

Chapter 3

Class Diagrams: The Essentials

If someone were to come up to you in a dark alley and say, “Psst, wanna see a UML diagram?” that diagram would probably be a class diagram. The majority of UML diagrams I see are class diagrams.

The class diagram is not only widely used but also subject to the greatest range of modeling concepts. Although the basic elements are needed by everyone, the advanced concepts are used less often. Therefore, I’ve broken my discussion of class diagrams into two parts: the essentials (this chapter) and the advanced (Chapter 5).

A **class diagram** describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term **feature** as a general term that covers properties and operations of a class.

Figure 3.1 shows a simple class model that would not surprise anyone who has worked with order processing. The boxes in the diagram are classes, which are divided into three compartments: the name of the class (in bold), its attributes, and its operations. Figure 3.1 also shows two kinds of relationships between classes: associations and generalizations.

Properties

Properties represent structural features of a class. As a first approximation, you can think of properties as corresponding to fields in a class. The reality is rather involved, as we shall see, but that’s a reasonable place to start.

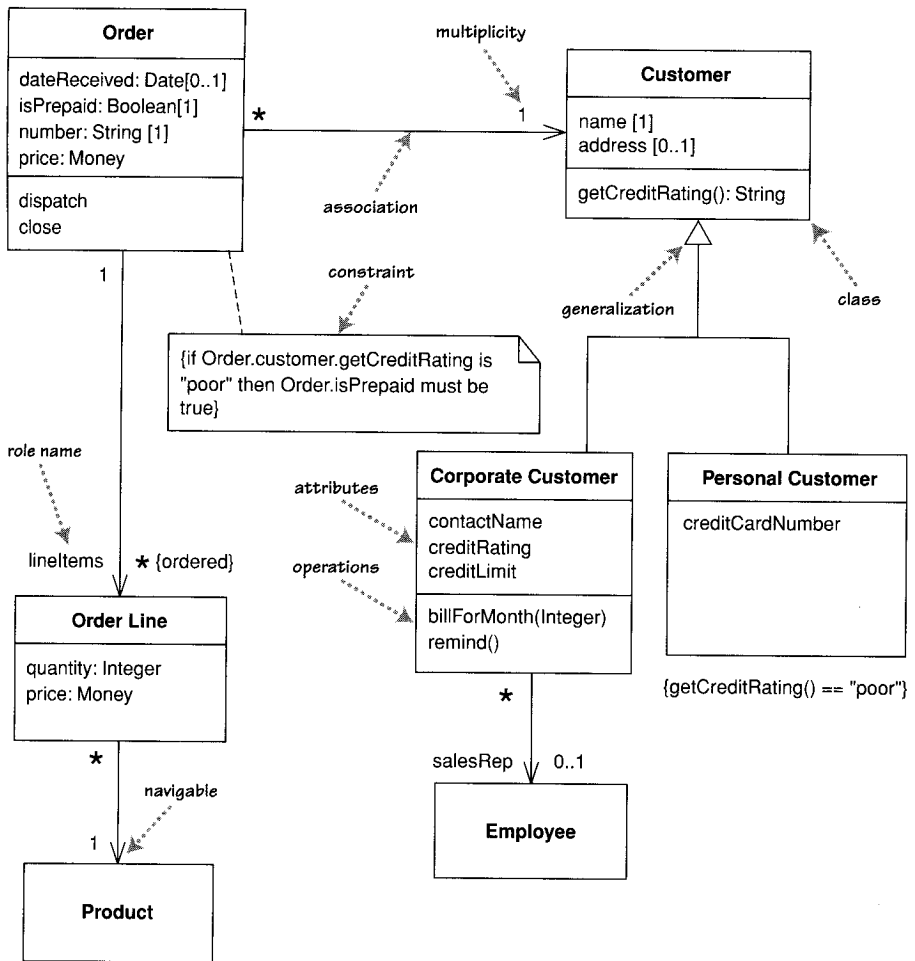


Figure 3.1 A simple class diagram

Properties are a single concept, but they appear in two quite distinct notations: attributes and associations. Although they look quite different on a diagram, they are really the same thing.

Attributes

The **attribute** notation describes a property as a line of text within the class box itself. The full form of an attribute is:

visibility name: type multiplicity = default {property-string}

An example of this is:

```
- name: String [1] = "Untitled" {readOnly}
```

Only the name is necessary.

- This visibility marker indicates whether the attribute is public (+) or private (-); I'll discuss other visibilities on page 83.
- The name of the attribute—how the class refers to the attribute—roughly corresponds to the name of a field in a programming language.
- The type of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in a programming language.
- I'll explain multiplicity on page 38.
- The default value is the value for a newly created object if the attribute isn't specified during creation.
- The {property-string} allows you to indicate additional properties for the attribute. In the example, I used {readOnly} to indicate that clients may not modify the property. If this is missing, you can usually assume that the attribute is modifiable. I'll describe other property strings as we go.

Associations

The other way to notate a property is as an association. Much of the same information that you can show on an attribute appears on an association. Figures 3.2 and 3.3 show the same properties represented in the two different notations.

An **association** is a solid line between two classes, directed from the source class to the target class. The name of the property goes at the target end of the

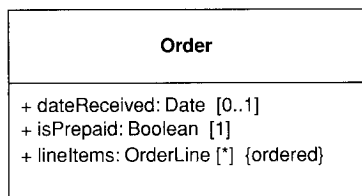


Figure 3.2 *Showing properties of an order as attributes*

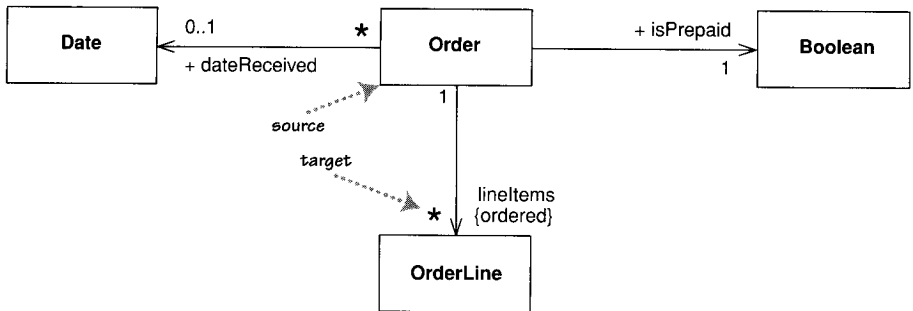


Figure 3.3 Showing properties of an order as associations

association, together with its multiplicity. The target end of the association links to the class that is the type of the property.

Although most of the same information appears in both notations, some items are different. In particular, associations can show multiplicities at both ends of the line.

With two notations for the same thing, the obvious question is, Why should you use one or the other? In general, I tend to use attributes for small things, such as dates or Booleans—in general, value types (page 73)—and associations for more significant classes, such as customers and orders. I also tend to prefer to use class boxes for classes that are significant for the diagram, which leads to using associations, and attributes for things less important for that diagram. The choice is much more about emphasis than about any underlying meaning.

Multiplicity

The **multiplicity** of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are

- 1 (An order must have exactly one customer.)
- 0..1 (A corporate customer may or may not have a single sales rep.)
- * (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)

More generally, multiplicities are defined with a lower bound and an upper bound, such as 2..4 for players of a game of canasta. The lower bound may be

any positive number or zero; the upper is any positive number or * (for unlimited). If the lower and upper bounds are the same, you can use one number; hence, 1 is equivalent to 1..1. Because it's a common case, * is short for 0..*.

In attributes, you come across various terms that refer to the multiplicity.

- **Optional** implies a lower bound of 0.
- **Mandatory** implies a lower bound of 1 or possibly more.
- **Single-valued** implies an upper bound of 1.
- **Multivalued** implies an upper bound of more than 1: usually *.

If I have a multivalued property, I prefer to use a plural form for its name.

By default, the elements in a multivalued multiplicity form a set, so if you ask a customer for its orders, they do not come back in any order. If the ordering of the orders in association has meaning, you need to add {ordered} to the association end. If you want to allow duplicates, add {nonunique}. (If you want to explicitly show the default, you can use {unordered} and {unique}.) You may also see collection-oriented names, such as {bag} for unordered, nonunique.

UML 1 allowed discontinuous multiplicities, such as 2, 4 (meaning 2 or 4, as in cars in the days before minivans). Discontinuous multiplicities weren't very common and UML 2 removed them.

The default multiplicity of an attribute is [1]. Although this is true in the meta-model, you can't assume that an attribute in a diagram that's missing a multiplicity has a value of [1], as the diagram may be suppressing the multiplicity information. As a result, I prefer to explicitly state a [1] multiplicity if it's important.

Programming Interpretation of Properties

As with anything else in the UML, there's no one way to interpret properties in code. The most common software representation is that of a field or property of your programming language. So the Order Line class from Figure 3.1 would correspond to something like the following in Java:

```
public class OrderLine...
    private int quantity;
    private Money price;
    private Order order;
    private Product product
```

In a language like C#, which has properties, it would correspond to:

```
public class OrderLine ...
    public int Quantity;
    public Money Price;
    public Order Order;
    public Product Product;
```

Note that an attribute typically corresponds to public properties in a language that supports properties but to private fields in a language that does not. In a language without properties, you may see the fields exposed through accessor (getting and setting) methods. A read-only attribute will have no setting method (with fields) or set action (for properties). Note that if you don't give a name for a property, it's common to use the name of the target class.

Using private fields is a very implementation-focused interpretation of the diagram. A more interface-oriented interpretation might instead concentrate on the getting methods rather than the underlying data. In this case, we might see the Order Line's attributes corresponding to the following methods:

```
public class OrderLine...
    private int quantity;
    private Product product;
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public Money getPrice() {
        return product.getPrice().multiply(quantity);
    }
}
```

In this case, there is no data field for price; instead, it's a computed value. But as far as clients of the Order Line class are concerned, it looks the same as a field. Clients can't tell what is a field and what is computed. This information hiding is the essence of encapsulation.

If an attribute is multivalued, this implies that the data concerned is a collection. So an Order class would refer to a collection of Order Lines. Because this multiplicity is ordered, that collection must be ordered, (such as a List in Java or an IList in .NET). If the collection is unordered, it should, strictly, have no meaningful order and thus be implemented with a set, but most people implement unordered attributes as lists as well. Some people use arrays, but the UML implies an unlimited upper bound, so I almost always use a collection for data structure.

Multivalued properties yield a different kind of interface to single-valued properties (in Java):

```
class Order {
    private Set lineItems = new HashSet();
    public Set getLineItems() {
        return Collections.unmodifiableSet(lineItems);
    }
    public void addLineItem (OrderItem arg) {
        lineItems.add (arg);
    }
    public void removeLineItem (OrderItem arg) {
        lineItems.remove(arg);
    }
}
```

In most cases, you don't assign to a multivalued property; instead, you update with add and remove methods. In order to control its Line Items property, the order must control membership of that collection; as a result, it shouldn't pass out the naked collection. In this case, I used a protection proxy to provide a read-only wrapper to the collection. You can also provide a nonupdatable iterator or make a copy. It's okay for clients to modify the member objects, but the clients shouldn't directly change the collection itself.

Because multivalued attributes imply collections, you almost never see collection classes on a class diagram. You would show them only in very low level implementation diagrams of collections themselves.

You should be very afraid of classes that are nothing but a collection of fields and their accessors. Object-oriented design is about providing objects that are able to do rich behavior, so they shouldn't be simply providing data to other objects. If you are making repeated calls for data by using accessors, that's a sign that some behavior should be moved to the object that has the data.

These examples also reinforce the fact that there is no hard-and-fast correspondence between the UML and code, yet there is a similarity. Within a project team, team conventions will lead to a closer correspondence.

Whether a property is implemented as a field or as a calculated value, it represents something an object can always provide. You shouldn't use a property to model a transient relationship, such as an object that is passed as a parameter during a method call and used only within the confines of that interaction.

Bidirectional Associations

The associations we've looked at so far are called unidirectional associations. Another common kind of association is a bidirectional association, such as Figure 3.4.

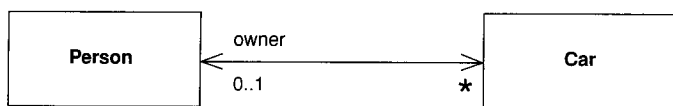


Figure 3.4 A bidirectional association

A bidirectional association is a pair of properties that are linked together as inverses. The Car class has property `owner:Person[1]`, and the Person class has a property `cars:Car[*]`. (Note how I named the cars property in the plural form of the property’s type, a common but non-normative convention.)

The inverse link between them implies that if you follow both properties, you should get back to a set that contains your starting point. For example, if I begin with a particular MG Midget, find its owner, and then look at its owner’s cars, that set should contain the Midget that I started from.

As an alternative to labeling an association by a property, many people, particularly if they have a data-modeling background, like to label an association by using a verb phrase (Figure 3.5) so that the relationship can be used in a sentence. This is legal and you can add an arrow to the association to avoid ambiguity. Most object modelers prefer to use a property name, as that corresponds better to responsibilities and operations.

Some people name every association in some way. I choose to name an association only when doing so improves understanding. I’ve seen too many associations with such names as “has” or “is related to.”

In Figure 3.4, the bidirectional nature of the association is made obvious by the **navigability arrows** at both ends of the association. Figure 3.5 has no arrows; the UML allows you to use this form either to indicate a bidirectional association or when you aren’t showing navigability. My preference is to use the double-headed arrow of Figure 3.4 when you want to make it clear that you have a bidirectional association.

Implementing a bidirectional association in a programming language is often a little tricky because you have to be sure that both properties are kept



Figure 3.5 Using a verb phrase to name an association

synchronized. Using C#, I use code along these lines to implement a bidirectional association:

```
class Car...
    public Person Owner {
        get {return _owner;}
        set {
            if (_owner != null) _owner.friendCars().Remove(this);
            _owner = value;
            if (_owner != null) _owner.friendCars().Add(this);
        }
    }
    private Person _owner;
...

class Person ...
    public IList Cars {
        get {return ArrayList.ReadOnly(_cars);}
    }
    public void AddCar(Car arg) {
        arg.Owner = this;
    }
    private IList _cars = new ArrayList();
    internal IList friendCars() {
        //should only be used by Car.Owner
        return _cars;
    }
....
```

The primary thing is to let one side of the association—a single-valued side, if possible—control the relationship. For this to work, the slave end (Person) needs to leak the encapsulation of its data to the master end. This adds to the slave class an awkward method, which shouldn't really be there, unless the language has fine-grained access control. I've used the naming convention of "friend" here as a nod to C++, where the master's setter would indeed be a friend. Like much property code, this is pretty boilerplate stuff, which is why many people prefer to use some form of code generation to produce it.

In conceptual models, navigability isn't an important issue, so I don't show any navigability arrows on conceptual models.

Operations

Operations are the actions that a class knows to carry out. Operations most obviously correspond to the methods on a class. Normally, you don't show

those operations that simply manipulate properties, because they can usually be inferred.

The full UML syntax for operations is:

```
visibility name (parameter-list) : return-type {property-string}
```

- This visibility marker is public (+) or private (-); others on page 83.
- The name is a string.
- The parameter-list is the list of parameters for the operation.
- The return-type is the type of the returned value, if there is one.
- The property-string indicates property values that apply to the given operation.

The parameters in the parameter list are notated in a similar way to attributes. The form is:

```
direction name: type = default value
```

- The name, type, and default value are the same as for attributes.
- The direction indicates whether the parameter is input (in), output (out) or both (inout). If no direction is shown, it's assumed to be in.

An example operation on account might be:

```
+ balanceOn (date: Date) : Money
```

With conceptual models, you shouldn't use operations to specify the interface of a class. Instead, use them to indicate the principal responsibilities of that class, perhaps using a couple of words summarizing a CRC responsibility (page 65).

I often find it useful to distinguish between operations that change the state of the system and those that don't. UML defines a **query** as an operation that gets a value from a class without changing the system state—in other words, without side effects. You can mark such an operation with the property string {query}. I refer to operations that do change state as **modifiers**, also called commands.

Strictly, the difference between query and modifiers is whether they change the observable state [Meyer]. The observable state is what can be perceived from the outside. An operation that updates a cache would alter the internal state but would have no effect that's observable from the outside.

I find it helpful to highlight queries, as you can change the order of execution of queries and not change the system behavior. A common convention is to try

to write operations so that modifiers do not return a value; that way, you can rely on the fact that operations that return a value are queries. [Meyer] refers to this as the Command-Query separation principle. It's sometimes awkward to do this all the time, but you should do it as much as you can.

Other terms you sometimes see are getting methods and setting methods. A **getting method** returns a value from a field (and does nothing else). A **setting method** puts a value into a field (and does nothing else). From the outside, a client should not be able to tell whether a query is a getting method or a modifier is a setting method. Knowledge of getting and setting methods is entirely internal to the class.

Another distinction is between operation and method. An **operation** is something that is invoked on an object—the procedure declaration—whereas a **method** is the body of a procedure. The two are different when you have polymorphism. If you have a supertype with three subtypes, each of which overrides the supertype's `getPrice` operation, you have one operation and four methods that implement it.

People usually use the terms *operation* and *method* interchangeably, but there are times when it is useful to be precise about the difference.

Generalization

A typical example of **generalization** involves the personal and corporate customers of a business. They have differences but also many similarities. The similarities can be placed in a general Customer class (the supertype), with Personal Customer and Corporate Customer as subtypes.

This phenomenon is also subject to various interpretations at the various perspectives of modeling. Conceptually, we can say that Corporate Customer is a subtype of Customer if all instances of Corporate Customer are also, by definition, instances of Customer. A Corporate Customer is then a special kind of Customer. The key idea is that everything we say about a Customer—associations, attributes, operations—is true also for a Corporate Customer.

With a software perspective, the obvious interpretation is inheritance: The Corporate Customer is a subclass of Customer. In mainstream OO languages, the subclass inherits all the features of the superclass and may override any superclass methods.

An important principle of using inheritance effectively is **substitutability**. I should be able to substitute a Corporate Customer within any code that requires

a Customer, and everything should work fine. Essentially, this means that if I write code assuming I have a Customer, I can freely use any subtype of Customer. The Corporate Customer may respond to certain commands differently from another Customer, using polymorphism, but the caller should not need to worry about the difference. (For more on this, see the Liskov Substitution Principle (LSP) in [Martin].)

Although inheritance is a powerful mechanism, it brings in a lot of baggage that isn't always needed to achieve substitutability. A good example of this was in the early days of Java, when many people didn't like the implementation of the built-in Vector class and wanted to replace it with something lighter. However, the only way they could produce a class that was substitutable for Vector was to subclass it, and that meant inheriting a lot of unwanted data and behavior.

Many other mechanisms can be used to provide substitutable classes. As a result, many people like to differentiate between subtyping, or interface inheritance, and subclassing, or implementation inheritance. A class is a **subtype** if it is substitutable for its supertype, whether or not it uses inheritance. **Subclassing** is used as a synonym for regular inheritance.

Many other mechanisms are available that allow you to have subtyping without subclassing. Examples are implementing an interface (page 69) and many of the standard design patterns [Gang of Four].

Notes and Comments

Notes are comments in the diagrams. Notes can stand on their own, or they can be linked with a dashed line to the elements they are commenting (Figure 3.6). They can appear in any kind of diagram.

The dashed line can sometimes be awkward because you can't position exactly where this line ends. So a common convention is to put a very small open circle at the end of the line. Sometimes, it's useful to have an in-line comment on a diagram element. You can do this by prefixing the text with two dashes: --.

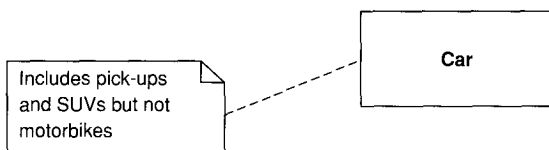


Figure 3.6 A note is used as a comment on one or more diagram elements

Dependency

A **dependency** exists between two elements if changes to the definition of one element (the **supplier**) may cause changes to the other (the **client**). With classes, dependencies exist for various reasons: One class sends a message to another; one class has another as part of its data; one class mentions another as a parameter to an operation. If a class changes its interface, any message sent to that class may no longer be valid.

As computer systems grow, you have to worry more and more about controlling dependencies. If dependencies get out of control, each change to a system has a wide ripple effect as more and more things have to change. The bigger the ripple, the harder it is to change anything.

The UML allows you to depict dependencies between all sorts of elements. You use dependencies whenever you want to show how changes in one element might alter other elements.

Figure 3.7 shows some dependencies that you might find in a multilayered application. The Benefits Window class—a user interface, or **presentation class**—is dependent on the Employee class: a **domain object** that captures the essential behavior of the system—in this case, business rules. This means that if the employee class changes its interface, the Benefits Window may have to change.

The important thing here is that the dependency is in only one direction and goes from the presentation class to the domain class. This way, we know that we can freely alter the Benefits Window without those changes having any effect on the Employee or other domain objects. I've found that a strict separation of presentation and domain logic, with the presentation depending on the domain but not vice versa, has been a valuable rule for me to follow.

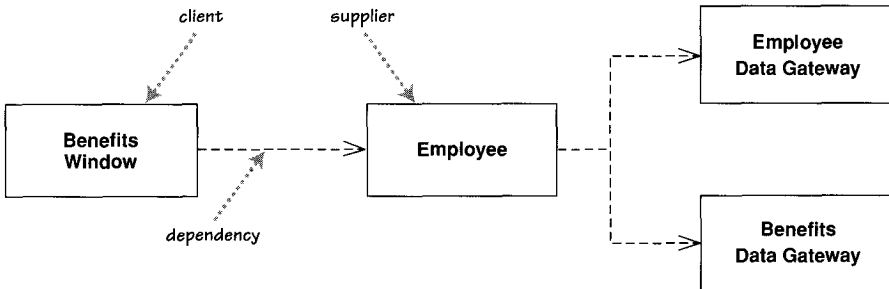


Figure 3.7 Example dependencies

A second notable thing from this diagram is that there is no direct dependency from the Benefits Window to the two Data Gateway classes. If these classes change, the Employee class may have to change. But if the change is only to the implementation of the Employee class, not its interface, the change stops there.

The UML has many varieties of dependency, each with particular semantics and keywords. The basic dependency that I've outlined here is the one I find the most useful, and I usually use it without keywords. To add more detail, you can add an appropriate keyword (Table 3.1).

The basic dependency is not a transitive relationship. An example of a **transitive** relationship is the “larger beard” relationship. If Jim has a larger beard than Grady, and Grady has a larger beard than Ivar, we can deduce that Jim has a larger beard than Ivar. Some kind of dependencies, such as substitute, are transitive, but in most cases there is a significant difference between direct and indirect dependencies, as there is in Figure 3.7.

Many UML relationships imply a dependency. The navigable association from Order to Customer in Figure 3.1 means that Order is dependent on Customer. A subclass is dependent on its superclass but not vice versa.

Table 3.1 *Selected Dependency Keywords*

Keyword	Meaning
«call»	The source calls an operation in the target.
«create»	The source creates instances of the target.
«derive»	The source is derived from the target.
«instantiate»	The source is an instance of the target. (Note that if the source is a class, the class itself is an instance of the class class; that is, the target class is a metaclass).
«permit»	The target allows the source to access the target's private features.
«realize»	The source is an implementation of a specification or interface defined by the target (page 69).
«refine»	Refinement indicates a relationship between different semantic levels; for example, the source might be a design class and the target the corresponding analysis class.
«substitute»	The source is substitutable for the target (page 45).
«trace»	Used to track such things as requirements to classes or how changes in one model link to changes elsewhere.
«use»	The source requires the target for its implementation.

Chapter 5

Class Diagrams: Advanced Concepts

The concepts described in Chapter 3 correspond to the key notations in class diagrams. Those concepts are the first ones to understand and become familiar with, as they will comprise 90 percent of your effort in building class diagrams.

The class diagram technique, however, has bred dozens of notations for additional concepts. I find that I don't use these all the time, but they are handy when they are appropriate. I'll discuss them one at a time and point out some of the issues in using them.

You'll probably find this chapter somewhat heavy going. The good news is that during your first pass through the book, you can safely skip this chapter and come back to it later.

Keywords

One of the challenges of a graphical language is that you have to remember what the symbols mean. With too many, users find it very difficult to remember what all the symbols mean. So the UML often tries to reduce the number of symbols and use keywords instead. If you find that you need a modeling construct that isn't in the UML but is similar to something that is, use the symbol of the existing UML construct but mark it with a keyword to show that you have something different

An example of this is the interface. A UML **interface** (page 69) is a class that has only public operations, with no method bodies. This corresponds to interfaces in Java, COM (Component Object Module), and CORBA. Because it's a

special kind of class, it is shown using the class icon with the keyword «interface». Keywords are usually shown as text between guillemets. As an alternative to keywords, you can use special icons, but then you run into the issue of everyone having to remember what they mean.

Some keywords, such as {abstract}, show up in curly brackets. It's never really clear what should technically be in guillemets and what should be in curlies. Fortunately, if you get it wrong, only serious UML weenies will notice—or care.

Some keywords are so common that they often get abbreviated: «interface» often gets abbreviated to «I» and {abstract} to {A}. Such abbreviations are very useful, particularly on whiteboards, but nonstandard, so if you use them, make sure you find a spot to spell out what they mean.

In UML 1, the guillemets were used mainly for stereotypes. In UML 2, stereotypes are defined very tightly, and describing what is and isn't a stereotype is beyond the scope of this book. However, because of UML 1, many people use the term *stereotype* to mean the same as *keyword*, although that is no longer correct.

Stereotypes are used as part of profiles. A **profile** takes a part of the UML and extends it with a coherent group of stereotypes for a particular purpose, such as business modeling. The full semantics of profiles are beyond this book. Unless you are into serious meta-model design, you're unlikely to need to create one yourself. You're more likely to use one created for a specific modeling purpose, but fortunately, use of a profile doesn't require you to know the gory details of how they are tied into the meta-model.

Responsibilities

Often, it's handy to show responsibilities (page 63) on a class in a class diagram. The best way to show them is as comment strings in their own compartment in the class (Figure 5.1). You can name the compartment, if you wish, but I usually don't, as there's rarely any potential for confusion.

Static Operations and Attributes

The UML refers to an operation or an attribute that applies to a class rather than to an instance as **static**. This is equivalent to static members in C-based languages. Static features are underlined on a class diagram (see Figure 5.2).

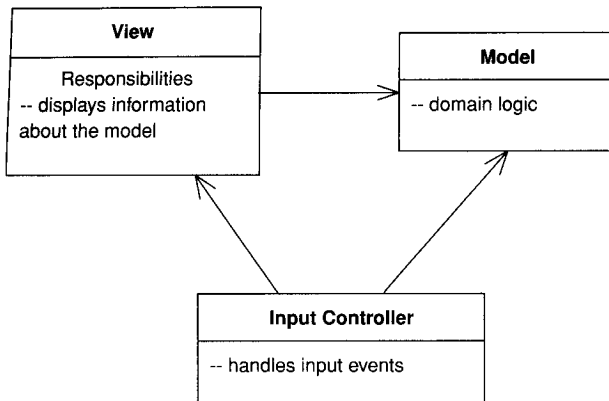


Figure 5.1 *Showing responsibilities in a class diagram*

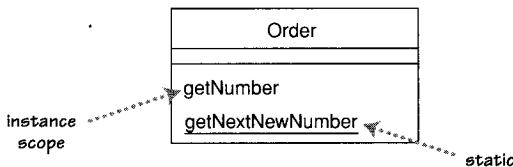


Figure 5.2 *Static notation*

Aggregation and Composition

One of the most frequent sources of confusion in the UML is aggregation and composition. It's easy to explain glibly: **Aggregation** is the part-of relationship. It's like saying that a car has an engine and wheels as its parts. This sounds good, but the difficult thing is considering what the difference is between aggregation and association.

In the pre-UML days, people were usually rather vague on what was aggregation and what was association. Whether vague or not, they were always inconsistent with everyone else. As a result, many modelers think that aggregation is important, although for different reasons. So the UML included aggregation (Figure 5.3) but with hardly any semantics. As Jim Rumbaugh says, "Think of it as a modeling placebo" [Rumbaugh, UML Reference].

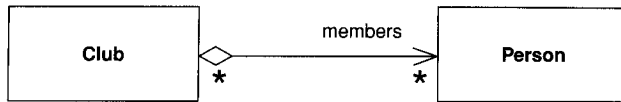


Figure 5.3 Aggregation

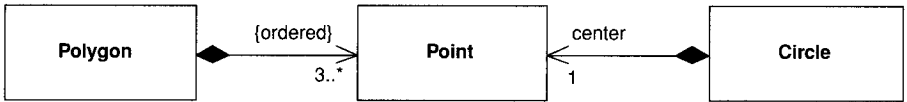


Figure 5.4 Composition

As well as aggregation, the UML has the more defined property of **composition**. In Figure 5.4, an instance of *Point* may be part of a polygon or may be the center of a circle, but it cannot be both. The general rule is that, although a class may be a component of many other classes, any instance must be a component of only one owner. The class diagram may show multiple classes of potential owners, but any instance has only a single object as its owner.

You'll note that I don't show the reverse multiplicities in Figure 5.4. In most cases, as here, it's 0..1. Its only other possible value is 1, for cases in which the component class is designed so that it can have only one other class as its owner.

The “no sharing” rule is the key to composition. Another assumption is that if you delete the polygon, it should automatically ensure that any owned *Points* also are deleted.

Composition is a good way of showing properties that own by value, properties to value objects (page 73), or properties that have a strong and somewhat exclusive ownership of particular other components. Aggregation is strictly meaningless; as a result, I recommend that you ignore it in your own diagrams. If you see it in other people's diagrams, you'll need to dig deeper to find out what they mean by it. Different authors and teams use it for very different purposes.

Derived Properties

Derived properties can be calculated based on other values. When we think about a date range (Figure 5.5), we can think of three properties: the start date,

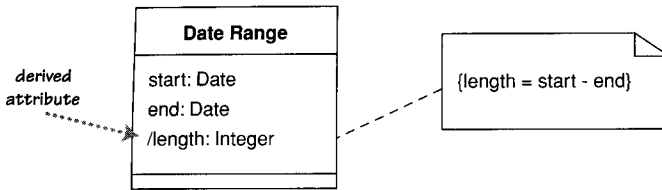


Figure 5.5 *Derived attribute in a time period*

the end date, and the number of days in the period. These values are linked, so we can think of the length as being derived from the other two values.

Derivation in software perspectives can be interpreted in a couple of different ways. You can use derivation to indicate the difference between a calculated value and a stored value. In this case, we would interpret Figure 5.5 as indicating that the start and end are stored but that the length is computed. Although this is a common use, I'm not so keen, because it reveals too much of the internals of `DateRange`.

My preferred thinking is that it indicates a constraint between values. In this case, we are saying that the constraint among the three values holds, but it isn't important which of the three values is computed. In this case, the choice of which attribute to mark as derived is arbitrary and strictly unnecessary, but it's useful to help remind people of the constraint. This usage also makes sense with conceptual diagrams.

Derivation can also be applied to properties using association notation. In this case, you simply mark the name with a `/`.

Interfaces and Abstract Classes

An **abstract class** is a class that cannot be directly instantiated. Instead, you instantiate an instance of a subclass. Typically, an abstract class has one or more operations that are abstract. An **abstract operation** has no implementation; it is pure declaration so that clients can bind to the abstract class.

The most common way to indicate an abstract class or operation in the UML is to *italicize* the name. You can also make properties abstract, indicating an abstract property or accessor methods. Italics are tricky to do on a whiteboard, so you can use the label: `{abstract}`.

An interface is a class that has no implementation; that is, all its features are abstract. Interfaces correspond directly to interfaces in C# and Java and are a

common idiom in other typed languages. You mark an interface with the keyword «interface».

Classes have two kinds of relationships with interfaces: providing and requiring. A class **provides an interface** if it is substitutable for the interface. In Java and .NET, a class can do that by implementing the interface or implementing a subtype of the interface. In C++, you subclass the class that is the interface.

A class **requires an interface** if it needs an instance of that interface in order to work. Essentially, this is having a dependency on the interface.

Figure 5.6 shows these relationships in action, based on a few collection classes from Java. I might write an `Order` class that has a list of line items. Because I'm using a list, the `Order` class is dependent on the `List` interface. Let's assume that it uses the methods `equals`, `add`, and `get`. When the objects connect,

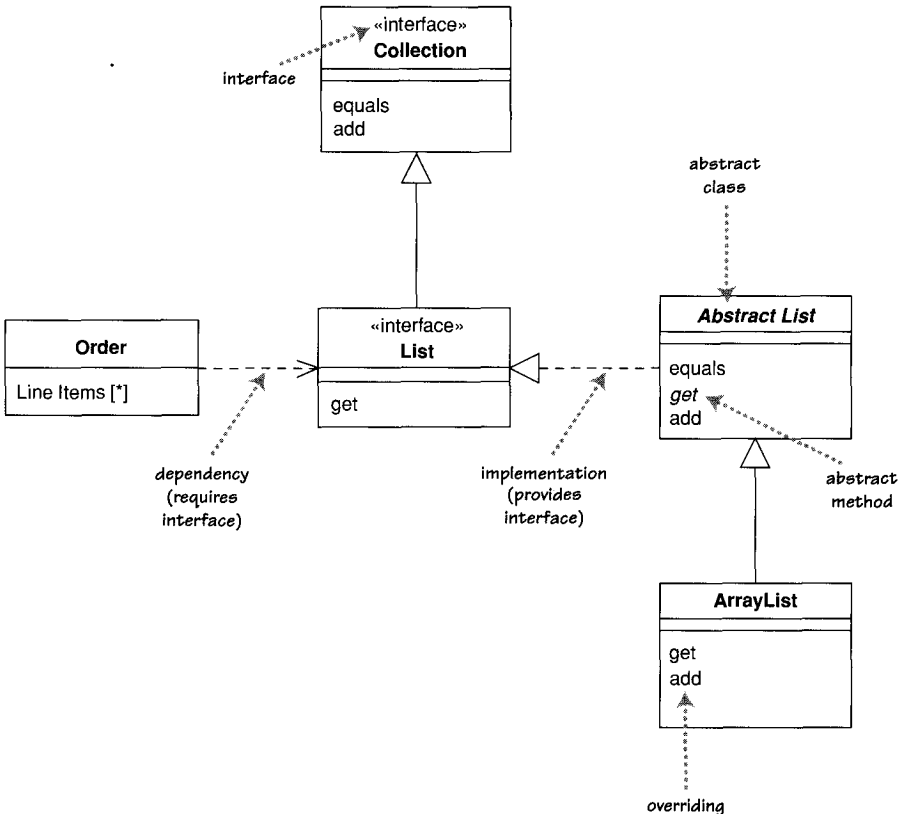


Figure 5.6 A Java example of interfaces and an abstract class

the `Order` will actually use an instance of `ArrayList` but need not know that in order to use those three methods, as they are all part of the `List` interface.

The `ArrayList` itself is a subclass of the `AbstractList` class. `AbstractList` provides some, but not all, the implementation of the `List` behavior. In particular, the `get` method is abstract. As a result, `ArrayList` implements `get` but also overrides some of the other operations on `AbstractList`. In this case, it overrides `add` but is happy to inherit the implementation of `equals`.

Why don't I simply avoid this and have `Order` use `ArrayList` directly? By using the interface, I allow myself the advantage of making it easier to change implementations later on if I need to. Another implementation may provide performance improvements, some database interaction features, or other benefits. By programming to the interface rather than to the implementation, I avoid having to change all the code should I need a different implementation of `List`. You should always try to program to an interface like this; always use the most general type you can.

I should also point out a pragmatic wrinkle in this. When programmers use a collection like this, they usually initialize the collection with a declaration, like this:

```
private List lineItems = new ArrayList();
```

Note that this strictly introduces a dependency from `Order` to the concrete `ArrayList`. In theory, this is a problem, but people don't worry about it in practice. Because the type of `lineItems` is declared as `List`, no other part of the `Order` class is dependent on `ArrayList`. Should we change the implementation, there's only this one line of initialization code that we need to worry about. It's quite common to refer to a concrete class once during creation but to use only the interface afterward.

The full notation of Figure 5.6 is one way to notate interfaces. Figure 5.7 shows a more compact notation. The fact that `ArrayList` implements `List` and `Collection` is shown by having ball icons, often referred to as lollipops, out of it. The fact that `Order` requires a `List` interface is shown by the socket icon. The connection is nicely obvious.

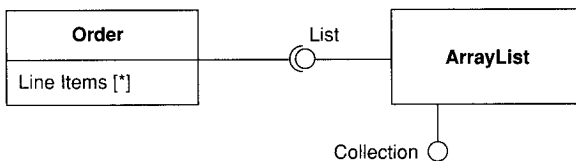


Figure 5.7 *Ball-and-socket notation*

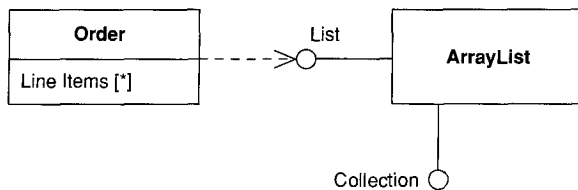


Figure 5.8 Older dependencies with lollipops

The UML has used the lollipop notation for a while, but the socket notation is new to UML 2. (I think it's my favorite notational addition.) You'll probably see older diagrams use the style of Figure 5.8, where a dependency stands in for the socket notation.

Any class is a mix of an interface and an implementation. Therefore, we may often see an object used through the interface of one of its superclasses. Strictly, it wouldn't be legal to use the lollipop notation for a superclass, as the superclass is a class, not a pure interface. But I bend these rules for clarity.

As well as on class diagrams, people have found lollipops useful elsewhere. One of the perennial problems with interaction diagrams is that they don't provide a very good visualization for polymorphic behavior. Although it's not normative usage, you can indicate this along the lines of Figure 5.9. Here, we can see that, although we have an instance of Salesman, which is used as such by the Bonus Calculator, the Pay Period object uses the Salesman only through its Employee interface. (You can do the same trick with communication diagrams.)

Read-Only and Frozen

On page 37, I described the `{readOnly}` keyword. You use this keyword to mark a property that can only be read by clients and that cannot be updated. Similar yet different is the `{frozen}` keyword from UML 1. A property is **frozen** if it cannot change during the lifetime of an object; such properties are often called immutable. Although it was dropped from UML 2, `{frozen}` is a very useful concept, so I would continue to use it. As well as marking individual properties as frozen, you can apply the keyword to a class to indicate that all properties of all instances are frozen. (I have heard that frozen may well be reinstated shortly.)

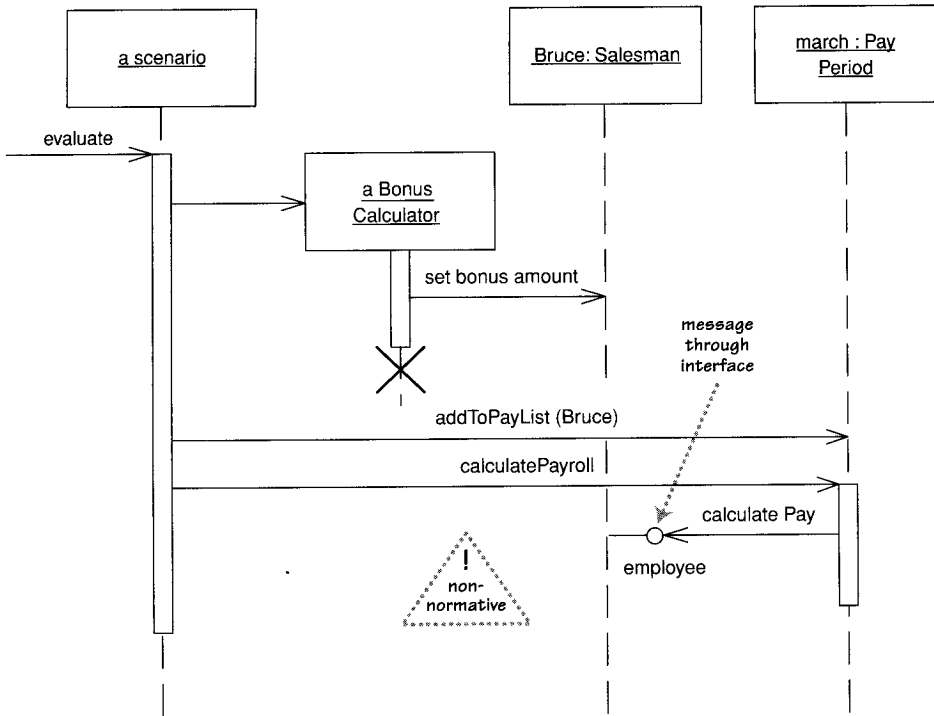


Figure 5.9 Using a lollipop to show polymorphism in a sequence diagram

Reference Objects and Value Objects

One of the common things said about objects is that they have identity. This is true, but it is not quite as simple as that. In practice, you find that identity is important for reference objects but not so important for value objects.

Reference objects are such things as Customer. Here, identity is very important because you usually want only one software object to designate a customer in the real world. Any object that references a Customer object will do so through a reference, or pointer; all objects that reference this Customer will reference the same software object. That way, changes to a Customer are available to all users of the Customer.

If you have two references to a Customer and wish to see whether they are the same, you usually compare their identities. Copies may be disallowed; if

they are allowed, they tend to be made rarely, perhaps for archive purposes or for replication across a network. If copies are made, you need to sort out how to synchronize changes.

Value objects are such things as Date. You often have multiple value objects representing the same object in the real world. For example, it is normal to have hundreds of objects that designate 1-Jan-04. These are all interchangeable copies. New dates are created and destroyed frequently.

If you have two dates and wish to see whether they are the same, you don't look at their identities but rather at the values they represent. This usually means that you have to write an equality test operator, which for dates would make a test on year, month, and day—or whatever the internal representation is. Each object that references 1-Jan-04 usually has its own dedicated object, but you can also share dates.

Value objects should be immutable; in other words, you should not be able to take a date object of 1-Jan-04 and change the same date object to be 2-Jan-04. Instead, you should create a new 2-Jan-04 object and use that instead. The reason is that if the date were shared, you would update another object's date in an unpredictable way, a problem referred to as **aliasing**.

In days gone by, the difference between reference objects and value objects was clearer. Value objects were the built-in values of the type system. Now you can extend the type system with your own classes, so this issue requires more thought.

The UML uses the concept of **data type**, which is shown as a keyword on the class symbol. Strictly, data type isn't the same as value object, as data types can't have identity. Value objects may have an identity, but don't use it for equality. Primitives in Java would be data types, but dates would not, although they would be value objects.

If it's important to highlight them, I use composition when associating with a value object. You can also use a keyword on a value type; common conventional ones I see are «value» or «struct».

Qualified Associations

A **qualified association** is the UML equivalent of a programming concept variously known as associative arrays, maps, hashes, and dictionaries. Figure 5.10 shows a way that uses a qualifier to represent the association between the Order and Order Line classes. The qualifier says that in connection with an Order, there may be one Order Line for each instance of Product.



Figure 5.10 *Qualified association*

From a software perspective, this qualified association would imply an interface along the lines of

```

class Order ...
  public OrderLine getLineItem(Product aProduct);
  public void addLineItem(Number amount, Product forProduct);
  
```

Thus, all access to a given Order Line requires a Product as an argument, suggesting an implementation using a key and value data structure.

It's common for people to get confused about the multiplicities of a qualified association. In Figure 5.10, an Order may have many Line Items, but the multiplicity of the qualified association is the multiplicity in the context of the qualifier. So the diagram says that an Order has 0..1 Line Items per Product. A multiplicity of 1 would indicate that Order would have to have a Line Item for every instance of Product. A * would indicate that you would have multiple Line Items per Product but that access to the Line Items is indexed by Product.

In conceptual modeling, I use the qualifier construct only to show constraints along the lines of “single Order Line per Product on Order.”

Classification and Generalization

I often hear people talk about subtyping as the *is a* relationship. I urge you to beware of that way of thinking. The problem is that the phrase *is a* can mean different things.

Consider the following phrases.

1. Shep is a Border Collie.
2. A Border Collie is a Dog.
3. Dogs are Animals.
4. A Border Collie is a Breed.
5. Dog is a Species.

Now try combining the phrases. If I combine phrases 1 and 2, I get “Shep is a Dog”; 2 and 3 taken together yield “Border Collies are Animals.” And 1 plus 2 plus 3 gives me “Shep is an Animal.” So far, so good. Now try 1 and 4: “Shep is a Breed.” The combination of 2 and 5 is “A Border Collie is a Species.” These are not so good.

Why can I combine some of these phrases and not others? The reason is that some are **classification**—the object Shep is an instance of the type Border Collie—and some are **generalization**—the type Border Collie is a subtype of the type Dog. Generalization is transitive; classification is not. I can combine a classification followed by a generalization but not vice versa.

I make this point to get you to be wary of *is a*. Using it can lead to inappropriate use of subclassing and confused responsibilities. Better tests for subtyping in this case would be the phrases “Dogs are kinds of Animals” and “Every instance of a Border Collie is an instance of a Dog.”

The UML uses the generalization symbol to show generalization. If you need to show classification, use a dependency with the «instantiate» keyword.

Multiple and Dynamic Classification

Classification refers to the relationship between an object and its type. Mainstream programming languages assume that an object belongs to a single class. But there are more options to classification than that.

In **single classification**, an object belongs to a single type, which may inherit from supertypes. In **multiple classification**, an object may be described by several types that are not necessarily connected by inheritance.

Multiple classification is different from multiple inheritance. Multiple inheritance says that a type may have many supertypes but that a single type must be defined for each object. Multiple classification allows multiple types for an object without defining a specific type for the purpose.

For example, consider a person subtyped as either man or woman, doctor or nurse, patient or not (see Figure 5.11). Multiple classification allows an object to have any of these types assigned to it in any allowable combination, without the need for types to be defined for all the legal combinations.

If you use multiple classification, you need to be sure that you make it clear which combinations are legal. UML 2 does this by placing each generalization relationship into a **generalization set**. On the class diagram, you label the generalization arrowhead with the name of the generalization set, which in UML 1

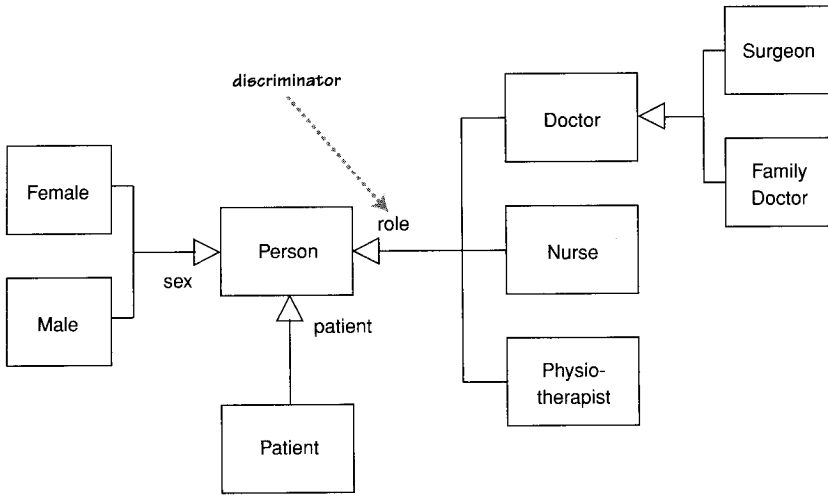


Figure 5.11 Multiple classification

was called the discriminator. Single classification corresponds to a single generalization set with no name.

Generalization sets are by default disjoint: Any instance of the supertype may be an instance of only one of the subtypes within that set. If you roll up generalizations into a single arrow, they must all be part of the same generalization set, as shown in Figure 5.11. Alternatively, you can have several arrows with the same text label.

To illustrate, note the following legal combinations of subtypes in the diagram: (Female, Patient, Nurse); (Male, Physiotherapist); (Female, Patient); and (Female, Doctor, Surgeon). The combination (Patient, Doctor, Nurse) is illegal because it contains two types from the role generalization set.

Another question is whether an object may change its class. For example, when a bank account is overdrawn, it substantially changes its behavior. Specifically, several operations, including “withdraw” and “close,” get overridden.

Dynamic classification allows objects to change class within the subtyping structure; **static classification** does not. With static classification, a separation is made between types and states; dynamic classification combines these notions.

Should you use multiple, dynamic classification? I believe that it is useful for conceptual modeling. For software perspectives, however, the distance between it and the implementations is too much of a leap. In the vast majority of UML

diagrams, you'll see only single static classification, so that should be your default.

Association Class

Association classes allow you to add attributes, operations, and other features to associations, as shown in Figure 5.12. We can see from the diagram that a person may attend many meetings. We need to keep information about how awake that person was; we can do this by adding the attribute *attentiveness* to the association.

Figure 5.13 shows another way to represent this information: Make *Attendance* a full class in its own right. Note how the multiplicities have moved.

What benefit do you gain with the association class to offset the extra notation you have to remember? The association class adds an extra constraint, in that there can be only one instance of the association class between any two participating objects. I feel the need for another example.

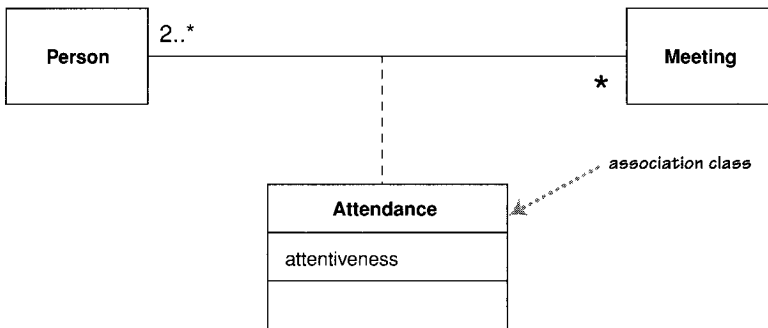


Figure 5.12 Association class

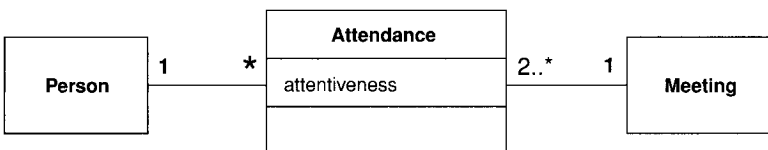


Figure 5.13 Promoting an association class to a full class

Take a look at the two diagrams in Figure 5.14. These diagrams have much the same form. However, we can imagine one Company playing different roles in the same Contract, but it's harder to imagine a Person having multiple competencies in the same skill; indeed, you would probably consider that an error.

In the UML, only the latter case is legal. You can have only one competency for each combination of Person and Skill. The top diagram in Figure 5.14 would not allow a Company to have more than one Role on a single contract. If you need to allow this, you need to make Role a full class, in the style of Figure 5.13.

Implementing association classes isn't terribly obvious. My advice is to implement an association class as if it were a full class but to provide methods that get information to the classes linked by the association class. So for Figure 5.12, I would see the following methods on Person:

```
class Person
  List getAttendances()
  List getMeetings()
```

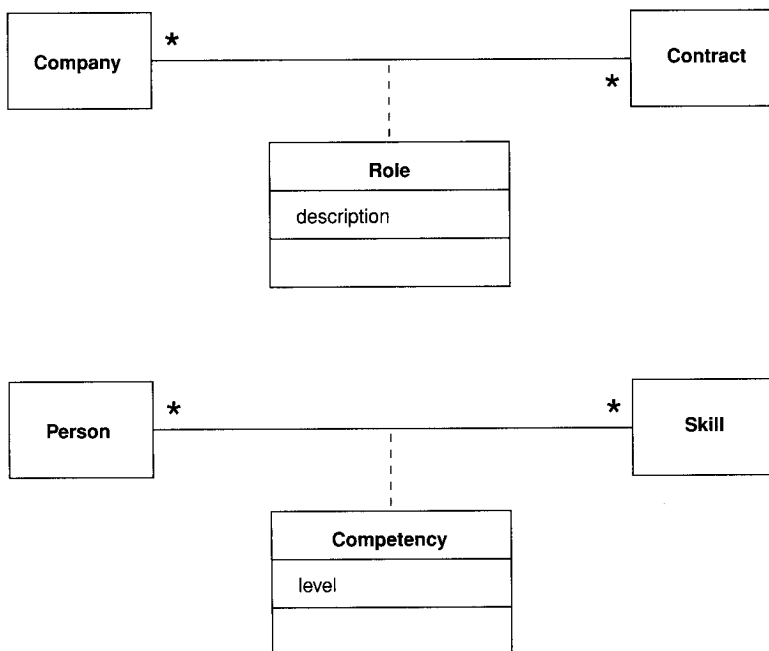


Figure 5.14 Association class subtleties (Role should probably not be an association class)

This way, a client of `Person` can get hold of the people at the meeting; if they want details, they can get the `Attendances` themselves. If you do this, remember to enforce the constraint that there can be only one `Attendance` object for any pair of `Person` and `Meeting`. You should place a check in whichever method creates the `Attendance`.

You often find this kind of construct with historical information, such as in Figure 5.15. However, I find that creating extra classes or association classes can make the model tricky to understand, as well as tilt the implementation in a particular direction that's often unsuitable.

If I have this kind of temporal information, I use a «temporal» keyword on the association (see Figure 5.16). The model indicates that a `Person` may work for only a single `Company` at one time. Over time, however, a `Person` may work for several `Companies`. This suggests an interface along the lines of:

```
class Person ...
  Company getEmployer();//get current employer
  Company getEmployer(Date);//get employer at a given date
  void changeEmployer(Company newEmployer,Date changeDate);
  void leaveEmployer (Date changeDate);
```

The «temporal» keyword is not part of the UML, but I mention it here for two reasons. First, it is a notion I have found useful on several occasions in my modeling career. Second, it shows how you can use keywords to extend the UML. You can read a lot more about this at <http://martinfowler.com/ap2/timeNarrative.html>.

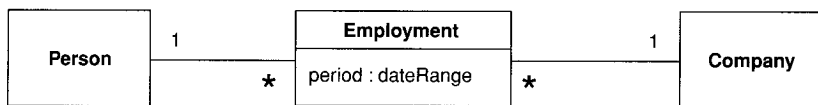


Figure 5.15 Using a class for a temporal relationship



Figure 5.16 «Temporal» keyword for associations

Template (Parameterized) Class

Several languages, most noticeably C++, have the notion of a **parameterized class**, or **template**. (Templates are on the list to be included in Java and C# in the near future.)

This concept is most obviously useful for working with collections in a strongly typed language. This way, you can define behavior for sets in general by defining a template class Set.

```
class Set <T> {
    void insert (T newElement);
    void remove (T anElement);
}
```

When you have done this, you can use the general definition to make Set classes for more specific elements:

```
Set <Employee> employeeSet;
```

You declare a template class in the UML by using the notation shown in Figure 5.17. The T in the diagram is a placeholder for the type parameter. (You may have more than one.)

A use of a parameterized class, such as Set<Employee>, is called a **derivation**. You can show a derivation in two ways. The first way mirrors the C++ syntax (see Figure 5.18). You describe the derivation expression within angle brackets in the form <parameter-name::parameter-value>. If there's only one parameter, conventional use often omits the parameter name. The alternative notation (see Figure 5.19) reinforces the link to the template and allows you to rename the bound element.

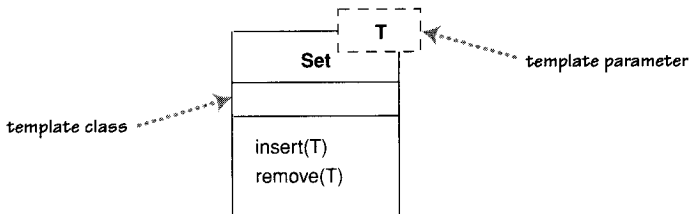


Figure 5.17 *Template class*

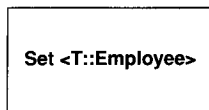


Figure 5.18 Bound element (version 1)

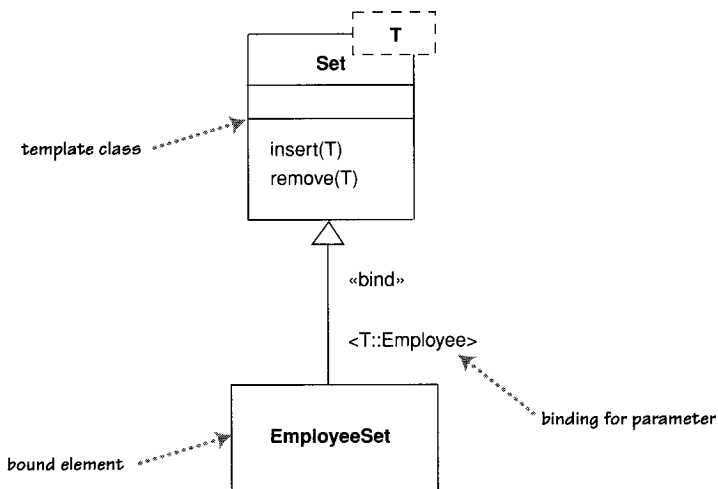


Figure 5.19 Bound element (version 2)

The `«bind»` keyword is a stereotype on the refinement relationship. This relationship indicates that `EmployeeSet` will conform to the interface of `Set`. You can think of the `EmployeeSet` as a subtype of `Set`. This fits the other way of implementing type-specific collections, which is to declare all appropriate subtypes.

Using a derivation is *not* the same as subtyping, however. You are not allowed to add features to the bound element, which is completely specified by its template; you are adding only restricting type information. If you want to add features, you must create a subtype.

Enumerations

Enumerations (Figure 5.20) are used to show a fixed set of values that don't have any properties other than their symbolic value. They are shown as the class with the `«enumeration»` keyword.

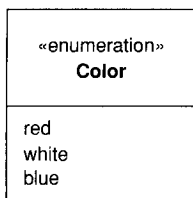


Figure 5.20 Enumeration

Active Class

An **active class** has instances, each of which executes and controls its own thread of control. Method invocations may execute in a client's thread or in the active object's thread. A good example of this is a command processor that accepts command objects from the outside and then executes the commands within its own thread of control.

The notation for active classes has changed from UML 1 to UML 2, as shown in Figure 5.21. In UML 2, an active class has extra vertical lines on the side; in UML 1, it had a thick border and was called an active object.

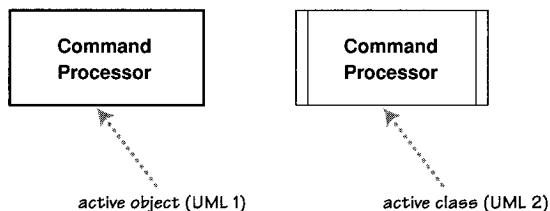


Figure 5.21 Active class

Visibility

Visibility is a subject that is simple in principle but has complex subtleties. The simple idea is that any class has public and private elements. Public elements can be used by any other class; private elements can be used only by the owning class. However, each language makes its own rules. Although many languages use such terms as *public*, *private*, and *protected*, they mean different things in