# chapter 9 Relationships

## 9.1 Chapter roadmap

This chapter discusses relationships between objects, and relationships between classes. To find out what a relationship is, read Section 9.2. The chapter is then organized under three separate threads. We discuss links (relationships between objects) in Section 9.3, associations (relationships between classes) in Section 9.4, and, finally, dependencies (catch-all relationships) in Section 9.5.

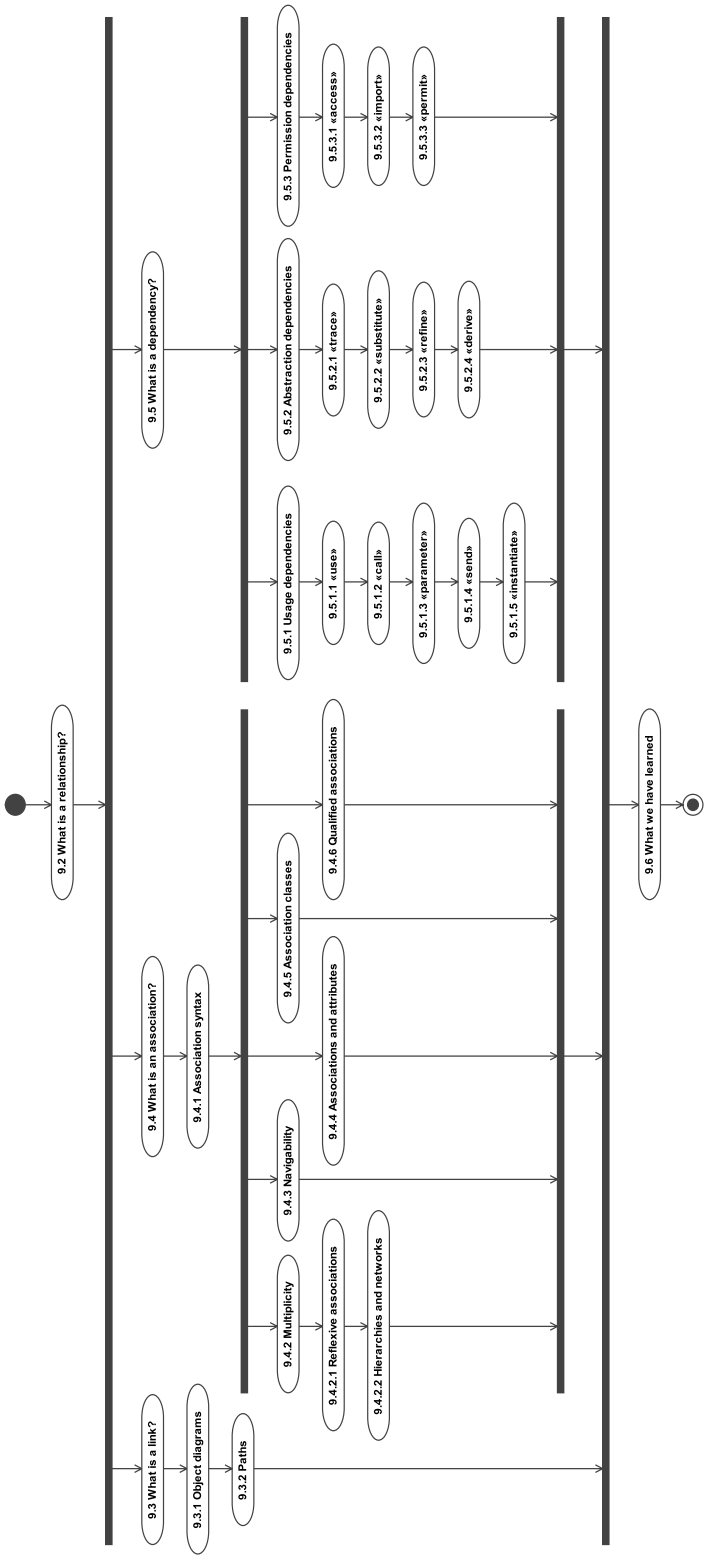

9.2 What is a relationship?
9.3 What is a link?
9.3.1 Object diagrams
9.3.2 Paths
9.4 What is an association?
9.4.1 Association syntax
9.4.2 Multiplicity
9.4.2.1 Reflexive associations
9.4.2.2 Hierarchies and networks
9.4.3 Navigability
9.4.4 Associations and attributes
9.4.5 Association classes
9.4.6 Qualified associations
9.5 What is a dependency?
9.5.1 Usage dependencies
9.5.1.1 «use»
9.5.1.2 «call»
9.5.1.3 «parameter»
9.5.1.4 «send»
9.5.1.5 «instantiate»
9.5.2 Abstraction dependencies
9.5.2.1 «trace»
9.5.2.2 «substitute»
9.5.2.3 «refine»
9.5.2.4 «derive»
9.5.3 Permission dependencies
9.5.3.1 «access»
9.5.3.2 «import»
9.5.3.3 «permit»
9.6 What we have learned

Figure 9.1

## 9.2 What is a relationship?

Relationships are semantic (meaningful) connections between modeling elements—they are the UML way of connecting things together. You have already seen a few types of relationships:

- between actors and use cases (association);
- between use cases and use cases (generalization, «include», «extend»);
- between actors and actors (generalization).

UML relationships connect things.

In this chapter we explore connections between objects and connections between classes. We start with links and associations, and then, in Chapter 10, look at generalization and inheritance.

To create a functioning OO system, you can't let the objects stand alone in glorious isolation. You need to connect them so that they can perform useful work of benefit to the users of the system. Connections between objects are called links, and when objects work together, we say that they collaborate.

If there is a link between two objects, there must also be some semantic connection between their classes. This is really common sense—for objects to communicate directly with each other, the classes of those objects must know about each other in some way. Connections between classes are known as associations. Links between objects are actually instances of the associations between their classes.

## 9.3 What is a link?

To create an object-oriented program, objects need to communicate with each other. In fact, an executing OO program is a harmonious community of cooperating objects.

Objects send messages to one another over connections called links.

A link is a semantic connection between two objects that allows messages to be sent from one object to the other. An executing OO system contains many objects that come and go, and many links (that also come and go) that join those objects. Messages are passed back and forth between objects over these links. On receipt of a message, an object will invoke its corresponding operation.

Links are implemented in different ways by different OO languages. Java implements links as object references; C++ may implement links as pointers, as references, or by direct inclusion of one object by another.

Whatever the approach, a minimal requirement for a link is that *at least one* of the objects must have an object reference to the other.

### 9.3.1 Object diagrams

An object diagram is a diagram that shows objects and their relationships at a point in time. It is like a snapshot of part of an executing OO system at a particular instant, showing the objects and the links between them.

Objects that are connected by links may adopt various roles relative to each other. In Figure 9.2, you can see that the ila object adopts the role of chairperson in its link with the bookClub object. You indicate this on the object diagram by placing the role name at the appropriate end of the link. You can put role names at either or both ends of a link. In this case, the bookClub object always plays the role of "club" and so there is no real point in showing this on the diagram—it would not really add anything to our understanding of the object relationships.

Figure 9.2 tells us that at a particular point in time, the object ila is playing the role of chairperson. However, it is important to realize that links are *dynamic* connections between objects. In other words, they are not necessarily fixed over time. In this example, the chairperson role may pass at some point to erica or naomi, and we could easily create an object diagram to show this new state of affairs.

Normally, a single link connects exactly two objects as shown in Figure 9.2. However, UML does allow a single link to connect more than two objects. This is known as an n-ary link and is shown as a diamond with a path to each participating object. Many modelers (ourselves included) consider this idiom to be unnecessary. It is rarely used and UML modeling tools do not always support it, so we do not say anything more about it here.

Object diagrams are snapshots of an executing OO system.

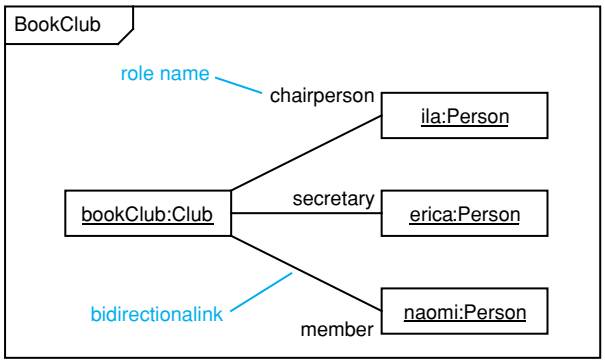



**Figure 9.2**

Considering Figure 9.2 in more depth, you can see that there are three links between four objects:

- a link between bookClub and ila;
- a link between bookClub and erica;
- a link between bookClub and naomi.

Use navigability to specify which directions messages may pass over a link.

In Figure 9.2 the links are bidirectional, so you can just as correctly say that the link connects ila to bookClub or that the link connects bookClub to ila.

If a link is unidirectional, you use navigability to specify in which direction messages may pass over the link.

You can show navigability by placing an arrowhead (navigable) or cross (not navigable) on the end of a link. Think of navigability as being a bit like a one-way system in a city. Messages can only flow in the direction indicated by the arrowhead.

The UML 2 specification allows three different modeling idioms for showing navigability, which we discuss in detail in Section 9.4.3. We use the most common idiom consistently throughout this book:

- all crosses are suppressed;
- bidirectional associations have *no* arrows;
- unidirectional associations have a single arrow.

The only real disadvantage of this idiom is that there is no way to indicate that navigability is undecided, because no navigability is taken to mean "not navigable".

For example, Figure 9.3 shows that the link between :PersonDetails and :Address is unidirectional. This means that the :PersonDetails object has an object reference to the :Address object, but *not* vice versa. Messages can *only* be sent from :PersonDetails to :Address.

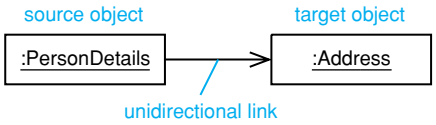



**Figure 9.3**

## 9.3.2 Paths

UML symbols, such as the object icon, use case icon, and class icon, are connected to other symbols by paths. A path is a "connected series of graphic

segments" (in other words, a line!) joining two or more symbols. There are three styles for drawing paths:

- orthogonal – where the path consists of a series of horizontal and vertical segments;
- oblique – where the path is a series of one or more sloping lines;
- curved – where the path is a curve.

It is a matter of personal preference as to which style of path is used, and the styles may even be mixed on the same diagram if this makes the diagram clearer and easier to read. We usually use the orthogonal style, as do many other modelers.

In Figure 9.4, we have adopted the orthogonal path style, and the paths have been combined into a tree. You can only combine paths that have the same properties. In this case, all the paths represent links and so we can legally combine them.

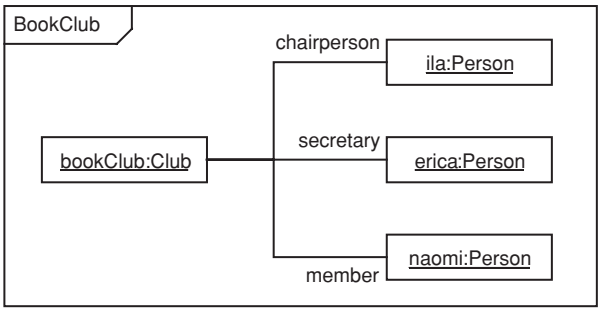



**Figure 9.4**

The visual neatness, readability, and general appeal of the diagrams is of crucial importance. Always remember that the majority of diagrams are drawn to be read by someone else. As such, no matter what style you adopt, neatness and clarity are vital.

## 9.4 What is an association?

Associations are connections between classes.

Associations are relationships between classes. Just as links connect objects, associations connect classes. The key point is that for there to be a link between two objects, there *must* be an association between the classes of those objects. This is because a link is an instance of an association, just as an object is an instance of a class.

Figure 9.5 shows the relationship between classes and objects, and between links and associations. Because you can't have a link without an association, it is clear that links *depend* on associations; you can model this with a dependency relationship (the dashed arrow) that we look at in more detail in Section 9.5. To make the semantics of the dependency between associations and links explicit, you stereotype the dependency «instantiate».

Objects are instances of classes, and links are instances of associations.

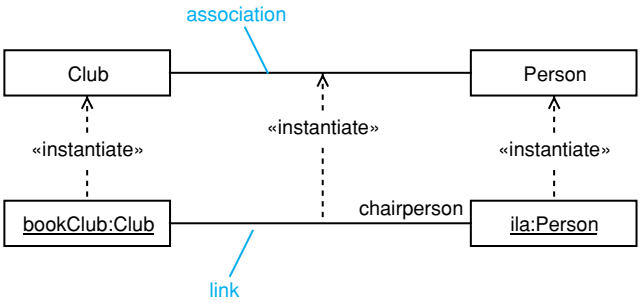



**Figure 9.5**

The semantics of the basic, unrefined association are very simple—an association between classes indicates that you can have links between objects of those classes. There are other more refined forms of association (aggregation and composition) that we look at in Section 18.3 in the design workflow.

### 9.4.1 Association syntax

Associations may have

- an association name;
- role names;
- multiplicity;
- navigability.

Association names should be verb phrases because they indicate an action that the source object is performing on the target object. The name may also be prefixed or postfixed with a small black arrowhead to indicate the direction in which the association name should be read. Association names are in lowerCamelCase.

In the example in Figure 9.6 you read the association as follows: "a `Company` employs many `Persons`." Although the arrow indicates the direction in which the association should be read, you can always read associations in the other direction as well. So in Figure 9.6 you can say, "each `Person` is employed by exactly one `Company`" at any point in time.

Association names are verb phrases that indicate the semantics of the association.

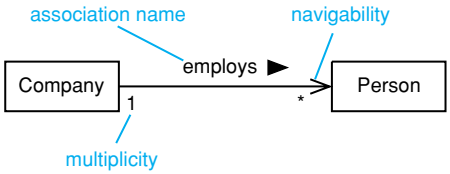



**Figure 9.6**

Alternatively, you can give role names to the classes on one or both ends of the association. These role names indicate the roles that objects of those classes play when they are linked by instances of this association. In Figure 9.7, you can see that a Company object will play the role employer, and Person objects will play the role employee when they are linked by instances of this association. Role names should be nouns or noun phrases as they name a role that objects can play.

Associations can have *either* an association name, *or* role names. Putting both role names *and* association names on the same association is theoretically legal, but this is very bad style—and overkill!

The key to good association names and role names is that they should read well. In Figure 9.6 a Company *employs* many Persons—this reads very well indeed. Reading the association the other way around, you can say that a Person *is employed* by exactly one Company at any point in time—it still reads very well. Similarly, the role names in Figure 9.7 clearly indicate the roles that objects of these classes will play when linked in this particular way.

Role names are noun phrases that indicate the roles played by objects linked by instances of the association.

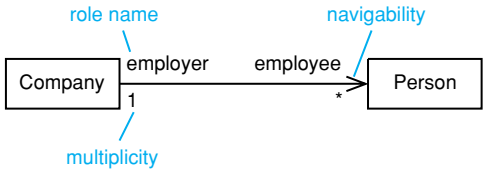



**Figure 9.7**

## 9.4.2 Multiplicity

Constraints are one of the three UML extensibility mechanisms, and multiplicity is the first type of constraint that we have seen. It is also by far the most common type of constraint. Multiplicity constrains the number of objects of a class that can be involved in a particular relationship *at any point*

*in time*. The phrase "at any point in time" is vital to understanding multiplicities. Considering Figure 9.8, you can see that at any point in time a Person object is employed by exactly one Company object. However, *over time* a Person object might be employed by a series of Company objects.

Looking at Figure 9.8, you can see something else that is interesting. A Person object can never be unemployed—it is always employed by exactly one Company object. The constraint therefore embodies two business rules of this model:

- that Person objects can only be employed by one Company at a time;
- that Person objects must *always* be employed.

Whether or not these are reasonable constraints depends entirely on the requirements of the system you are modeling, but this is what the model actually says.

You can see that multiplicity constraints are very important—they can encode key business rules in your model. However, these rules are "buried" in the details of the model. Literate modelers call this hiding of key business rules and requirements "trivialization". For a much more detailed discussion of this phenomenon, see [Arlow 1].

Multiplicity is specified as a comma-separated list of intervals, where each interval is of the form:

```
minimum..maximum
```

minimum and maximum may be integers or any expression that yields an integer result.

If multiplicity is not explicitly stated, then it is undecided.

If multiplicity is not explicitly stated, then it is undecided—there is no "default" multiplicity in UML. In fact, it is a common UML modeling error to assume that an undecided multiplicity defaults to a multiplicity of 1. Some examples of multiplicity syntax are given in Table 9.1.

Multiplicity specifies the number of objects that can participate in a relationship at any point in time.

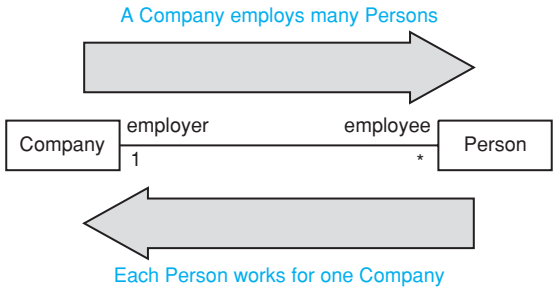



**Figure 9.8**

**Table 9.1**

| Adornment | Semantics |
|---|---|
| 0..1 | Zero or 1 |
| 1 | Exactly 1 |
| 0..* | Zero or more |
| * | Zero or more |
| 1..* | 1 or more |
| 1..6 | 1 to 6 |
| 1..3, 7..10, 15, 19..* | 1 to 3 *or* 7 to 10 *or* 15 exactly *or* 19 to many |

Always read the model exactly as written.

The example in Figure 9.9 illustrates that multiplicity is actually a powerful constraint that often has a big effect on the business semantics of the model.

If you read the example carefully, you see that

- a Company can have exactly seven employees;
- a Person can be employed by exactly one Company (i.e., in this model a Person can't have more than one job at a time);
- a BankAccount can have exactly one owner;
- a BankAccount can have one or many operators;
- a Person may have zero to many BankAccounts;
- a Person may operate zero to many BankAccounts.

When reading a UML model, it is vital to figure out exactly what the model actually says, rather than making any assumptions or hallucinating semantics. We call this "reading the model as written".

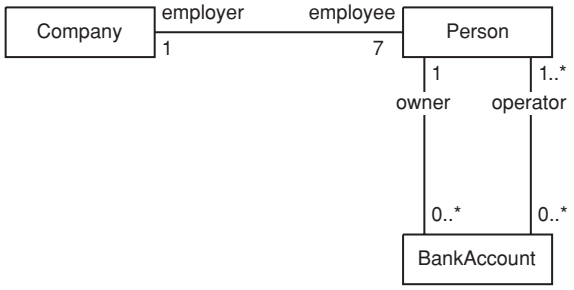



**Figure 9.9**

For example, Figure 9.9 states that a Company may have exactly seven employees, no more and no less. Most people would consider these semantics to be rather odd, or even incorrect (unless it is a very strange company), but this is what the model actually says. You must never lose sight of this.

There is a certain amount of debate as to whether multiplicity should be shown on analysis models. We think that it should, because multiplicity describes business rules, requirements, and constraints and can expose unwarranted assumptions made about the business. Clearly, such assumptions need to be exposed and challenged as early as possible.

### 9.4.2.1 *Reflexive associations*

When a class has an association to itself, this is a reflexive association.

It is quite common for a class to have an association to itself. This is called a reflexive association and it means that objects of that class have links to other objects of the same class. A good example of a reflexive association is shown in Figure 9.10. Each Directory object can have links to zero or more Directory objects that play the role subdirectory, and to zero or one Directory object that plays the role parent. In addition, each Directory object is associated with zero or more File objects. This models a generic directory structure quite well, although it's worth mentioning that specific file systems (such as Windows) may have different multiplicity constraints to this model.

The top half of Figure 9.10 shows the class diagram, and the bottom half shows an example object diagram that accords with that class diagram.

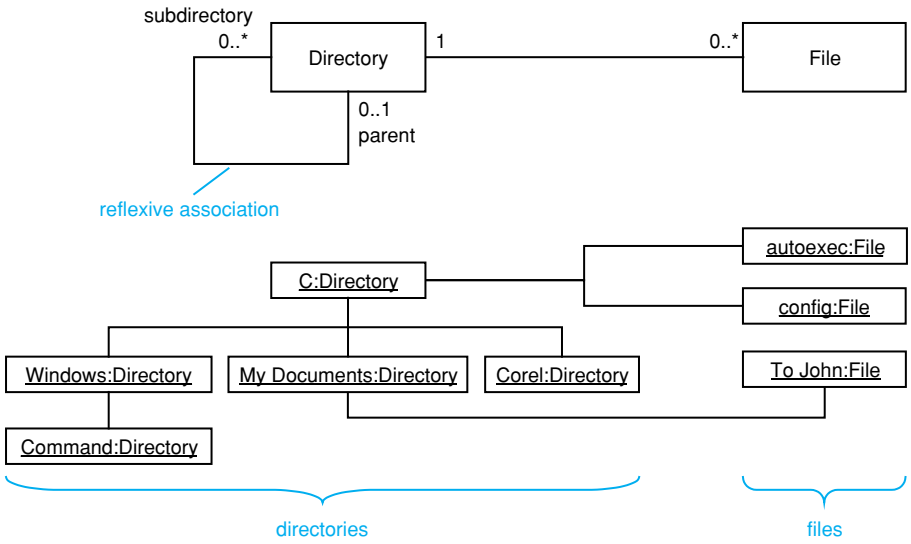



**Figure 9.10**

### 9.4.2.2 *Hierarchies and networks*

When modeling, you'll find that objects often organize themselves into hierarchies or networks. A hierarchy has one root object, and every other node in the hierarchy has exactly one object directly above it. Directory trees naturally form hierarchies. So do part breakdowns in engineering, and elements in XML and HTML documents. The hierarchy is a very ordered, structured, and somewhat rigid way of organizing objects. An example is shown in Figure 9.11.

In a hierarchy an object may have zero or one object above it.

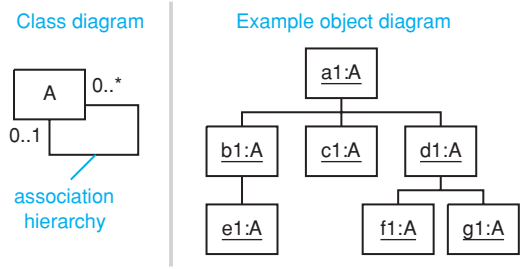



**Figure 9.11**

In the network, however, there is often no root object, although that is not precluded. In networks, each object may have many objects directly connected to it. There is no real concept of "above" or "below" in a network. It is a much more flexible structure in which it is possible that no node has primacy over another. The World Wide Web forms a complex network of nodes, as illustrated in a simple way in Figure 9.12.

In a network an object may have zero or many objects above it.

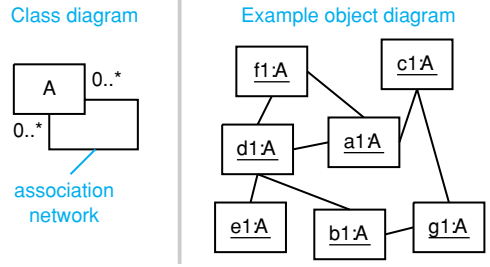



**Figure 9.12**

As an example to illustrate hierarchies and networks, let's consider products. There are two fundamental abstractions:

- ProductType – a type of product, such as "Inkjet Printer";
- ProductItem – a specific ink jet printer, serial number 0001123430.

ProductType and ProductItem are discussed in great detail in [Arlow 1]. ProductTypes often tend to form networks, so a ProductType such as a computer package may consist of a CPU, screen, keyboard, mouse, graphics card, and other ProductTypes. Each of these ProductTypes describes a *type* of product, not an individual item, and these types of products may participate in other composite ProductTypes, such as different computer packages.

However, if we consider the ProductItems, which are specific instances of a ProductType, any ProductItem, such as a specific CPU, can only be sold and delivered *once* as part of one package of goods. ProductItems, therefore, form hierarchies.

## 9.4.3 Navigability

Navigability indicates that objects of the source class "know about" objects of the target class.

Navigability shows us that it is possible to traverse from an object of the source class to one or more objects of the target class, depending on the multiplicity. You can think of navigability as meaning "messages can only be sent in the direction of the arrow". In Figure 9.13, Order objects can send messages to Product objects, but not vice versa.

One of the goals of good OO analysis and design is to minimize coupling between classes, and using navigability is a good way to do this. By making the association between Order and Product unidirectional, you can navigate easily from Order objects to Product objects, but there is no navigability back from Product objects to Order objects. So Product objects *do not know* that they may be participating in a particular Order and therefore have no coupling to Order.

Navigability is shown by appending either a cross or an arrowhead to an end of the relationship as shown in Figure 9.13.

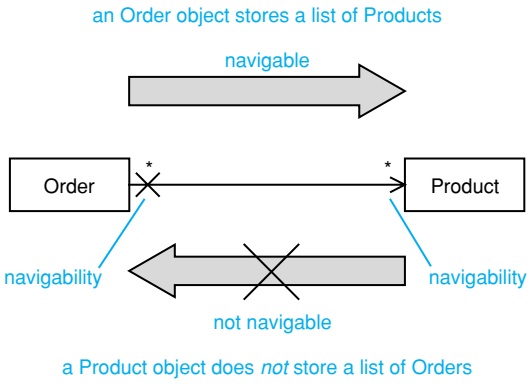



**Figure 9.13**

The UML 2.0 specification [UML2S] suggests three modeling idioms for using navigability on your diagrams.

1. Make navigability completely explicit. All arrows and crosses must be shown.
2. Make navigability completely invisible. No arrows or crosses are shown.
3. Suppress all crosses. Bidirectional associations have no arrows. Unidirectional associations have a single arrow.

These three idioms are summarized in Figure 9.14.

Idiom 1 makes navigability fully visible, but it can tend to clutter the diagrams.

Idiom 2 should usually be avoided, because it hides far too much valuable information.

Idiom 3 is a reasonable compromise. In fact, idiom 3 is the option that is used, almost exclusively, in practice. Because it represents current best practice, it is the option we use consistently throughout this book. The main advantages of idiom 3 are that it doesn't clutter the diagrams with too many

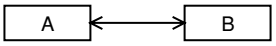
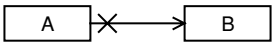
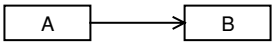
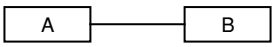
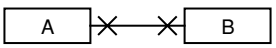
| UML 2 navigability idioms | | | |
|---|---|---|---|
| UML 2 syntax | Idiom 1:<br>Strict UML 2 navigability | Idiom 2:<br>No navigability | Idiom3:<br>Standard practice |
|  | A to B is navigable<br>B to A is navigable | | |
|  | A to B is navigable<br>B to A is not navigable | | |
|  | A to B is navigable<br>B to A is undefined | | A to B is navigable<br>B to A is not navigable |
|  | A to B is undefined<br>B to A is undefined | A to B is undefined<br>B to A is undefined | A to B is navigable<br>B to A is navigable |
|  | A to B is not navigable<br>B to A is not navigable | | |

**Figure 9.14**

arrows or crosses and that it is backwards compatible with earlier versions of UML. However, it does have disadvantages.

- It is not possible to tell from the diagram if navigability is present or if it has not yet been defined.
- It changes the meaning of the single arrowhead from navigable/undefined to navigable/not navigable. This is unfortunate but is just the way it is.
- You can't show associations that are not navigable in either direction (a cross at each end). These are useless in day-to-day modeling so it is not really an issue.

You can see a summary of idiom 3 in Figure 9.15.

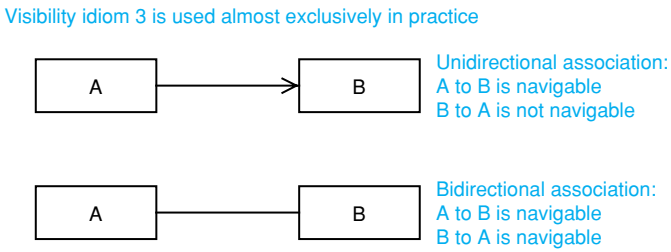



**Figure 9.15**

Even if an association is not navigable, it may still be possible to traverse the association but the cost will be high.

Even if an association is not navigable in a particular direction, it might *still* be possible to traverse the relationship in that direction. However, the computational cost of the traversal is likely to be very high. In the example in Figure 9.13, even though you can't navigate *directly* back from Product to Order, you could still find the Order object associated with a particular Product object by searching through all of the Order objects in turn. You have then traversed a non-navigable relationship, but at high computational cost. One-way navigability is like a one-way street—you might not be able to go down it directly, but you might still be able to get to the end of it by some other (longer) route.

If there is a role name on the target end of the relationship, objects of the source class may reference objects of the target class by using this role name.

In terms of implementation in OO languages, navigability implies that the source object holds an object reference to the target object. The source object may use this object reference to send messages to the target object. You could represent that on an object diagram as a unidirectional link with associated message.

### 9.4.4 Associations and attributes

There is a close link between class associations and class attributes.

An association between a source class and a target class means that objects of the source class can hold an object reference to objects of the target class. Another way to look at this is that an association is equivalent to the source class having a pseudo-attribute of the target class. An object of the source class can refer to an object of the target class by using this pseudo-attribute; see Figure 9.16.

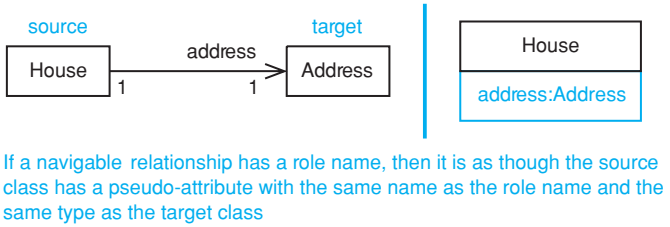



**Figure 9.16**

There is no commonly used OO programming language that has a specific language construct to support associations. Therefore, when code is automatically generated from a UML model, one-to-one associations turn into attributes of the source class.

In Figure 9.17, the generated code has a `House` class that contains an attribute called `address`, which is of type `Address`. Notice how the role name provides the attribute name, and the class on the end of the association provides the attribute class. The Java code below was generated from the model in Figure 9.17:

```
public class House
{
    private Address address;
}
```

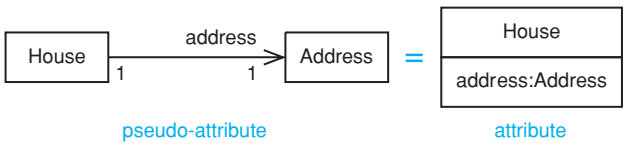



**Figure 9.17**

You can see that there is a class House that has one attribute called address that is of type Address. Notice that the address attribute has private visibility—this is typically the default for most code generation.

Target multiplicities greater than 1 are implemented as either

- an attribute of type array (a construct that is supported in most languages); or
- an attribute of some type that is a collection.

Collections are just classes whose instances have the specialized behavior of being able to store and retrieve references to other objects. A common Java example of a collection is a Vector, but there are many more. We discuss collections in more detail in Section 18.10.

This notion of pseudo-attributes is fine for one-to-one and one-to-many relationships, but it begins to break down when you consider many-to-many relationships. You will see how these are implemented in Chapter 18.

You use associations only when the target class is an important part of the model. Otherwise, you model the relationship by using attributes. Important classes are business classes that describe part of the business domain. Unimportant classes are library components such as String classes and Date and Time classes.

To some extent, the choice of explicit associations versus attributes is a matter of style. The best approach is always one in which the model and the diagrams express the problem clearly and precisely. Often it is clearer to show an association to another class than to model the same relationship as an attribute that would be much harder to see. When the target multiplicity is greater than 1, this is a pretty good indication that the target is important to the model, and so you generally use associations to model the relationship.

If the target multiplicity is exactly 1, the target object may actually be just a part of the source, and so not worth showing as an association—it may be better modeled as an attribute. This is especially true if the multiplicity is exactly 1 at *both* ends of the relationship (as in Figure 9.17) where neither source nor target can exist alone.

### 9.4.5 Association classes

A common problem in OO modeling is this: when you have a many-to-many relationship between two classes, there are sometimes some attributes that can't easily be accommodated in either of the classes. We can illustrate this by considering the simple example in Figure 9.18.

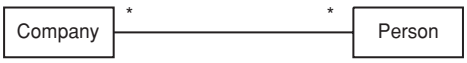



**Figure 9.18**

At first glance, this seems like a fairly innocuous model:

- each Person object can work for many Company objects;
- each Company object can employ many Person objects.

However, what happens if you add the business rule that each Person has a salary with each Company they are employed by? Where should the salary be recorded—in the Person class or in the Company class?

You can't really make the Person salary an attribute of the Person class, as each Person instance may work for many Companies and may have a different salary with each Company. Similarly, you can't really make the Person salary an attribute of Company, as each Company instance employs many Persons, all with potentially different salaries.

The answer is that the salary is actually a *property of the association itself*. For each employment association that a Person object has with a Company object, there is a specific salary.

UML allows you to model this situation with an association class as shown in Figure 9.19. It is important to understand this syntax—many people think that the association class is just the box hanging off the association. However, nothing could be further from the truth. The association class is actually the association line (including all role names and multiplicities), the dashed descending line, and the class box on the end of the dashed line. In short, it is the whole lot—everything shown in the indicated area.

An association class is an association that is also a class.

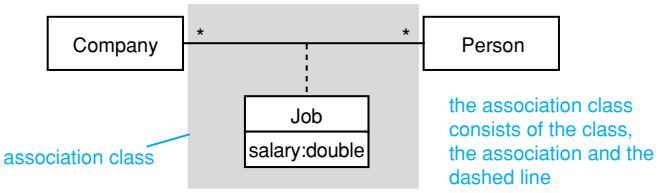



**Figure 9.19**

In fact, an association class is an association that is *also* a class. Not only does it connect two classes like an association, it defines a set of features that belong to the association itself. Association classes can have attributes, operations, and other associations.

An association class means that there can only be one link between any two objects at any point in time.

Instances of the association class are really *links* that have attributes and operations. The unique identity of these links is determined *exclusively* by the identities of the objects on either end. This factor constrains the semantics of the association class—you can only use it when there is a *single unique link* between two objects at any point in time. This is simply because each link, which is an instance of the association class, must have its own unique identity. In Figure 9.19, using the association class means that you constrain the model such that for a given Person object and a given Company object, there can only be *one* Job object. In other words, each Person can only have one Job with a given Company.

If, however, you have the situation where a given Person object can have more than one Job with a given Company object, then you can't use an association class—the semantics just don't match!

A reified association allows more than one link between any two objects at a particular point in time.

But you still need somewhere to put the salary for each Company/Job/Person combination, and so you reify (make real) the relationship by expressing it as a normal class. In Figure 9.20, Job is now just an ordinary class, and you can see that a Person may have many Jobs where each Job is for exactly one Company.

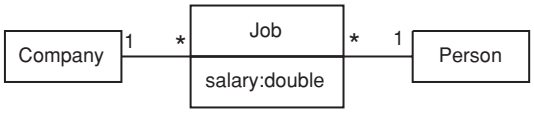



**Figure 9.20**

To be frank, many object modelers just don't understand the semantic difference between association classes and reified relationships, and the two are therefore often used interchangeably. However, the difference is really very simple: you can use association classes *only* when each link has a unique identity. Just remember that link identity is determined by the identities of the objects on the ends of the link.

## 9.4.6 Qualified associations

You can use a qualified association to reduce an n-to-many association to an n-to-one association by specifying a unique object (or group of objects) from the target set. They are very useful modeling elements as they illustrate how you can look up, or navigate to, specific objects in a collection.

Consider the model in Figure 9.21. A Club object is linked to a set of Member objects, and a Member object is likewise linked to exactly one Club object.

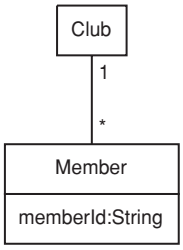



**Figure 9.21**

A qualified association selects a single member from the target set.

The following question arises: given a Club object that is linked to a set of Member objects, how could you navigate to one specific Member object? Clearly, you need some unique key that you can use to look up a particular Member object from the set. This is known as a qualifier. Many qualifiers are possible (name, credit card number, social security number), but in the example above, every Member object has a memberId attribute value that is unique to that object. This, then, is the look-up key in this model.

You can show this look-up on the model by appending a qualifier to the Club end of the association. It is important to recognize that this qualifier belongs to the *association end* and *not* to the Club class. This qualifier specifies a unique key, and in doing so resolves the one-to-many relationship to a one-to-one relationship as shown in Figure 9.22.

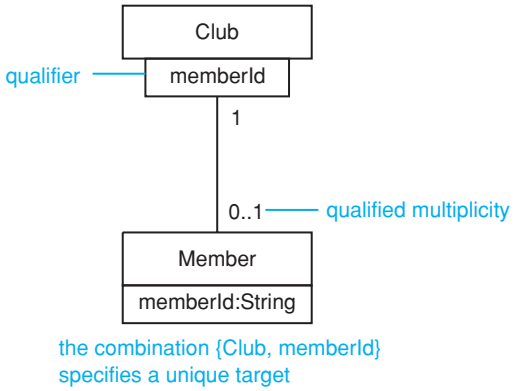



**Figure 9.22**

Qualified associations are a great way of showing how you select a specific object from a set by using a unique key. Qualifiers *usually* refer to an attribute on the target class, but may be some other expression provided it is understandable and selects a single object from the set.

## 9.5 What is a dependency?

A dependency indicates a relationship between two or more model elements whereby a change to one element (the supplier) may affect or supply information needed by the other element (the client). In other words, the client depends in some way on the supplier. We use dependencies to model relationships between classifiers where one classifier depends on the other in some way, but the relationship is not really an association or generalization.

In a dependency relationship, the client depends in some way on the supplier.

For example, you may pass an object of one class as a parameter to an operation of an object of a different class. There is clearly some sort of relationship between the classes of those objects, but it is not really an association. You can use the dependency relationship (specialized by certain predefined stereotypes) as a catch-all to model this kind of relationship. You have already seen one type of dependency, the «instantiate» relationship, but there are many more. We look at the common dependency stereotypes in the next sections.

UML 2 specifies three basic types of dependency, shown in Table 9.2. We include a discussion of these for completeness, but in day-to-day modeling you rarely use anything other than a plain dashed dependency arrow and you typically don't bother specifying the type of dependency.

**Table 9.2**

| Type | Semantics |
|---|---|
| Usage | The client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency |
| Abstraction | This indicates a relationship between client and supplier, where the supplier is more abstract than the client.<br><br>What do we mean by "more abstract"? This could mean that the supplier is at a different point in development than the client (e.g., in the analysis model rather than the design model) |
| Permission | The supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents |

Dependencies don't just occur between classes. They can commonly occur between

- packages and packages;
- objects and classes.

They can also occur between an operation and a class, although it is quite rare to show this explicitly on a diagram because it is usually too great a level of detail. Some examples of different types of dependency are shown in Figure 9.23, and we discuss these in the remaining sections of this chapter.

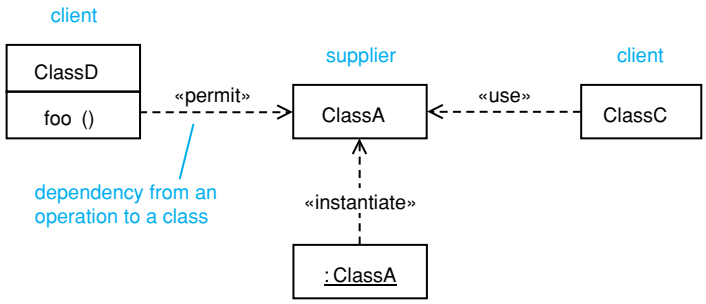



**Figure 9.23**

Most of the time, you just use an unadorned dotted arrow to indicate a dependency and don't worry about what type of dependency it is. In fact, the type of the dependency is often clear without a stereotype just from context. However, if you want or need to be more specific about the type of dependency, then UML defines a whole range of standard stereotypes that you can use.

## 9.5.1 Usage dependencies

There are five usage dependencies: «use», «call», «parameter», «send», and «instantiate». We look at each of these in the next few subsections.

### *9.5.1.1* «use»

The most common dependency stereotype is «use», which simply states that the client makes use of the supplier in some way. If you see just a dashed dependency arrow with no stereotype, then you can be pretty sure that «use» is intended.

Figure 9.24 shows two classes, A and B, that have a «use» dependency between them. This dependency is generated by any of the following cases.

1. An operation of class A needs a parameter of class B.
2. An operation of class A returns a value of class B.
3. An operation of class A uses an object of class B somewhere in its implementation, but *not* as an attribute.

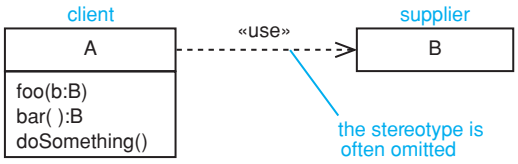



**Figure 9.24**

Cases 1 and 2 are straightforward, but case 3 is more interesting. You would have this case if one of the operations of class A created a transient object of class B. Here is a Java code fragment for this case:

```
class A
{
  ...
  void doSomething()
    {
      B myB = new B();
      // Use myB in some way
      ...
    }
}
```

Although you can use a single «use» dependency as a catch-all for the three cases listed above, there are other more specific dependency stereotypes that you could apply.

You can model cases 1 and 2 more accurately by a «parameter» dependency, and case 3 by a «call» dependency. However, this is a level of detail that is rarely (if ever) required in a UML model, and most modelers find it much clearer and easier to just put a single «use» dependency between the appropriate classes as shown above.

### *9.5.1.2* «call»

The «call» dependency is between operations—the client operation invokes the supplier operation. This type of dependency tends not to be very widely used in UML modeling. It applies at a deeper level of detail than most modelers are prepared to go. Also, very few modeling tools currently support dependencies between operations.

### *9.5.1.3* «parameter»

The supplier is a parameter of the client operation.

#### *9.5.1.4* «send»

The client is an operation that sends the supplier (which must be a signal) to some unspecified target. We discuss signals in Section 15.6 but, for now, just think of them as special types of classes used to transfer data between the client and the target.

#### *9.5.1.5* «instantiate»

The client is an instance of the supplier.

### 9.5.2 Abstraction dependencies

Abstraction dependencies model dependencies between things that are at different levels of abstraction. An example might be a class in an analysis model, and the same class in the design model. There are four abstraction dependencies: «trace», «substitute», «refine», and «derive».

#### *9.5.2.1* «trace»

You often use a «trace» dependency to illustrate a relationship in which the supplier and the client represent the same concept but are in different models. For example, the supplier and the client might be at different stages of development. The supplier could be an analysis view of a class, and the client a more detailed design view. You could also use «trace» to show a relationship between a functional requirement such as "The ATM shall allow the withdrawal of cash up to the credit limit of the card" and the use case that supports this requirement.

#### *9.5.2.2* «substitute»

The «substitute» relationship indicates that the client may be substituted for the supplier at runtime. The substitutability is based on the client and the supplier conforming to common contracts and interfaces, that is, they must both make available the same set of services. Note that this substitutability is *not* achieved through specialization/generalization relationships between the client and supplier (we discuss specialization/generalization in Section 10.2). In fact, «substitute» is specifically designed to be used in those environments that do *not* support specialization/generalization.

#### *9.5.2.3* «refine»

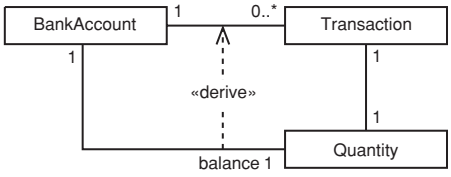
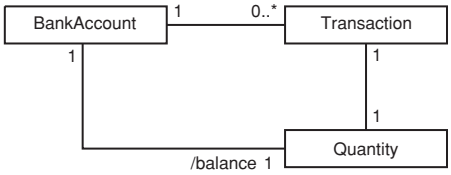
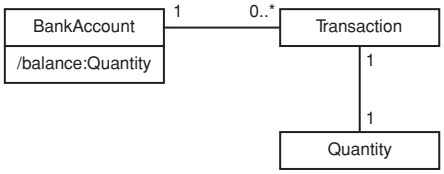
Whereas the «trace» dependency is between elements in different models, «refine» may be used between elements in the same model. For example, you

may have two versions of a class in a model, one of which is optimized for performance. As performance optimization is a type of refinement, you can model this as a «refine» dependency between the two classes, along with a note stating the nature of the refinement.

### 9.5.2.4 «derive»

You use the «derive» stereotype when you want to show explicitly that a thing can be derived in some way from some other thing. For example, if you have a BankAccount class and the class contains a list of Transactions where each Transaction contains a Quantity of money, you can always calculate the current balance on demand by summing Quantity over all the Transactions. There are three ways of showing that the balance of the account (a Quantity) can be derived. These are shown in Table 9.3.

**Table 9.3**

| Model | Description |
|---|---|
|  | The BankAccount class has a derived association to Quantity where Quantity plays the role of the balance of the BankAccount<br><br>This model emphasizes that the balance is derived from the BankAccount's collection of Transactions |
|  | In this case a slash is used on the role name to indicate that the relationship between BankAccount and Quantity is derived<br><br>This is less explicit as it does not show what the balance is derived from |
|  | Here the balance is shown as a derived attribute – this is indicated by the slash that prefixes the attribute name<br><br>This is the most concise expression of the dependency |

All of these ways of showing that balances can be derived are equivalent, although the first model in Table 9.3 is the most explicit. We tend to prefer explicit models.

## 9.5.3 Permission dependencies

Permission dependencies are about expressing the ability of one thing to access another thing. There are three permission dependencies: «access», «import», and «permit».

### *9.5.3.1* «access»

The «access» dependency is between packages. Packages are used in UML to group things. The essential point here is that «access» allows one package to access all of the public contents of another package. However, packages each define a namespace and with «access» the namespaces remain separate. This means that items in the client package must use pathnames when they want to refer to items in the supplier package. See Chapter 11 for a more detailed discussion.

### *9.5.3.2* «import»

The «import» dependency is conceptually similar to «access» except that the namespace of the supplier is merged into the namespace of the client. This allows elements in the client to access elements in the supplier without having to qualify element names with the package name. However, it can sometimes give rise to namespace clashes when an element in the client has the same name as an element in the supplier. Clearly, in this case you must use pathnames to resolve the conflict. Chapter 11 provides a more detailed discussion.

### *9.5.3.3* «permit»

The «permit» dependency allows a *controlled* violation of encapsulation, but on the whole it should be avoided. The client element has access to the supplier element, whatever the *declared* visibility of the supplier. There is often a «permit» dependency between two very closely related classes where it is advantageous (probably for performance reasons) for the client class to access the private members of the supplier. Not all computer languages support «permit» dependencies—C++ allows a class to declare friends that have permission to access its private members, but this feature has, perhaps wisely, been excluded from Java and C#.

## 9.6 What we have learned

In this chapter you have begun to look at relationships, which are the glue of UML models. You have learned the following.

- Relationships are semantic connections between things.
- Connections between objects are called links.
  - A link occurs when one object holds an object reference to another object.
  - Objects realize system behavior by collaborating:
    - collaboration occurs when objects send each other messages across links;
    - when a message is received by an object, it executes the appropriate operation.
  - Different OO languages implement links in different ways.
- Object diagrams show objects and their links at a particular point in time.
  - They are snapshots of an executing OO system at a particular time.
  - Objects may adopt roles with respect to each other – the role played by an object in a link defines the semantics of its part in the collaboration.
  - N-ary links may connect more than two objects – they are drawn as a diamond with a path to each object but are not widely used.
- Paths are lines connecting UML modeling elements:
  - orthogonal style – straight lines with right-angled bends;
  - oblique style – slanted lines;
  - curved style – curved lines;
  - be consistent and stick to one style or the other, unless mixing styles increases the readability of the diagram (it usually doesn't).
- Associations are semantic connections between classes.
  - If there is a link between two objects, there *must* be an association between the classes of those objects.
  - Links are instances of associations just as objects are instances of classes.
  - Associations may optionally have the following.
    - Association name:
      - may be prefixed or postfixed with a small black arrowhead to indicate the direction in which the name should be read;
      - should be a verb or verb phrase;
      - in lowerCamelCase;
      - use either an association name or role names but *not* both.

  - Role names on one or both association ends:
    - should be a noun or noun phrase describing the semantics of the role;
    - in lowerCamelCase.
  - Multiplicity:
    - indicates the number of objects that can be involved in the relationship at any point in time;
    - objects may come and go, but multiplicity constrains the number of objects in the relationship at any point in time;
    - multiplicity is specified by a comma-separated list of intervals, for example, 0..1, 3..5;
    - there is no default multiplicity – if multiplicity is not explicitly shown, then it is undecided.
  - Navigability:
    - shown by an arrowhead on one end of the relationship – if a relationship has no arrowheads, then it is bidirectional;
    - navigability indicates that you can traverse the relationship in the direction of the arrow;
    - you may also be able to traverse back the other way, but it will be computationally expensive to do so.
- An association between two classes is equivalent to one class having a pseudo-attribute that can hold a reference to an object of the other class:
  - you can often use associations and attributes interchangeably;
  - use association when you have an important class on the end of the association that you wish to emphasize;
  - use attributes when the class on the end of the relationship is unimportant (e.g., a library class such as String or Date).
- An association class is an association that is also a class:
  - it may have attributes, operations, and relationships;
  - you can use an association class when there is exactly one unique link between any pair of objects at any point in time;
  - if a pair of objects may have many links to each other at a given point in time, then you reify the relationship by replacing it with a normal class.
- Qualified associations use a qualifier to select a unique object from the target set:
  - the qualifier must be a unique key into the target set;
  - qualified associations reduce the multiplicity of n-to-many relationships, to n-to-one;
  - they are a useful way of drawing attention to unique identifiers.

- Dependencies are relationships in which a change to the supplier affects or supplies information to the client.
  - The client depends on the supplier in some way.
  - Dependencies are drawn as a dashed arrow from client to supplier.
  - Usage dependencies:
    - «use» – the client makes use of the supplier in some way – this is the catch-all;
    - «call» – the client operation invokes the supplier operation;
    - «parameter» – the supplier is a parameter or return value from one of the client's operations;
    - «send» – the client sends the supplier (which must be a signal) to the specified target;
    - «instantiate» – the client is an instance of the supplier.
  - Abstraction dependencies:
    - «trace» – the client is a historical development of the supplier;
    - «substitute» – the client can be substituted for the supplier at runtime;
    - «refine» – the client is a version of the supplier;
    - «derive» – the client can be derived in some way from the supplier:
      - you may show derived relationships explicitly by using a «derive» dependency;
      - you may show derived relationships by prefixing the role or relationship name with a slash;
      - you may show derived attributes by prefixing the attribute name with a slash.
  - Permission dependencies:
    - «access» – a dependency between packages where the client package can access all of the public contents of the supplier package – the namespaces of the packages remain separate;
    - «import» – a dependency between packages where the client package can access all of the public contents of the supplier package – the namespaces of the packages are merged;
    - «permit» – a controlled violation of encapsulation where the client may access the private members of the supplier – this is not widely supported and should be avoided if possible.