**UML - 2022**

**Dependency** is often confused as Association. Dependency is normally created when you receive a reference to a class as part of a particular operation / method. **Dependency** is a [**directed relationship**](https://www.uml-diagrams.org/uml-core.html#directed-relationship) which is used to show that some UML element or a set of elements requires, needs or depends on other model elements for **specification** or **implementation**.

Symbol :

public class Order {

public void processPayment(PaymentSystem ps) {

}

}

public class PaymentSystem {

}

If A---->B that means that a method in A has a parameter of type B, or that a method of A creates an instance of B to do some task and then destroy it.

**Dependency is a loose form of Association**. It is "optional" to show dependency in class diagrams. The best example will be a **Teacher teaches a subject of Physics**. The java code is given below.

public class Subject {

private String subjectName = "Physics";

public String getSubjectName() {

return subjectName;

}

}

public class Teacher {

public String teaches( Subject subject ) {

return subject.getSubjectName();

}

}

**Association:**  
The association represents the static relationship between two classes along with the multiplicity. E.g. an employee can have one primary address associated with it but can have multiple mobile numbers. Association are represented as thin line connecting two classes. Association can be unidirectional (shown by arrow at one end) or bidirectional (shown by arrow at both end).

Symbol :

|  |  |  |
| --- | --- | --- |
| 0..1 | No instances or one instance | A flight seat can have no or one passenger only |
| 1 | Exactly one instance | An order can have only one customer |
| 0..\* or \* | Zero or more instances | A class can have zero or more students. |
| 1..\* | One or more instances (at least one) | A flight can have one or more passenger |

The unidirectional relationship shows that the source object can invoke methods of the destination class. In Java a possible example can be the instance variable of source class referencing the destination class.Association Example

public class Car {

private String modelNumber;

private Customer owner;

}

public class Customer {

private String name;

private String address;

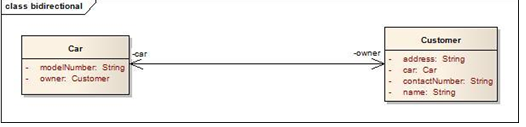
private String contactNumber;

}

Diagram

Description automatically generated

Let’s look at an example of bidirectional association: Bidirectional association



public class Car {

private String modelNumber;

private Customer owner;

}

public class Customer {

private String name;

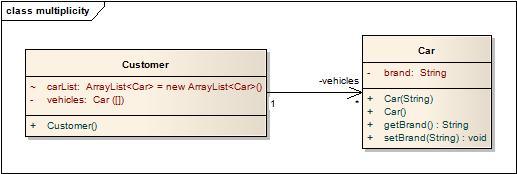
private String address;

private String contactNumber;

private Car car;

}

In the bidirectional association each of the class in this relationship refers to each other by calling each others method. In the above Java example it is depicted as instance variable of Car class in called inside the Customer class and vice versa. In the above example the car and owner refers to the roles and is depicted by the name of instance variable in the code.  
**Multiplicity:** Assume a scenario where a customer has multiple cars. How do we represent this situation in Java and UML?

[](http://i2.wp.com/idiotechie.com/wp-content/uploads/2012/12/multiplicity-in-association.jpg)

Multiplicity in association

The above diagram explains a unidirectional association with a one to may relationship. Both use of ArrayList and Array is for illustration purposes only.

**An association is a "has a" relationship.** It implies that one class retains a long lasting relationship to another object. Most of the times this implies that a class has an attribute ( a field member) which is the type of the second class, then A has a B. Example will be Class Man uses a Class Pen for writing.

Class Client{

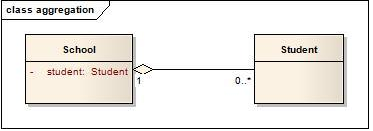
protected String name;

}

**An association is used when one object wants another object to perform a service for it.** Examples are given.

* Employee uses BusService for transportation.
* Client-Server model.
* Computer uses keyboard as input device.

**Aggregation**: **has a relationship**. Example will be a Class Man has a Class Car ( Car is still there when Man dies ) . This shows “has a” relationship. It is a form of association relationship. This relationship highlights that a whole is made of its parts. **So if a whole is destroyed the part still remains**. In UML this is represented through a hollow diamond with the diamond symbol pointing towards the whole.  
In case of Java the aggregation follows the same structure as association. It is represented through the instance variables of a class.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg  


public class School {

private Student student;

}

public class Student {

}

In this case a student is a part of the School. However during design it is preferred to use association instead of aggregation as it is not a recommended option.

**Composition** : Composition: owns a – Strong Relationship

Ex:a Class Man owns a Class Heart ( When Man die, Heart die ) **Diagram**: http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

Composite aggregation is depicted as a binary association decorated with a **filled black diamond** at the aggregate (whole) end.

Composite aggregation is depicted as binary associations with filled black diamond

Folder could contain many files, while each File has exactly one Folder parent. **If Folder is deleted, all contained Files are deleted as well**. In cases where in addition to the part-of relationship between ClassA and ClassB - there’s a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link. Java code for Composition is given below.

public class Heart {

private String condition;

public String getCondition() {

return condition;

}

}

public class Head {

private String condition;

public String getCondition() {

return condition;

}

}

[](http://lh5.ggpht.com/_aUOgqE3fGXc/Sh35YNKDw9I/AAAAAAAAAao/E4v4uDJcD5w/s1600-h/image%5b5%5d.png)

public class Person {

private Heart heart;

private Head head;

private Leg[] legs;

public Person() {

heart = new Heart();

head = new Head();

legs = new Leg[2];

}

}

public class Leg {

private String leftLegName;

private String rightLegName;

public String getLeftLegName() {

return leftLegName;

}

public String getRightLegName() {

return rightLegName;

}

}

**Generalization : IS-A** example will be a Class Man is a Class Human ( Man is a Human ) , Crow is a Bird. This property represents the inheritance feature of the object oriented concept. In Java this can relate to the “extends” keyword. The inheritance should ideally follow the Liskov Substitution Principle i.e. the subtype should be able to substitute for its supertype. It helps to make the code implicitly follow the Open Close Principle i.e. Open for extension but closed for modification.

http://javapapers.com/wp-content/uploads/2010/06/generalization.jpg

Java code for Generalization is given below.

**public** **class** Bird {

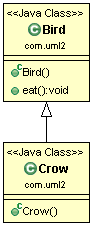
**public** **void** eat() {

}

}

**public** **class** Crow **extends** Bird {

}



**Realization (Interface in Java)**

This is related to the relationship between the class and the interface. **The realization is equivalent to the “*implements*” keyword in Java**.

http://javapapers.com/wp-content/uploads/2010/06/realization.jpg

Java code for Realization

**public** **class** **AccountValidatorImpl** **implements** Validator {

@Override

**public** **boolean** isValid() {

**return** **false**;

}

}

**public** **interface** **Validator** {

**public** **boolean** isValid();

}

**public** **class** LoanValidatorImpl **implements** Validator {

@Override

**public** **boolean** isValid() {

**return** **false**;

}

}

