UML Class Diagrams and Relationships

This document is intended to provide a bit more explanation regarding the relationships between and among classes in UML class diagrams.

1. Class Association Relationship

[Ref: 1.a, 2, 3, 4] An **association relationship** exits between two classes *A* and *B* if there is some kind of linkage between *A* and *B*. For example, suppose we have two classes *Automobile* and *Tire* where in the real world, an automobile normally has four tires. Then there is a link or association between the *Automobile* and *Tire* classes.

An association relationship says nothing about declaration of instance data in either *A* or *B* or the **life cycles** of instances of *A* and *B*. In Java, the life cycle of an object of a class is the time between when the object is instantiated using the **new** keyword and when the object is deleted from memory by the garbage collector [5]. In an association between an object of *A* named *a* and an object of *B* named *b*: (1) *a* may be deleted and due to the relationship between *a* and *b*, *b* may also be deleted at the same time; (2) *a* may be deleted but *b* may live on; (3) and vice versa, *b* may be deleted and *a* may also be deleted at the same time; and (4) *b* may be deleted and *a* may live on.

An association is drawn in UML as a solid line connecting the two classes. Multiplicities may be drawn on either end of the line; e.g., we would draw a solid line connecting *Automobile* and *Tire* and on the *Automobile* end of the line we would draw **1** above the line; then on the *Tire* end we would draw **4** above the line. In English, this says that one (1) automobile is associated with four (4) tires.

By the way, this type of association between two classes is called a **binary association**. In UML, higher degree associations may be drawn but we will not discuss those. The next three types of relationships—class dependency, composition, aggregation—are all specialized forms of association.

2. Class Dependency Relationship

[Ref: 1.b] Picture two classes: *Client* and *Supplier* where a *Supplier* supplies some sort of information to a *Client*. A **class dependency relationship** exists between two classes if changes to the definition of one class (the *Supplier*) may cause changes to the other class (the *Client*).

We can also say a dependency relationship exists between *Client* and *Supplier* if class *Client* somehow "**uses**" class *Supplier*, e.g., the type of an input parameter to a method in class *Client* is of the class *Supplier*. Since class *Client* **uses** *Supplier,* if the code in *Supplier* changes, then we may need to also make changes to *Client*. For example, *Supplier* may contain a method named *public int supplyX*(*int a*) which has to be modified so now it has two parameters resulting in *public int supplyX*(*int a, int b*). Suppose a method in *Client* calls the old *supplyX* method by passing one parameter. Now, because *supplyX*() of *Supplier* has changed, we need to modify the code in *Client* and change the method call to *supplyX*() to have two parameters.

Dependency can also be called a "**knows about**" relationship. Class *Client* **knows about** *Supplier* because *Client* uses *Supplier*.

A dependency relationship is drawn in UML as a dashed line with an "open" arrowhead at the end of the dashed line, with the arrowhead pointing toward the class on which the other class depends, i.e.., the arrowhead would point toward class *Supplier*. To put the UML diagram into words then, you can look at the class without the arrowhead (*Client*) and say to yourself, "*Client* **depends on**", and then look at the class with the arrowhead, *Supplier*, and say, "*Supplier*", to complete the sentence, "*Client* **depends on** *Supplier*." Or, you can say "*Client* **knows about** *Supplier*." Or say, "*Client* **uses** *Supplier*." All would be correct.

3. Composition Relationship

[Ref: 1.c, 2, 3] A composition relationship exists between two classes *Composite* and *Composed* when *Composite* declares instance data which is of the class *Composed* and—this is very important—a *Composite* object instantiates the *Composed* objects, so consequently, when a *Composite* object is deleted (in Java, all prior references to the object no longer exists and the objects is going to be garbage collected) all of the *Composed* instance variables are also deleted. We can say that the **life cycles** of the *Composite* object and the *Composed* object(s) **are the same**. We will discuss **aggregation relationships** in §4 which is very similar to composition relationships with one of the primary differences being that in aggregation, the *Composed* object(s) outlive the *Composite* object, i.e, the life cycles of the *Composite* object and the *Composed* object(s) **are not the same**.

For example, suppose we have classes named *Rectangle* and *Line*. Geometrically, a rectangle is **composed of** four lines so *Rectangle* would be the compositing class (equivalent to *Composite*) and *Line* would be the composed class (equivalent to *Composed*). Now, let the *Rectangle* class contain four instance variables, *mLine1*, *mLine2*, *mLine3*, and *mLine4* (I commonly preface class data members with an **m** which stands for **m**ember), each of which is an instance of the *Line* class. In the *Rectangle* constructor, *mLine1*–*mLine4* are instantiated by a statement such as *mLine1* = new *Line*(...). Because *mLine*–*mLine4* are declared and created within the *Rectangle* object, when the *Rectangle* object is deleted, *mLine*–*mLine4* will also be deleted.

Composition is often referred to as a "**has-a**" relationship, e.g., we can say a *Rectangle* **has a** *Line*, and more specifically, a *Rectangle* **has** four *Lines*.

In UML class diagrams, composition is drawn with a solid line connecting the *Composite* and *Composed* classes, with a solid, shaded diamond symbol drawn on the end of the line at the *Composite* class.

4. Aggregation Relationship

[Ref: 1.d, 2, 3] Aggregation and composition relationships are very closely related. To understand an aggregation relationship, it may be helpful to first look at the dictionary definition of **aggregate** (noun): *a "whole" which is formed by combining several separate elements or parts*. For example, a galaxy (the whole) is formed as a combination of stars and gas (the several separate elements or parts). In object-orientation, we might have a class named *Triangle* and a class named *Line*. Geometrically, a triangle consists of (or aggregates) three lines, so we can say *Triangle* aggregates *Line*. *Triangle* is the **aggregating class** and *Line* is the **aggregate class**.

Aggregation, like composition, is often referred to as a "**has a**" relationship, e.g., we can say a *Triangle* **has a** *Line*, and more specifically, a *Triangle* **has** three *Lines*. An aggregation relationship is drawn in UML as a solid line connecting the two related classes with an unshaded diamond symbol drawn on the end of the line closest to the aggregating class. That is, we would draw a solid line between *Triangle* and *Line* and on the end of the line closest to *Triangle* we would draw an unshaded diamond symbol. In OO code, an aggregating relationship is formed when, e.g., class *Triangle* declares three instance variables of the class *Line*. This is similar to the composition relationship where *Composition* declares instance variables of the *Composite* class.

There seems to be much confusion and disagreement in the UML community regarding exactly what the definition of aggregation should be, especially in regard to aggregation's very similar cousin, composition. As we mentioned in §3, a commonly accepted difference between aggregation and composition concerns the life cycles of the objects: (1) In composition where class *C* is composed of class *D*, an object of class *C* declares one or more data members which are of class *D* and then instantiates those data members within some method of *C*, usually a constructor; when an object of *C* is deleted, the data members of *D* are also deleted. (2) In aggregation, suppose we have the same classes *C* and *D*, when an object of *C* is deleted, the data members of *D* are not deleted.

Note also that in aggregation, the aggregating class commonly declares an instance data member which is a collection of the aggregate class objects. For example, rather than declaring three *Line* data members named *mLine1*, *mLine2*, and *mLine3* we could declare the data member *Line*[] *mLines* where *mLines* will be a simple array with three elements, each of which is a *Line* object. Composition may also declare such data members but it appears to be more common with aggregation.

5. Inheritance or Generalization Relationship
[Ref: 1.e, 6] As we did with aggregation, to understand generalization relationships, it may be help to look at the dictionary definition of **generalization** (noun): *a general concept obtained by inference from specific cases.* Another and perhaps better definition is: *the process of formulating general concepts by inferring common properties of specific instances.* In either definition, note that the key terms are **general** and **specific**.

For example, it is a **generalization** to say that all automobiles have tires (I have never seen one that is drivable and does not have tires). Or we can say that we know that all automobiles **specifically** have tires (I have never seen one that drives on Lego blocks or bananas). Or, **in general**, all automobiles have various parts and **specifically**, one of those parts is the tires. (Note that there may be some automobiles which do not have tires but the internet does not know that as my Google search for "automobiles without tires" produced page hits discussing how many automotive companies are saving a few pennies by not including a spare tire in new cars.)

A generalization relationship is commonly referred to as an "**is-a**" relationship, where the left-hand side of is-a represents a specific concept and the right-hand side reflects a general concept. For example, a car **is-a** vehicle. On the left-hand side of that sentence, *car* is something that is specific and *vehicle* is something is more general (which includes boats and other transportation objects). Consequently, we might have a class named *Car* and a class named *Vehicle*. Class *Vehicle*, being a generalization of *Car* would define properties that are common to all *Car*s, e.g., has four tires, has a steering wheel, moves, and so on.

Another, more common, term for generalization is **inheritance**. Because *Vehicle* is a generalization of *Car*, *Vehicle* would declare properties that are common to all vehicles but not properties that are specific to cars. Class *Car* would inherit from *Vehicle* and would declare properties that are specific to cars. So we can say that *Car* **inherits from** *Vehicle*. In Java, *Vehicle* is referred to as the **superclass** and *Car* as the **subclass**, so a subclass inherits from a superclass. Another more generic term is **parent** for the class that is being inherited from (*Vehicle*) and **child** for the class that is inheriting from the parent class (*Car*).

In UML, a generalization or inheritance relationship is drawn with a solid line connecting the two classes and with a closed, unfilled arrow on the end of the line next to the inheritance class, i.e., the superclass.

References

1. UML Class and Object Diagrams Overview (this appears to be very comprehensive website)
    a. UML Association
    b. UML Dependency
    c. UML Compositon
    d. UML Aggregation
    e. UML Generalization
2. Design Codes: UML Class Diagram - Association, Aggregation, and Composition
3. Software Engineering: Question about the differences between association, aggregation, and composition
4. UML 2 Class Diagrams: An Agile Introduction – Associations
5. Java for Dummies: What is the life cycle of an object in Java
6. IBM UML Basics: The Class Diagram