Class Relationships

Real-World Relationships:

• Parent-child relationships among members of a species

• Friends relationship among users on Facebook

• Students who part of the same team

• City-city relationship for a flight network

In the above cases, two objects of the same class have a relationship with each other

A program will often use multiple classes

E.g.: for handling data in IMDB, classes could be

• Movie, Actor, User, Review, Ratings, ...

• In Java, each class will be in its own .java file:

• Movie.java, Actor.java, User.java, Review.java, Ratings.java, etc

Relationships among objects from multiple classes

• Classes: Student, Course, Professor, Classroom

• Student <-> Course

• Professor <-> Course

• Course <-> Classroom

• Classes: Movie, Actor, User, Review

• Movie <-> Actor

• User <-> Review

• Movie <-> Review

Types of Relationships

• One-to-one: Patient <-> Patient Record

• One-to-many or many-to-one: Person (Mom) <-> Person (child)

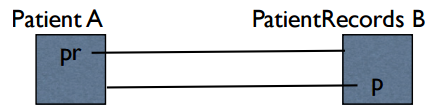
• A mom can have 1 or more children

• Many-to-many: Student <-> Course

• A student can take many courses

• A course can be taken by many students

One-to-one relationships • One-to-one relation among objects A and B • One way to represent it both ways: • A contains a reference to B • B contains a reference to A Patient A PatientRecords



Example

• Patient and PatientRecord

public class Patient {

private String name;

private String socialsecuritnumber;

private PatientRecords pr;

public Patient(String name, String s) {

this.name = name;

this.socialsecuritnumber = s;

pr = null;

}

public void setPatientRecords(PatientRecords r) {

this.pr = r;

}

}

public class PatientRecords {

private String doctor;

private Patient p;

public PatientRecords(String doctor) {

this.doctor = doctor;

p = null;

}

public void setPatient(Patient p) {

this.p = p;

}

}

<http://www.informit.com/guides/content.aspx?g=java&seqNum=560> Steven Haines

Defining classes and an inheritance hierarchy is interesting, but you need to define how those classes relate to one another if you are going to solve a real business problem. UML defines several different types of relationships, but they come in three main flavors:

* Aggregation
* Composition
* Association

Aggregation and composition define a “has-a” containment relationship. They are closely related, but the main difference comes in the notion of ownership. In a composition relationship we say that one object owns another object. For example, a car owns its engine, a library owns books, and a store owns a sign.

An aggregation relationship is similar to a composition relationship in that an object contains another object, but the object does not own the other object. Many times the difference will be evident by the life-cycle of its objects: if you destroy a car, its engine will be destroyed, if you close a library, its books will be sent away, and if a store closes, it's sign will be thrown in the trash. In this example, the engine, the books, and the sign only exist if the containing object exists. To illustrate how an aggregation may exist, consider that a driver sits in a car to drive it. The driver is contained by the car, but when it comes time to send the car to the junkyard, the driver will simply buy another car. This is an aggregation because the driver exists outside of the life-cycle of the car.

A composition is represented in UML as a line between the two objects with a filled-in diamond on the object that contains the other object. An aggregation is represented in UML as a line between the two objects with a hollow diamond on the object that contains the other object. Both relationships are depicted in [figure 1](http://www.informit.com/content/images/irf_guide_java_friesen4/elementLinks/java040110_uml-fig04.jpg).

[http://www.informit.com/content/images/irf_guide_java_friesen4/elementLinks/thjava040110_uml-fig04.jpg](javascript:popUp('/content/images/irf_guide_java_friesen4/elementLinks/java040110_uml-fig04.jpg'))

[**Figure 1.**](javascript:popUp('/content/images/irf_guide_java_friesen4/elementLinks/java040110_uml-fig04.jpg')) Composition vs Aggregation

In figure 1, the Car class contains the Engine class as a composition. Likewise, the Car class contains the Driver class as an aggregation.

Associations define non-containment relationships between objects. Associations define multiplicities in their relationships of form:

* One-to-One
* One-to-Many
* Many-to-One
* Many-to-Many

The multiplicities are specified using the notations:

* 1: there is one, and only one, instance at the end of the associations
* 0..1: there can be zero or one instance at the end of the association
* 0..\*: there can be zero or more instances at the end of the association
* 1..\*: there must be at least one, but there can be many instances at the end of the association

Most of the time, the multiplicities will logically flow from your understanding of the domain you are modeling.

<http://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/>

Association, Aggregation, Composition, Abstraction, Generalization, Realization, Dependency

These terms signify the relationships between classes. These are the building blocks of object oriented programming and very basic stuff. But still for some, these terms look like Latin and Greek. Just wanted to refresh these terms and explain in simpler terms.

Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

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

***Example:***A Student and a Faculty are having an association.

Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

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

Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

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

***Example:***A class contains students. A student cannot exist without a class. There exists composition between class and students.

**Difference between aggregation and composition**

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

***Example:***A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

## Abstraction

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

***Example:***A wire frame model of a car.

## Generalization

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

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

***Example:*** Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

## Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

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

***Example:*** A particular model of a car ‘GTB Fiorano’ that implements the blueprint of a car realizes the abstraction.

## Dependency

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

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

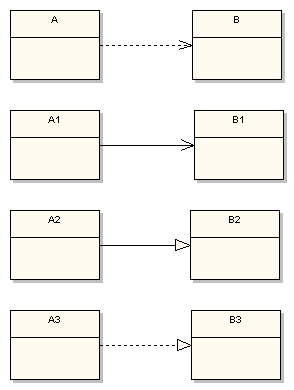
***Example:***Relationship between shape and circle is dependency.

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

**Understanding UML Class Relationships**

Vaughn Vernon, Principal Architect  
Copyright © 2004 Vaughn Vernon, ShiftMETHOD. All rights reserved.

Some time ago while interviewing dozens of prospective developers for a project, I discovered that around 90% of candidates claiming to know UML very well could not distinguish between some of the common UML relationship elements used in class diagrams:



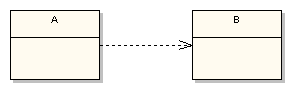
Can you name each of the above relationships using UML nomenclature? Can you describe what each of the relationships mean relative to your target programming language, such as Java?

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. You may say, “it doesn’t matter that much because I know what I mean and I know how to implement it.” True, in a project where the architect, the analyst, the designer, and the developer roles are all filled by one person (e.g. you), having a disconnect between diagram and implementation sort of works out — at least until someone else gets involved. Nonetheless, that’s a small-project mentality, not to mention the fact that it’s a near-sighted approach.

What about large organizations where huge systems and applications are developed? When different people or even different teams of people fill the above roles, all the people involved better understand what each of the UML relationships drawn on class diagrams represent. Otherwise, the misrepresentation and/or misinterpretation of the UML will result in incorrect implementations of code. With tight project timelines there’s no time for unnecessary re-implementation. This article is meant to help you and your team to interpret UML class relationships correctly the first time. So, let’s dig in!

**Dependency**

The UML *dependency* relationship is the least formal of them all. It means that the class at the source end of the relationship has some sort of dependency on the class at the target (arrowhead) end of the relationship. For example, the following simple states that class A depends on class B in some way:



While dependency may have broad meaning, it is best not to overuse the dependency relationship. In an analysis model class diagram such as a domain model diagram you may be tempted to convey that all the classes just depend on each other. Interestingly however, the *Rational Unified Process* (RUP) specifies that the general class relationship that should be used in the analysis model is *association* (covered next), and not dependency. Therefore, even when you are modeling higher-level concepts it is best not to use the dependency relationship loosely. It is just too nebulous.

Further, unless you use the dependency relationship in a constrained manner your model consumers (yourself or other developers) will simply have too broad an interpretation of its meaning. Generally those filling architect and designer roles in a project are there to give guidance to less experienced developers. Thus the dependency relationship should be used to convey a specific kind of guidance from architects and designers to developers.

So what should a dependency relationship represent? In our UML example above the 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. Again, use *association* to do that (covered next).

In Java, the following is the proper interpretation of the constrained dependency relationship:

import B; public class A { public void method1(B b) { // . . . } public void method2() { B tempB = new B(); // . . . } }

Actually either one of class B’s uses, as a parameter to a method, or as a local instance reference inside a method, would be appropriate reflection of a UML dependency relationship. Basically to Java the constrained dependency relationship means that you must import class B into class A so that class A may reference it in some way in a method. However, the following would be an incorrect implementation of the constrained dependency relationship in Java:

import B;

public class A {

private B b; // wrong!

public B getB() {

return b;

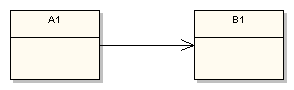
}

}

In the above example class B is used to define the state of an instance of class A by declaring an instance of class B with the instance scope. However, this is a misinterpretation of the dependency relationship. But that does lead us to the use of the next UML class relationship type, *association*.

**Association**

Now that the dependency relationship is understood-that is, we know what dependency means and what it does not mean-it is easier to understand the UML class relationship called *association*. Here’s an example of one class that has an association with another class:



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. This example manifests itself as the following Java code:

import B1;

public class A1 {

private B1 b1;

public B1 getB1() {

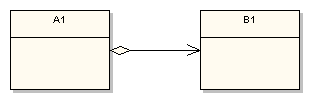
return b1;

}

}

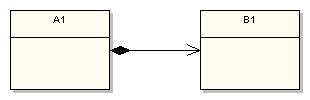
So, in a sense the association relationship specifies what the constrained dependency relationship does not. The association relationship does define the state of instances of the dependent class. The dependent class (A1) must, therefore, define an instance of the associated class (B1) within its class scope.

There’s more to the association relationship. Because we are now discussing state, it may be necessary to define the lifetime of the instances that make up the dependent object’s state, and how many of the associated class instances there are. These modeling techniques refer to aggregation/composition and multiplicity, respectively. For example, the following shows an aggregation association between two classes:



A clear diamond adornment has been added to the source side of the relationship. This means that A1 aggregates a 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.

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:



The numeric adornments next to the association arrow indicate the number of instances involved in the association. This example says that one instance of class A1 will always contain (state) references to many instances of class B1. There is a range of available multiplicity adornments that can be used, for example 0, 1, 0..1, 0..\*, 1..3, 1..\*, and so forth. Multiplicity may also be used when an association relationship shows aggregation or composition.

A guideline on the modeling of multiplicity is appropriate. In the above example the diagram states that one instance of A1 contains many instances of B1. This implies an array association. However, in your target programming language you may not want to implement this relationship as a literal array of B1 instances:

import B1;

public class A1 {

private B1[] b1;

// . . .

public B1 getB1(int anIndex) {

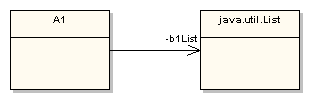
return b1[anIndex];

}

}

As you have probably experienced, dealing with an array in this way is complicated. You must reallocate the array each time a new item is added to it and you must check index ranges before accessing it. Clearly the method getB1(…) above is not well written, as an unhandled runtime exception is almost guaranteed at some point in time.

In Java you would likely decide to use some form of java.util.Collection, such as java.lang.List as implemented by java.lang.ArrayList rather than a literal array. An organizational standard should be authored that states how such associations should be modeled in UML. Certainly in an analysis model the above multiplicity association is appropriate. It clearly shows the intent of the relationship. But I suggest that in the design model such relationships will be clearer to developers who will be interpreting diagrams if you use the following standard:



Rather than using multiplicity in the design model, unless you absolutely intend for an array to be used, I believe the above to be a better use of UML. This association relationship states that one instance of A1 will declare an instance variable of class java.util.List and that the instance variable’s name will be b1List, which is expressed as a UML role name. Here’s a snippet of the corresponding code:

import java.util.ArrayList; import java.util.List;

import B1;

public class A1 {

private List b1List;

public A1() {

super();

b1List = new ArrayList();

}

// . . .

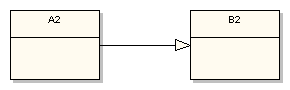
}

Many of the concerns we have with using Java arrays have disappeared because of the use of a pre-written and well-tested class. And besides, the UML perfectly communicates the designer’s intent to the programmer who will be interpreting the diagram.

There may be an exception to the above rule-of-thumb. If your UML tool allows you to specify an overriding collection class for various multiplicity types (e.g. ordered and qualified) you may determine that the use of multiplicity is justified. The code that the tool generates per your overriding collection class specifications will clearly indicate the architect’s or designer’s intent.

**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:



More specifically UML generalization corresponds to class extension in the Java language. The above diagram fragment would be implemented in Java as follows:

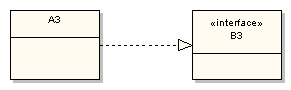
import B2;

public class A2 extends 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.

**Realization**

The final UML class relationship type I cover is 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. In the Java language the above realization relationship would be programmed as follows:

import B3;

public class A3 implements B3 {

// . . .

}

Realization is very important when designing object-oriented subsystems and frameworks. The interface being realized in a class diagram represents a contract between the subsystem or framework and its consumer. The interface publisher guarantees that any consumer implementing one or more of its public interfaces properly will have some level of consistent integration with the interface-defining subsystem or framework.

**Conclusion**

As I stated at the outset, UML can play a much greater role than drawing pictures. UML has the properties necessary to provide tremendous value in conveying how a software architecture and design should be implemented. If all technical project stakeholders understand the four UML relationship types used in class diagrams as discussed in this article, you can be certain that the quality of your UML modeling and of your implementations will increase.