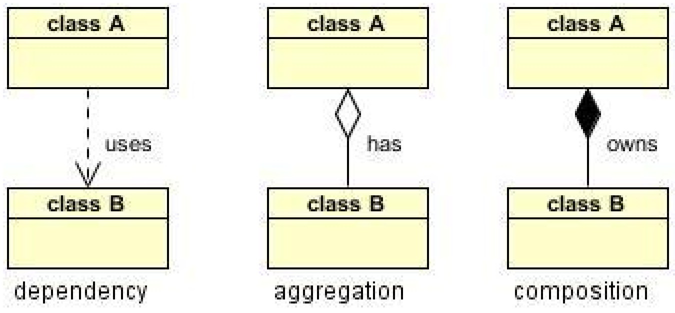
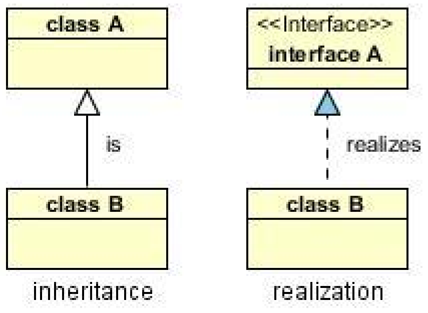
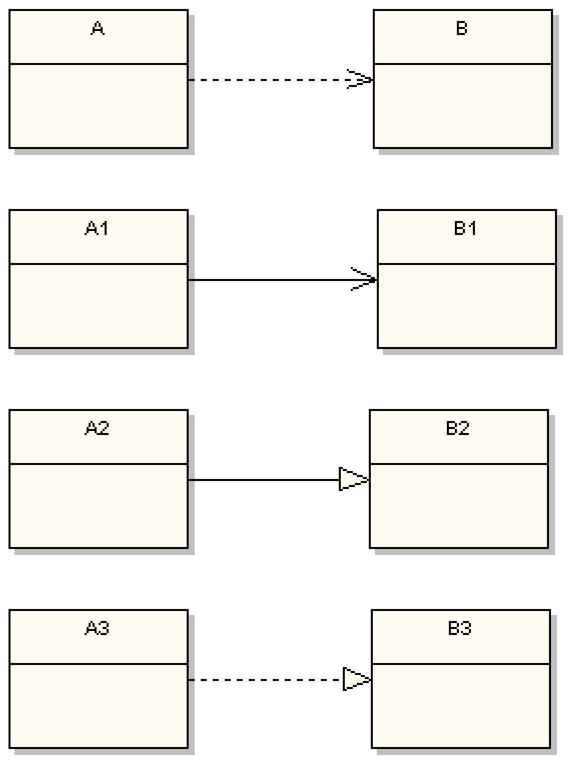
Understanding UML Class Relationships





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:

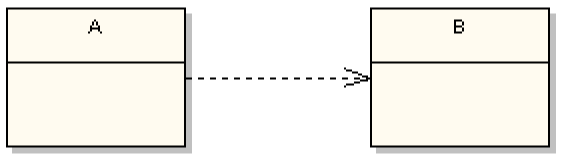


UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. You may say, “it doesn’t matter that much because I know what I mean and I know 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(); // . . . } }
```

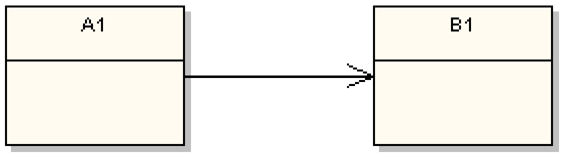
Actually either one of class B's uses, as a parameter to a method, or as a local instance reference inside a method, would be appropriate reflection of a UML **dependency** relationship. Basically to Java the constrained dependency relationship means that you must import class B into class A so that 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:

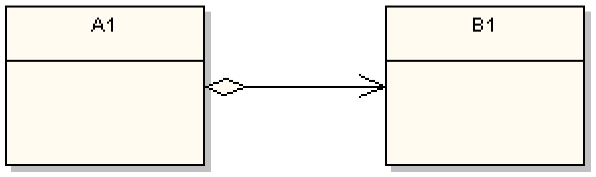


Association defines dependency, but a much stronger dependency than that described above with the plain *dependency* relationship. The arrowhead means that there is a one-way relationship. In this example it means that class A1 is associated with class B1. In other words, class A1 uses and contains one 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.

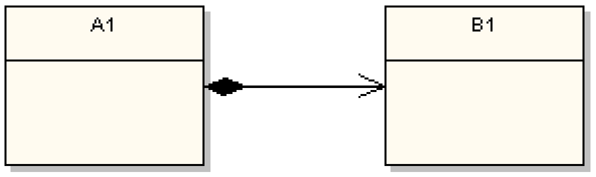
There's more to the association relationship. Because we are now discussing state, it may be necessary to define the **lifetime of the instances that make up the dependent object's state**, and how many of the associated class instances there are. These modeling techniques refer to aggregation/composition 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:



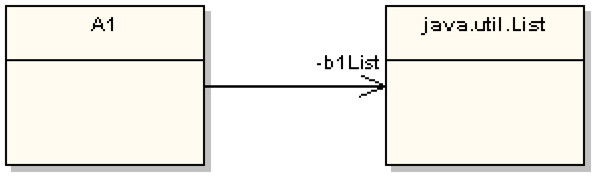
The numeric adornments next to the association arrow indicate the number of instances involved in the association. This example says that one instance of class A1 will always contain (state) references to many instances of class B1. There is a range of available multiplicity adornments that can be used, 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:

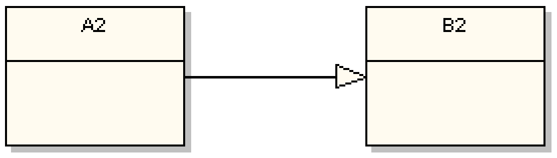
```
 import java.util.ArrayList;
import java.util.List;

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:



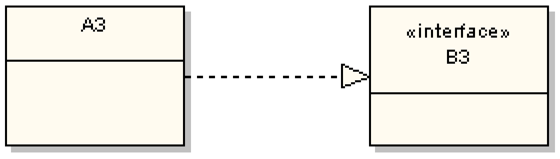
More specifically UML generalization corresponds to class extension in the Java language. The above diagram fragment would be implemented in Java as follows:

```
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.