

VAUGHN VERNON: SOFTWARE CRAFTSMAN

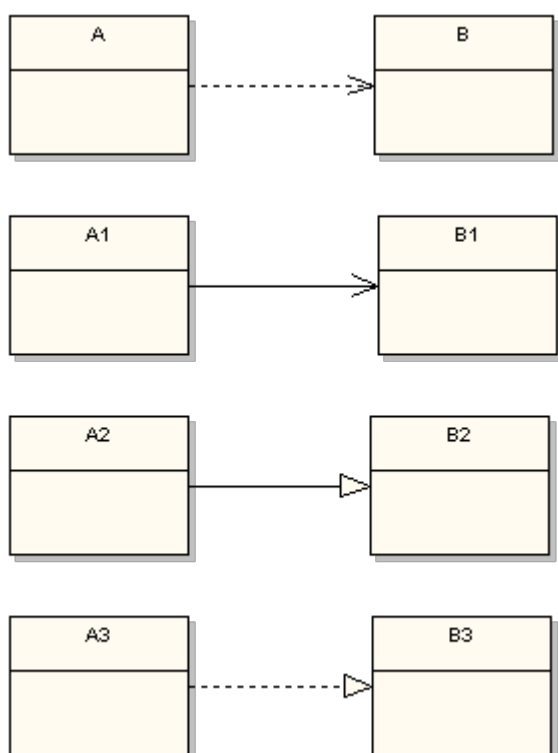
[BLOG](#) [BOOKS](#) [ONLINE TRAINING](#) [WORKSHOPS](#) [ABOUT](#)

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?

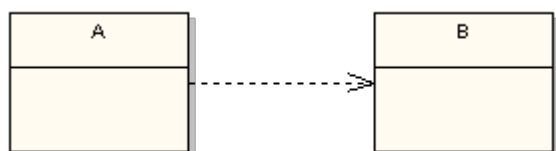
UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. You may say, “it doesn’t matter that much because I know what I mean and I know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

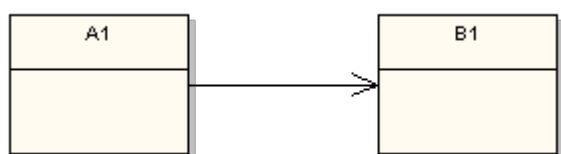
    private B b; // wrong!

    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

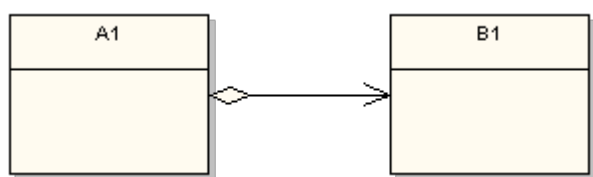
plain *dependency* relationship. The arrowhead means that there is a one-way relationship. In this example it means that class A1 is associated with class B1. In other words, class A1 uses and contains one instance of class B1, but B1 does not know about or contain any instances of class A1. This example 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.

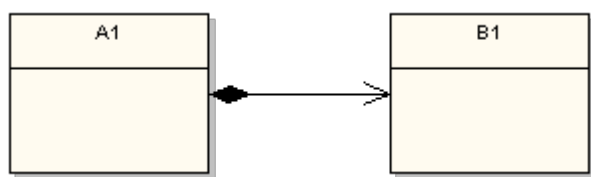
There's more to the association relationship. Because we are now discussing state, it may be necessary to define the lifetime of the instances that make up the dependent object's state, and how many of the associated class instances there are. These modeling techniques refer to aggregation/composition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

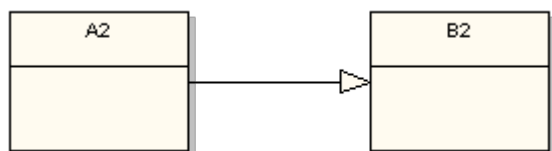
    // . . .
}
    
```

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:

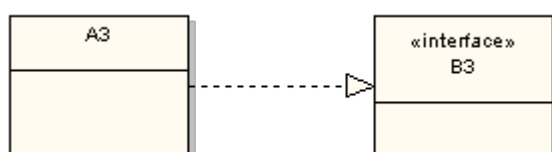
```
import B2;

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.

UML Association

Association is a **relationship** between **classifiers** which is used to show that instances of classifiers could be either **linked** to each other or **combined** logically or physically into some aggregation.

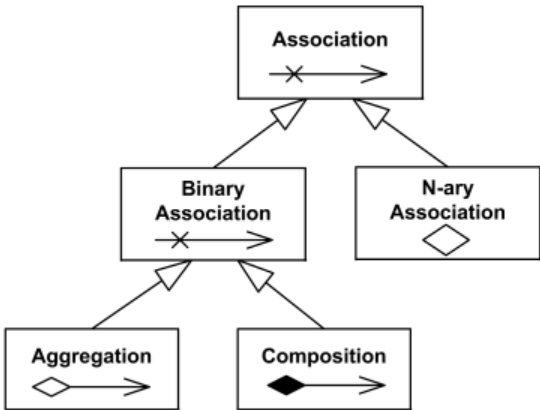
UML specification categorizes association as **semantic relationship**. Some other UML sources also categorize association as a **structural relationship**. Wikipedia states that association is **instance level** relationship and that associations can only be shown on **class diagrams**. Not sure where they got that information from but it is not based on UML specification. **Association** could be used on different types of UML **structure diagrams**:

- **class diagram associations**,
- **use case diagram associations**,
- **deployment diagram artifact associations**,
- **deployment diagram communication path**.

There are several concepts related to association:

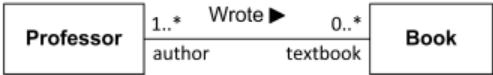
- **association end ownership**,
- **navigability**,
- **association arity**,
- **aggregation type**.

UML 2.4 specification states that for the association: "*Aggregation type, navigability, and end ownership are orthogonal concepts, ...*" which is clearly an overstatement. Orthogonal usually means completely independent. While notation for **aggregation type**, **navigability**, and **association end ownership** could be applied independently, the concepts themselves are not orthogonal. For example, in UML 2.4 end property of association owned by an end class is navigable, which clearly makes navigability dependent on ownership.



Association relationship overview diagram

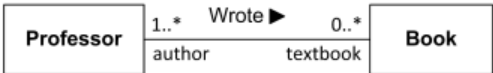
An association is usually drawn as a solid line connecting two classifiers or a single classifier to itself. Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the **association end**.



*Association **Wrote** between **Professor** and **Book** with association ends **author** and **textbook**.*

Association End

Association end is a connection between the line depicting an **association** and the icon depicting the connected **classifier**. Name of the association end may be placed near the end of the line. The association end name is commonly referred to as **role** name (but it is not defined as such in the UML 2.4 standard). The role name is optional and suppressible.



*Professor "playing the role" of **author** is associated with **textbook** end typed as **Book**.*

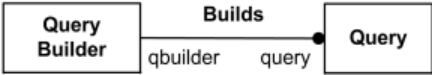
The idea of the **role** is that the same classifier can play the same or different roles in other associations. For example, Professor could be an author of some Books or an editor.

Association end could be **owned** either by

- **end classifier**, or
- **association** itself

Association ends of associations with **more than two ends** must be **owned by the association**.

Ownership of association ends by an associated **classifier** may be indicated graphically by a **small filled circle** (aka **dot**). The dot is drawn at the point where line meets the classifier. It could be interpreted as showing that the model includes a property of the type represented by the classifier touched by the dot. This property is owned by the classifier **at the other end**.



*Association end **query** is owned by classifier **QueryBuilder** and association end **qbuilder** is owned by association **Builds** itself*

The "ownership" dot may be used in combination with the other graphic line-path notations for properties of associations and association ends. These include aggregation type and navigability.

UML standard does not mandate the use of explicit end-ownership notation, but defines a notation which shall apply in models where such use is elected. The dot notation must be applied at the level of complete associations or higher, so that the **absence of the dot signifies ownership by the association**. In other words, in binary associations the dot will be omitted only for the ends which are **not owned by a classifier**.

Attribute notation can be used for an association end **owned by a class**, because an association end owned by a class is also an **attribute**. This notation may be used in conjunction with the line arrow notation to make it perfectly clear that the attribute is also an **association end**.



*Association end **qb** is an **attribute** of SearchService class and is **owned** by the class.*

Navigability

End property of association is **navigable** from the opposite end(s) of association if instances of the classifier at this end of the **link** can be accessed efficiently at runtime from instances at the other ends of the link.

UML specification does not dictate how efficient this access should be or any specific mechanism to achieve the efficiency. It is implementation specific.

When end property of association is marked as **not navigable**, in [UML 2.4] it means that *"access from the other ends may or may not be possible, and if it is, it might not be efficient."* The problem with this definition of **not navigable** is that it actually means "whatever" or "who cares?" navigability.

UML 2.4 also provides another definition of **navigability**:

*An end property of association that is owned by an end class, or that is a navigable owned end of the association indicates that the association is **navigable** from the opposite ends; otherwise, the association is **not navigable** from the opposite ends.*

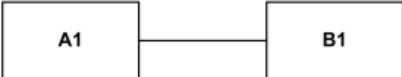
This definition is odd because it makes **navigability** strongly dependent on **ownership**, while these are assumed to be **orthogonal** concepts; some examples in UML 2.4 specs show end properties owned by a class as **not navigable**, which contradicts to the definition above; and navigability is defined using **"navigable owned end of the association"**.

Deprecated navigability convention:

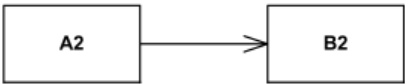
- non-navigable ends were assumed to be owned by the association
- navigable ends were assumed to be owned by the classifier at the opposite end.

Notation:

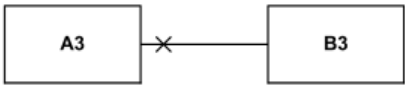
- **navigable** end is indicated by an **open arrowhead** on the end of an association
- **not navigable** end is indicated with a **small x** on the end of an association
- no adornment on the end of an association means **unspecified navigability**



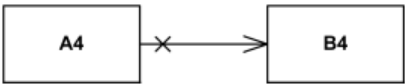
Both ends of association have **unspecified navigability**.



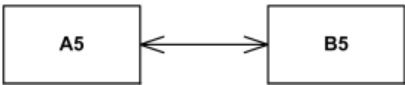
A2 has **unspecified navigability** while B2 is **navigable** from A2.



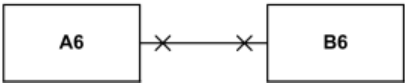
A3 is **not navigable** from B3 while B3 has **unspecified navigability**.



A4 is **not navigable** from B4 while B4 is **navigable** from A4.



A5 is **navigable** from B5 and B5 is **navigable** from A5.



A6 is **not navigable** from B6 and B6 is **not navigable** from A6.

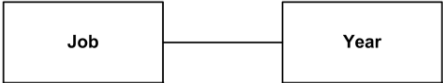
A **visibility** symbol can be added as an adornment on a navigable end to show the end's visibility as an attribute of the featuring classifier.

Arity

Each association has specific **arity** as it could relate two or more items.

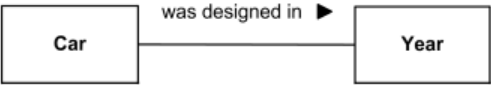
Binary Association

Binary association relates **two** typed instances. It is normally rendered as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). The line may consist of one or more connected segments.



Job and Year classifiers are associated

A small solid triangle could be placed next to or in place of the name of **binary association** (drawn as a solid line) to show the **order of the ends** of the association. The arrow points along the line in the direction of **the last end** in the order of the association ends. This notation also indicates that the association is to be **read** from the first end to the last end.

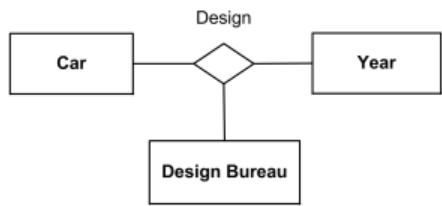


Order of the ends and reading: Car - was designed in - Year

UML 2.4 specification states that this arrow is used **for documentation purposes only** and **has no general semantic interpretation**. This is an odd clarification as UML diagrams are in fact used mostly for documentation purposes but even more important, this arrow according to the UML spec defines the **order** of association ends - which does belong to semantics.

N-ary Association

Any association may be drawn as a **diamond** (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. **N-ary association** with more than two ends can **only** be drawn this way.



*Ternary association **Design** relates three classifiers*

Shared and Composite Aggregation

Aggregation is a **binary association** representing some **whole/part relationship**. Aggregation type could be either:

- **shared aggregation** (aka **aggregation**), or
- **composite aggregation** (aka **composition**).

Association Class

An association may be refined to have its **own set of features**; that is, features that do not belong to any of the connected classifiers but rather to the association itself. Such an association is called an **association class**. It is both an association, connecting a set of classifiers and a class, and as such could have features and might be included in other associations.

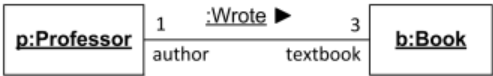
An association class can be seen as an association that also has class properties, or as a class that also has association properties.

An association class is shown as a class symbol attached to the association path by a dashed line. The association path and the association class symbol represent the same underlying model element, which has a single name. The association name may be placed on the path, in the class symbol, or on both, but they must be the same name.

Link

Link is an instance of an **association**. It is a tuple with one value for the each end of the association, where each value is an instance of the type of the end. Association has at least two ends, represented by properties (**end properties**).

Link is rendered using the same notation as for an association. Solid line connects **instances** rather than classifiers. Name of the link could be shown underlined though it is not required. End names (**roles**) and navigation arrows can be shown.



*Link **Wrote** between instance **p** of **Professor** playing **author** role and instance **b** of **Book** in the **textbook** role.*



UML Aggregation

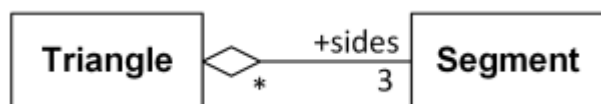
Shared aggregation (aggregation) is a **binary association** between a **property** and one or more composite objects which group together a set of instances. It is a "weak" form of **aggregation** when part instance is independent of the composite. Shared aggregation has the following characteristics:

- it is **binary association**,
- it is asymmetric - only one end of association can be an aggregation,
- it is transitive - aggregation links should form a directed, acyclic graph, so that no composite instance could be indirect part of itself,
- shared part could be included in several composites, and if some or all of the composites are deleted, shared part may still exist.

Notation

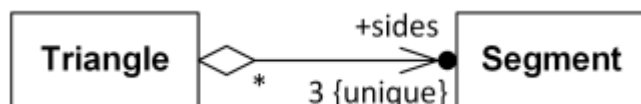
Shared aggregation is depicted as association decorated with a *hollow diamond* at the aggregate end of the association line. The diamond should be noticeably smaller than the diamond notation for N-ary associations.

Example below shows *Triangle* as an aggregate of exactly three line segments (sides). Multiplicity '*' of the *Triangle* association end means that each line *Segment* could be a part of several triangles, or might not belong to any triangle at all. Erasing specific *Triangle* instance does not mean that all or any segments will be deleted as well. (Note, that we named collection of three line Segments as 'sides', while usual UML convention is to use singular form, i.e. 'side', even for collections.)



Triangle has 'sides' collection of three line Segments.
Each line Segment could be part of none, one, or several triangles.

Shared aggregation could be depicted together with other association adornments such as **navigability** and **association end ownership**. In the example below line *Segment* is navigable from *Triangle*. Association end 'sides' is owned by *Triangle* (not by association itself), which means that 'sides' is an **attribute** of *Triangle*.



Triangle has 'sides' collection of three unique line Segments.
Line segments are navigable from Triangle.
Association end 'sides' is owned by Triangle, not by association itself.

Mistakes

Aggregation is asymmetric relationship - only **one end** of association is allowed to be marked as shared or composite aggregation. Both UML 1.x and 2.x don't allow a diamond to be attached to both ends of association line. The reasoning behind the example below was that each *Student* instance has a list of courses he/she is registered to, and every *Course* has a list of students registered for that course.



Aggregation mistake - only one end of association can be marked as aggregation.

It will not help if we draw two separate aggregations as shown below. Aggregation links should form a directed, **acyclic** graph, so that no composite instance should be direct or indirect part of itself.



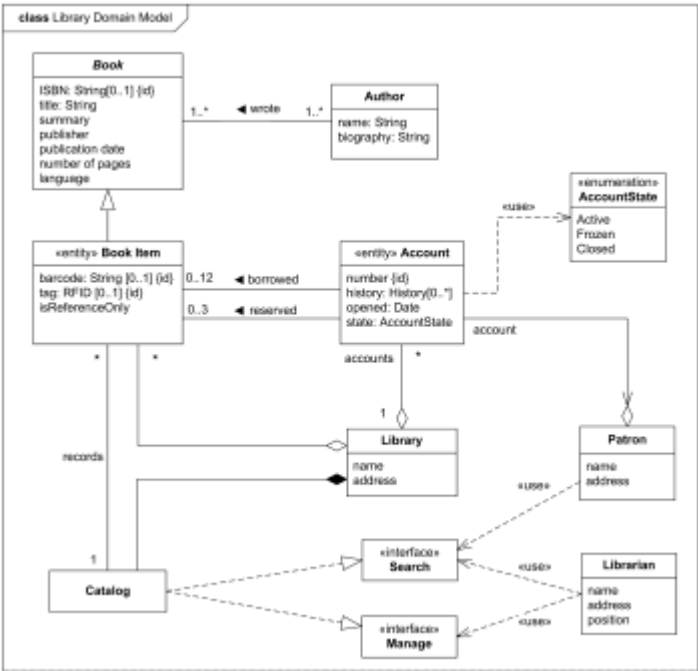
Aggregation mistake - no composite instance should be direct or indirect part of itself.

History

In UML 1.x aggregation kinds were *none*, *aggregate*, and *composite*. UML 2.0 renamed the aggregation kind described on this page from *aggregate* to *shared*, so that in UML 2.x aggregation kinds are: *none*, *shared*, and *composite*.

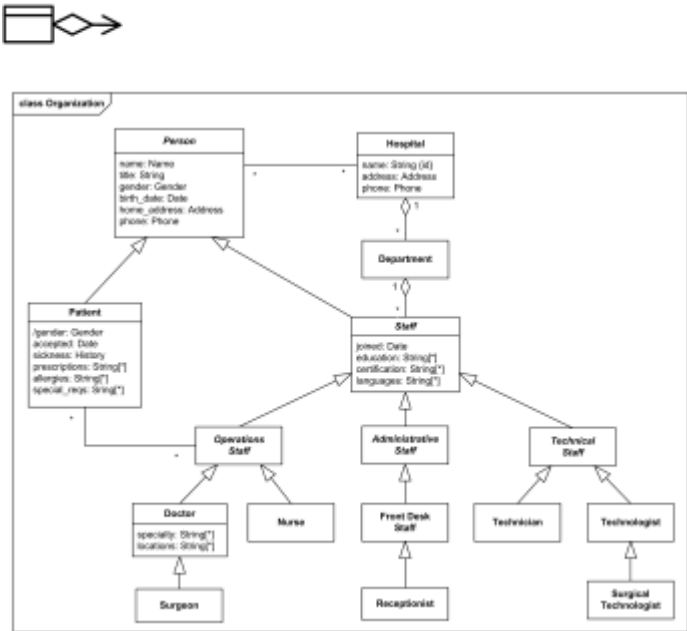
Examples

Library domain model

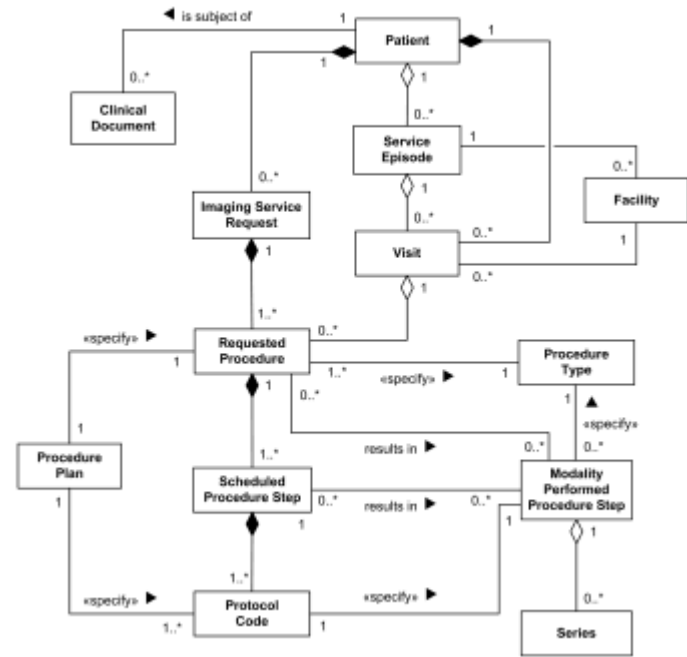


Hospital domain UML class diagram example


UML shared aggregation is relationship between a property and one or more composite objects used to group together a set of instances.



Digital imaging in medicine - DICOM model of the real world



Android Camera implementation classes

 Composition



UML Composition

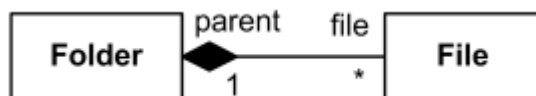
Composite aggregation (composition) is a "strong" form of **aggregation** with the following characteristics:

- it is **binary association**,
- it is a *whole/part* relationship,
- a part could be included in *at most one* composite (whole) at a time, and
- if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

Note, that UML does not define how, when and specific order in which parts of the composite are created. Also, in some cases a part can be removed from a composite before the composite is deleted, and so is not necessarily deleted as part of the composite.

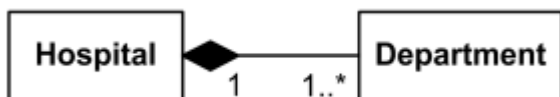
Notation

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



*Folder could contain many files, while each File has exactly one Folder parent.
If Folder is deleted, all contained Files are deleted as well.*

When composition is used in **domain models**, both whole/part relationship as well as event of composite "deletion" should be interpreted figuratively, not necessarily as physical containment and/or termination. UML specification needs to be updated to explicitly allow this interpretation.



*Hospital has 1 or more Departments, and
each Department belongs to exactly one Hospital.
If Hospital is closed, so are all of its Departments.*

Note, that though it seems odd, multiplicity of the composite (whole) could be specified as **0..1** ("at most one") which means that part is allowed to be a "stand alone", not owned by any specific composite.



*Each Department has some Staff, and each Staff could be
a member of one Department (or none). If Department is closed,
its Staff is relieved (but excluding the "stand alone" Staff).*

Mistakes

Composition is asymmetric relationship - only **one end** of association is allowed to be marked as shared or composite aggregation. Both UML 1.x and 2.x don't allow a diamond to be attached to both ends of association line.

