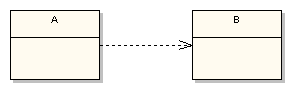
**UML Class Relationship**

1. Dependency
2. Association
3. Aggregation
4. Composition
5. Inheritance
6. Realization

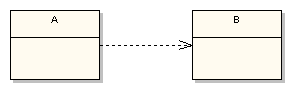
**How to interpret the relation:**

**Source ------------> Target**



Eg: Class A depends on Class B

1. **Dependency**



Dependency means that class A uses class B, but that class A does not contain an instance of class B as part of its own state.

It also means that if class B’s interface changes it will likely impact class A and require it to change.

I suggest that you constrain your use of dependency relationships to non-state related concerns.

You would use dependency to indicate that, for example, **class A receives an instance of class B as a parameter to at least one of its methods**.

You would also use dependency to indicate that **class A creates an instance of class B local to one of its methods (on the stack).**

You would not, however, use dependency to indicate that class A declares an instance variable of class B, as that would indicate a state-related concern (state of an object i.e. member variables). Again, use *association* to do that.

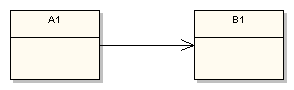
public class A {

public void method1 (B b) { // . . . }

public void method2 () { B tempB = new B()

}

1. **Association**



**Association defines dependency**, but a much stronger dependency than that described above with the plain *dependency* relationship.

The arrowhead means that there is a one-way relationship.

In this example it means that class A1 is associated with class B1. In other words, **class A1 uses and contains one instance of class B1**, but B1 does not know about or contain any instances of class A1.

public class A1 {

private B1 b1;

public B1 getB1() {

return b1;

}

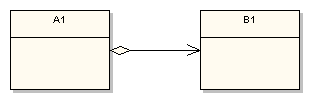
}

**Based on lifetime of the instances that make up the dependent object’s state they are further classified as:**

2. a. Aggregation

2. b. Composition

**2. a. Aggregation**

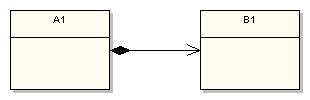


**A1 aggregates B1.**

Aggregation describes an association **where an instance of A1 contains a reference to an instance of B1** as part of the A1’s state, but the use of the specific instance of B1 is or may be shared among other aggregators.

A shared association means that the lifetime of the aggregated object, the instance of B1 in this case, is outside the scope of the referencing object. Therefore, **when a specific instance of A1 goes out of scope (e.g. garbage collected), the instance of B1 does not (of necessity) go out of scope.**

**2. b. Composition**



Composition on the other hand defines a relationship where the **scope of the containing object (an A1) and the contained object (a B1) is related**.

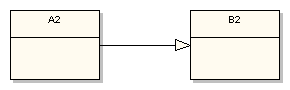
**When the containing object goes out of scope, then the contained object also goes out of scope**.

The composition adornment looks like the aggregation adornment, except the composition adornment is darkened.

**Difference between Aggregation and Composition:**

|  |  |
| --- | --- |
| **Aggregation** | **Composition** |
| final class Car {  private Engine engine;  void setEngine(Engine engine) {  this.engine = engine;  }  void move() {  if (engine != null)  engine.work();  }  } | final class Car {  private final Engine engine;  Car(EngineSpecs specs) {  engine = new Engine(specs);  }  void move() {  engine.work();  }  } |
| Car also performs its functions through an Engine, but the Engine is not always an internal part of the Car. | Engine is completely encapsulated by the Car. There is no way for the outside world to get a reference to the Engine |
| **Engines may be swapped, or even completely removed** | **Engine lives and dies with the car** |
| The outside world can still have a reference to the Engine, and tinker with it regardless of whether it's in the Car |  |
| Instance is created by some other class and referenced through getInstance() or returned from some function. | Instance is created in constructor. |

1. **Inheritance / Generalization**



UML generalization is one of the better-understood relationships, and symbolizes what is known as inheritance in the world of object-oriented programming. It is sometimes also called specialization because the subclass is a specialization of the more generic super class.

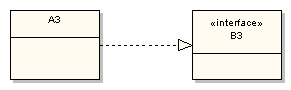
public class A2 : public B2 {

// . . .

}

B2 is the super class and A2 is the subclass in the relationship. Just remember that the generalization symbol forms a line from the subclass to its super class with the clear triangular arrowhead pointing at the super class.

1. **Realization**



This relationship is somewhat related to generalization, but a bit different.

In object-oriented programming parlance **realization represents the implementation of an interface by a class**.

So it represents how some characteristics of a class are defined, but says nothing about the implementation details

This diagram fragment states that class A3 implements or realizes the interface defined by B3

public class A3 implements B3 {

// . . .

}

**References**:

<https://vaughnvernon.co/?page_id=31>

<http://stackoverflow.com/questions/11881552/implementation-difference-between-aggregation-and-composition-in-java>