple outgoing flows. The direction of an outgoing flow is decided on the basis of the guard conditions.



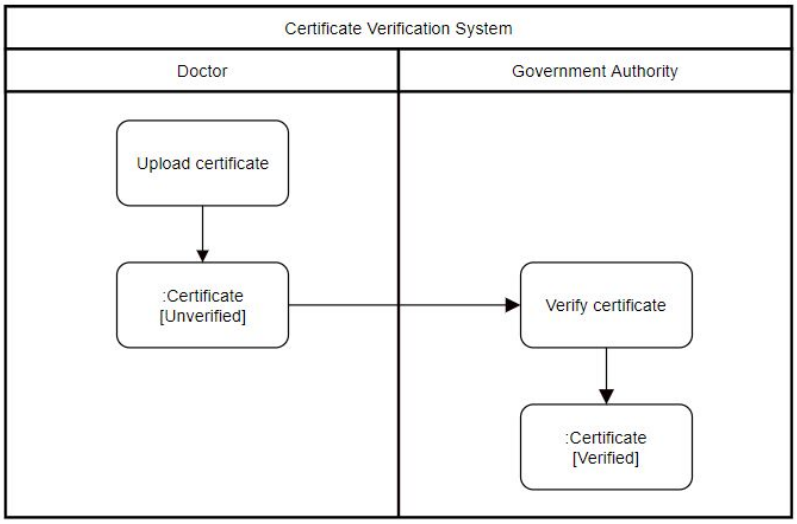
In the above example, the guard conditions are *Valid certificate & Invalid certificate*. When *Invalid certificate* is true, the flow terminates. If *Valid certificate* is true, the flow continues.

3. **Partitions:** Partitions are used to distinguish between various responsible parties and group the actions performed by the same responsible party.



4. **Fork and Join:** Fork is similar to a decision node with one input flow and multiple output flows. The difference is that, in the case of Fork all the outflows happen concurrently. Joins are the inverse of forks, with multiple input flows and a single output flow.

5. **Object flow:** Object flow is a representation of how a particular object gets updated after an activity. They are



In the above example, the object *Certificate* takes two states: *Unverified* and *Verified*. It takes the *Unverified* state before it is verified by the Government authority and takes the *Verified* state after it is verified by the Government authority.

1.4 Class diagram

A class diagram is used to show what classes will be present in the system and how they will be related to each other. Class diagrams represent the attributes and methods present inside classes and the relationships between the classes themselves.
The following are the different elements in a class diagram:

1. **Class, attributes and methods:** The class name is written on top. The box contains two partitions with attributes on the top partition and the methods on the bottom partition. The attributes are mentioned with the type. The methods may contain parameters and have a return type. The following template is an example of how classes are represented in class diagrams.

ClassName
attribute1: type attribute2: type
method1(parameter1 : type) : returnType method2(parameter1 : type) : returnType

2. **Visibility:** Visibility (also called access specifier) specifies how attributes and methods of a class will be accessed by different classes.Visibility is mainly of two types:
   a. Public (+) = Attributes and methods that are marked 'public' can be accessed from within or outside that class.
   b. Private (-) = Attributes and methods that are marked 'private' can be accessed only from within that class.

Order
+ orderId : int + orderName : String + databaseAddress : String
+ getTotalCost(productName : String, quantity : int) : int + checkStatus() : String - checkDatabase() : String - checkWithSupplier() : String

3. **Multiplicity:** Multiplicity is used to represent an array of objects.The syntax for multiplicity is *ClassName [minValue..maxValue]*

Order
+ name : String + userId : String + address : String - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

4. **Default values:** Some of these attributes can have a default value, which can be represented as *attributeName : type = defaultValue*.

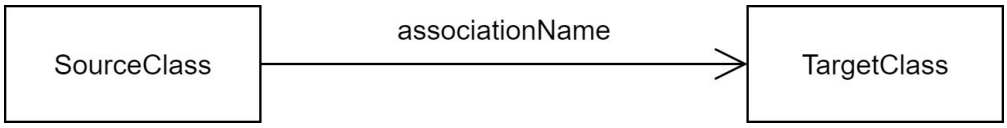
Order
+ name : String + userId : String + address : String = "Inventory Address" - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

5. **Property strings:** Attributes can also have properties. This can be represented as *attributeName : type = {PropertyString}*.

Order
+ name : String + userId : String = {unique} + address : String = "Inventory Address" - inventoryManagers : InventoryManager [5..20]
+ addInventoryManager() : void + updateInventoryManager() : void + removeInventoryManager() : void

Apart from the elements of the class, class diagrams also represent the relationship between classes. There are 4 types of relationships:

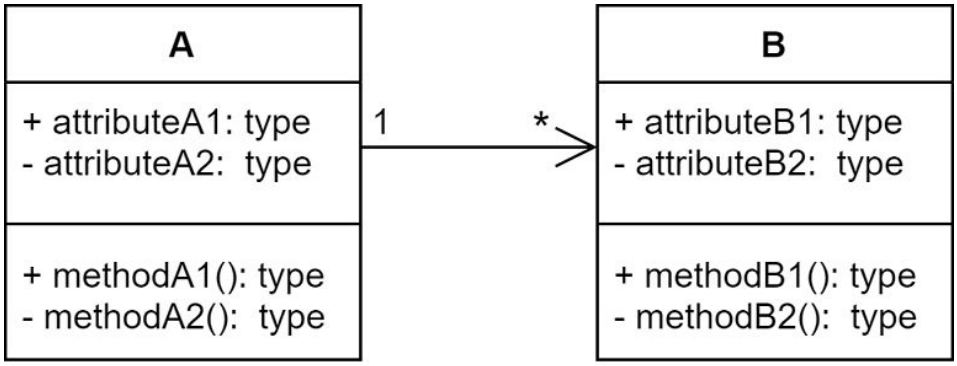
1. **Association relationship:** Association relationships are of two types: Unidirectional (as shown below) and bi-directional.



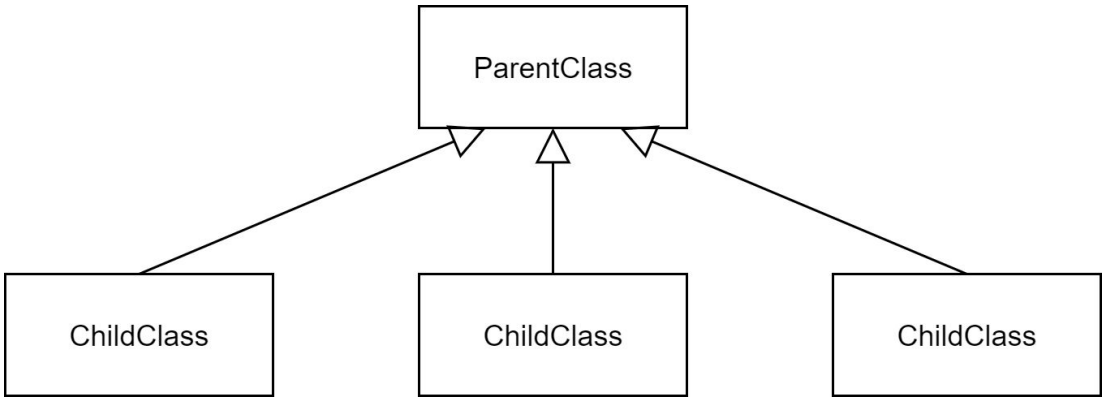
Association relationships can also have multiplicity associated with it, which is of 3 types:

- **a.** One-to-one
- **b.** One-to-many
- **c.** Many-to-many

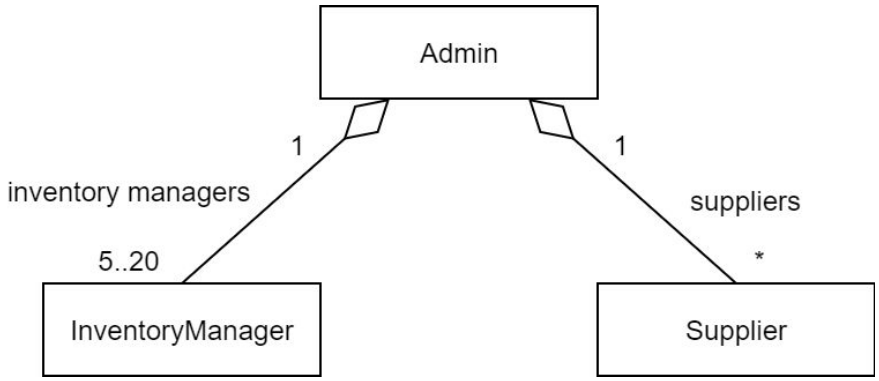
The following is an illustration association relationship between classes which is one-to-many.



2. **Generalisation Relationship:** A generalisation relationship has a parent class and multiple children classes. Using a generalisation relationship, you can put common attributes and methods of several classes into one general class, which saves you from repeating the same code and also allows code reusability.

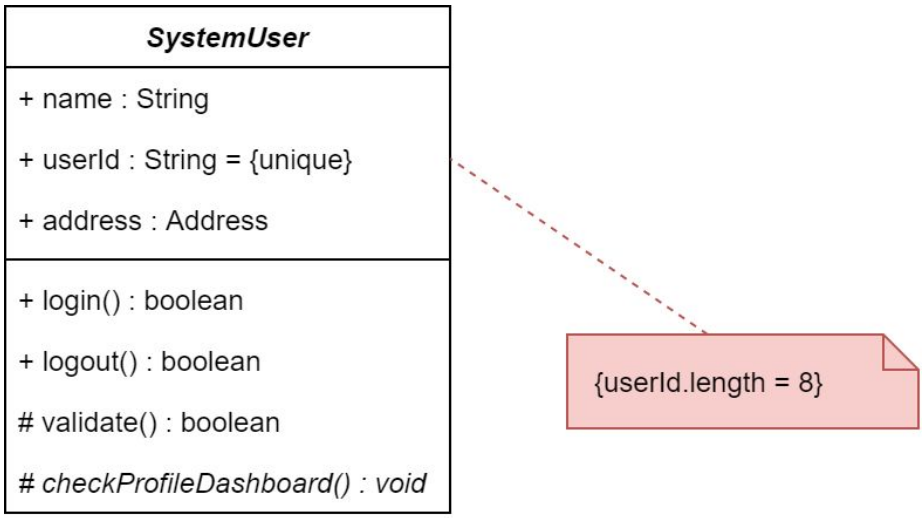


3. **Aggregation:** In an aggregation relationship, an object of one class (whole object) will contain objects of other classes (part objects). But when the whole object is deleted, part objects will not be deleted. An aggregation relationship has an association name and multiplicity associated with it.

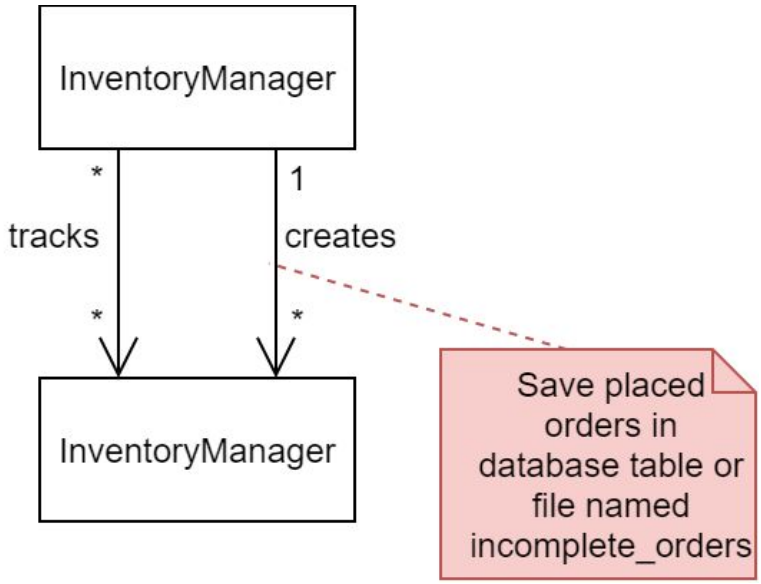


4. **Composition:** Similar to aggregation relationship, in a composition relationship, an object of one class (whole object) will contain objects of other classes (part objects). But in a composition relationship, when the whole object is deleted, part objects will also be deleted. A composition relationship also has an association name and multiplicity associated with it.

Constraints and notes can also be represented in class diagrams. Constraints refer to the conditions that must be preserved while writing code for that class. For example, if you want that userId for every user must be 8 characters long, then such condition you can specify a class diagram as shown below.



Notes are used to write decisions or assumptions made while designing your system. For example, while designing your system you decided that you will save placed orders in a database table called "incomplete_orders". Such decisions can be noted down in the class diagram as shown below.



Association class: Association class is used to capture the properties of the relationship between different classes. For example, when an Admin pays an invoice raised by a supplier, a transaction takes place. This transaction will have several properties which can be captured in the class diagram as shown below.