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.

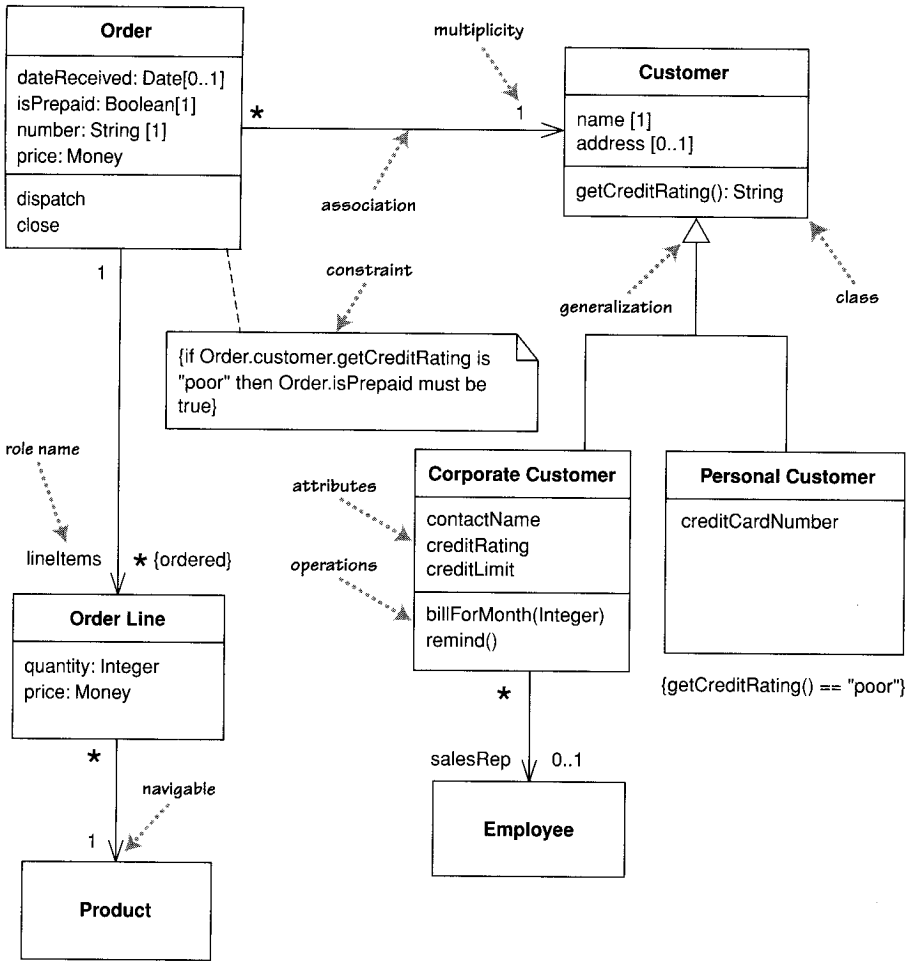

Figure 3.1 *A simple class diagram*

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

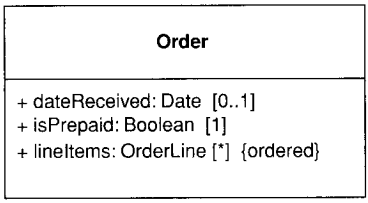

Figure 3.2 *Showing properties of an order as attributes*

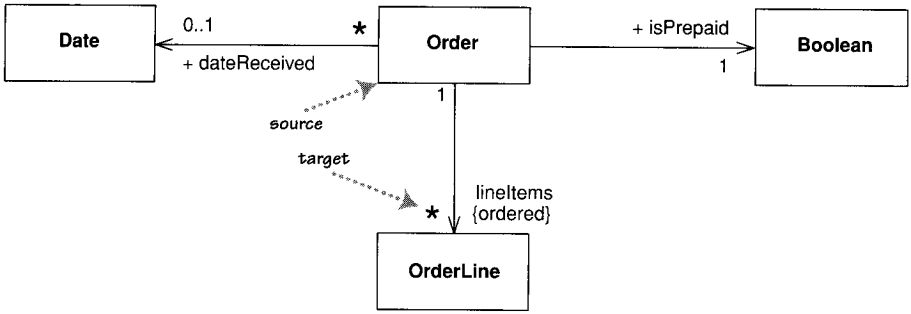

Figure 3.3 *Showing properties of an order as associations*

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

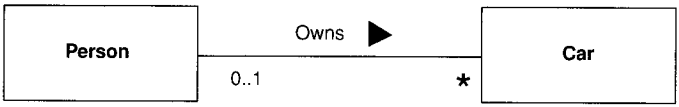

Figure 3.5 *Using a verb phrase to name an association*

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: --.

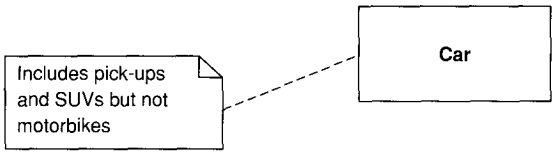

Figure 3.6 *A note is used as a comment on one or more diagram elements*

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.

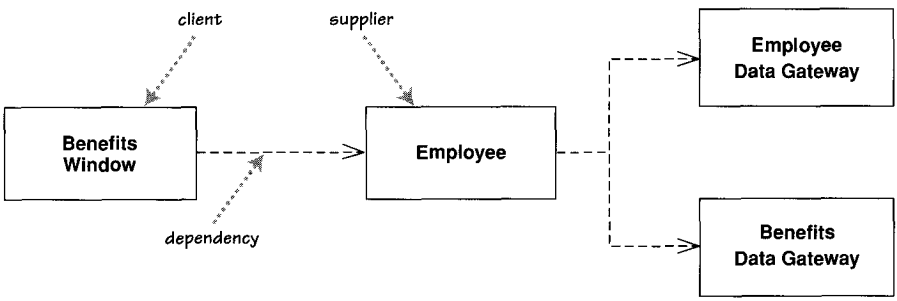

Figure 3.7 *Example dependencies*

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.

Your general rule should be to minimize dependencies, particularly when they cross large areas of a system. In particular, you should be wary of cycles, as they can lead to a cycle of changes. I'm not super strict on this. I don't mind mutual dependencies between closely related classes, but I do try to eliminate cycles at a broader level, particularly between packages.

Trying to show all the dependencies in a class diagram is an exercise in futility; there are too many and they change too much. Be selective and show dependencies only when they are directly relevant to the particular topic that you want to communicate. To understand and control dependencies, you are best off using them with package diagrams (pages 89).

The most common case I use for dependencies with classes is when illustrating a transient relationship, such as when one object is passed to another as a parameter. You may see these used with keywords «parameter», «local», and «global». You may also see these keywords on associations in UML 1 models, in which case they indicate transient links, not properties. These keywords are not part of UML 2.

Dependencies can be determined by looking at code, so tools are ideal for doing dependency analysis. Getting a tool to reverse engineer pictures of dependencies is the most useful way to use this bit of the UML.

Constraint Rules

Much of what you are doing in drawing a class diagram is indicating constraints. Figure 3.1 indicates that an Order can be placed only by a single Customer. The diagram also implies that each Line Item is thought of separately: You say "40 brown widgets, 40 blue widgets, and 40 red widgets," not "120 things" on the Order. Further, the diagram says that Corporate Customers have credit limits but Personal Customers do not.

The basic constructs of association, attribute, and generalization do much to specify important constraints, but they cannot indicate every constraint. These constraints still need to be captured; the class diagram is a good place to do that.

The UML allows you to use anything to describe constraints. The only rule is that you put them inside braces ({}). You can use natural language, a programming language, or the UML's formal Object Constraint Language (OCL) [Warmer and Kleppe], which is based on predicate calculus. Using a formal notation avoids the risk of misinterpretation due to an ambiguous natural language. However, it introduces the risk of misinterpretation due to writers and

readers not really understanding OCL. So unless you have readers who are comfortable with predicate calculus, I'd suggest using natural language.

Optionally, you can name a constraint by putting the name first, followed by a colon; for example, {disallow incest: husband and wife must not be siblings}.

Design by Contract

Design by Contract is a design technique developed by Bertrand Meyer [Meyer]. The technique is a central feature of the Eiffel language he developed. Design by Contract is not specific to Eiffel, however; it is a valuable technique that can be used with any programming language.

At the heart of Design by Contract is the assertion. An **assertion** is a Boolean statement that should never be false and, therefore, will be false only because of a bug. Typically, assertions are checked only during debug and are not checked during production execution. Indeed, a program should never assume that assertions are being checked.

Design by Contract uses three particular kinds of assertions: post-conditions, pre-conditions, and invariants. Pre-conditions and post-conditions apply to operations. A **post-condition** is a statement of what the world should look like after execution of an operation. For instance, if we define the operation "square root" on a number, the post-condition would take the form *input = result * result*, where *result* is the output and *input* is the input value. The post-condition is a useful way of saying what we do without saying how we do it—in other words, of separating interface from implementation.

A **pre-condition** is a statement of how we expect the world to be before we execute an operation. We might define a pre-condition for the "square root" operation of *input > = 0*. Such a pre-condition says that it is an error to invoke "square root" on a negative number and that the consequences of doing so are undefined.

On first glance, this seems a bad idea, because we should put some check somewhere to ensure that "square root" is invoked properly. The important question is who is responsible for doing so.

The pre-condition makes it explicit that the caller is responsible for checking. Without this explicit statement of responsibilities, we can get either too little checking—because both parties assume that the other is responsible—or too much—both parties check. Too much checking is a bad thing because it leads to a lot of duplicate checking code, which can

significantly increase the complexity of a program. Being explicit about who is responsible helps to reduce this complexity. The danger that the caller forgets to check is reduced by the fact that assertions are usually checked during debugging and testing.

From these definitions of pre-condition and post-condition, we can see a strong definition of the term **exception**. An exception occurs when an operation is invoked with its pre-condition satisfied yet cannot return with its post-condition satisfied.

An **invariant** is an assertion about a class. For instance, an Account class may have an invariant that says that *balance == sum(entries.amount())*. The invariant is "always" true for all instances of the class. Here, "always" means "whenever the object is available to have an operation invoked on it."

In essence, this means that the invariant is added to pre-conditions and post-conditions associated with all public operations of the given class. The invariant may become false during execution of a method, but it should be restored to true by the time any other object can do anything to the receiver.

Assertions can play a unique role in subclassing. One of the dangers of inheritance is that you could redefine a subclass's operations to be inconsistent with the superclass's operations. Assertions reduce the chances of this. The invariants and post-conditions of a class must apply to all subclasses. The subclasses can choose to strengthen these assertions but cannot weaken them. The pre-condition, on the other hand, cannot be strengthened but may be weakened.

This looks odd at first, but it is important to allow dynamic binding. You should always be able to treat a subclass object as if it were an instance of the superclass, per the principle of substitutability. If a subclass strengthened its pre-condition, a superclass operation could fail when applied to the subclass.

When to Use Class Diagrams

Class diagrams are the backbone of the UML, so you will find yourself using them all the time. This chapter covers the basic concepts; Chapter 5 discusses many of the advanced concepts.

The trouble with class diagrams is that they are so rich, they can be overwhelming to use. Here are a few tips.

- Don't try to use all the notations available to you. Start with the simple stuff in this chapter: classes, associations, attributes, generalization, and constraints. Introduce other notations from Chapter 5 only when you need them.
- I've found conceptual class diagrams very useful in exploring the language of a business. For this to work, you have to work hard on keeping software out of the discussion and keeping the notation very simple.
- Don't draw models for everything; instead, concentrate on the key areas. It is better to have a few diagrams that you use and keep up to date than to have many forgotten, obsolete models.

The biggest danger with class diagrams is that you can focus exclusively on structure and ignore behavior. Therefore, when drawing class diagrams to understand software, always do them in conjunction with some form of behavioral technique. If you're going well, you'll find yourself swapping between the techniques frequently.

Where to Find Out More

All the general UML books I mentioned in Chapter 1 talk about class diagrams in more detail. Dependency management is a critical feature of larger projects. The best book on this topic is [Martin].

Chapter 4

Sequence Diagrams

Interaction diagrams describe how groups of objects collaborate in some behavior. The UML defines several forms of interaction diagram, of which the most common is the sequence diagram.

Typically, a sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.

To begin the discussion, I'll consider a simple scenario. We have an order and are going to invoke a command on it to calculate its price. To do that, the order needs to look at all the line items on the order and determine their prices, which are based on the pricing rules of the order line's products. Having done that for all the line items, the order then needs to compute an overall discount, which is based on rules tied to the customer.

Figure 4.1 is a sequence diagram that shows one implementation of that scenario. Sequence diagrams show the interaction by showing each participant with a lifeline that runs vertically down the page and the ordering of messages by reading down the page.

One of the nice things about a sequence diagram is that I almost don't have to explain the notation. You can see that an instance of order sends `getQuantity` and `getProduct` messages to the order line. You can also see how we show the order invoking a method on itself and how that method sends `getDiscountInfo` to an instance of customer.

The diagram, however, doesn't show everything very well. The sequence of messages `getQuantity`, `getProduct`, `getPricingDetails`, and `calculateBasePrice` needs to be done for each order line on the order, while `calculateDiscounts` is invoked just once. You can't tell that from this diagram, although I'll introduce some more notation to handle that later.

Most of the time, you can think of the participants in an interaction diagram as objects, as indeed they were in UML 1. But in UML 2, their roles are much more complicated, and to explain it all fully is beyond this book. So I use the

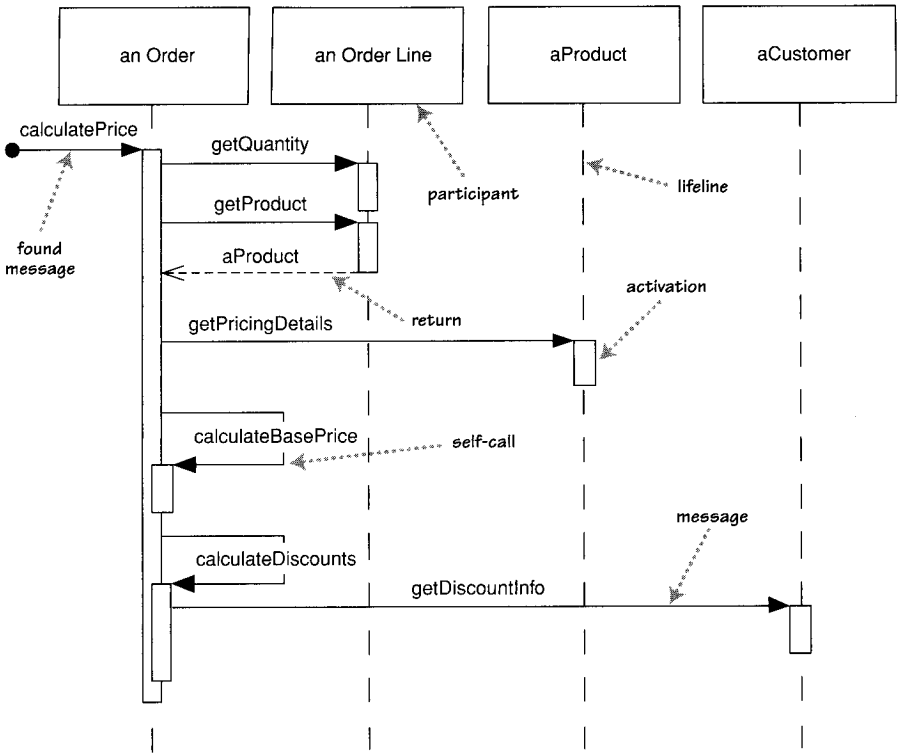

Figure 4.1 *A sequence diagram for centralized control*

term **participants**, a word that isn't used formally in the UML spec. In UML 1, participants were objects and so their names were underlined, but in UML 2, they should be shown without the underline, as I've done here.

In these diagrams, I've named the participants using the style `anOrder`. This works well most of the time. A fuller syntax is `name : Class`, where both the name and the class are optional, but you must keep the colon if you use the class. (Figure 4.4, shown on page 58, uses this style.)

Each lifeline has an activation bar that shows when the participant is active in the interaction. This corresponds to one of the participant's methods being on the stack. Activation bars are optional in UML, but I find them extremely valuable in clarifying the behavior. My one exception is when exploring a design during a design session, because they are awkward to draw on whiteboards.

Naming often is useful to correlate participants on the diagram. The call `getProduct` is shown returning `aProduct`, which is the same name, and therefore the

same participant, as the `aProduct` that the `getPricingDetails` call is sent to. Note that I've used a return arrow for only this call; I did that to show the correspondance. Some people use returns for all calls, but I prefer to use them only where they add information; otherwise, they simply clutter things. Even in this case, you could probably leave the return out without confusing your reader.

The first message doesn't have a participant that sent it, as it comes from an undetermined source. It's called a **found message**.

For another approach to this scenario, take a look at Figure 4.2. The basic problem is still the same, but the way in which the participants collaborate to implement it is very different. The Order asks each Order Line to calculate its own Price. The Order Line itself further hands off the calculation to the Product; note how we show the passing of a parameter. Similarly, to calculate the discount, the Order invokes a method on the Customer. Because it needs information from the Order to do this, the Customer makes a reentrant call (`getBaseValue`) to the Order to get the data.

The first thing to note about these two diagrams is how clearly the sequence diagram indicates the differences in how the participants interact. This is the great strength of interaction diagrams. They aren't good at showing details of algorithms, such as loops and conditional behavior, but they make the calls between participants crystal clear and give a really good picture about which participants are doing which processing.

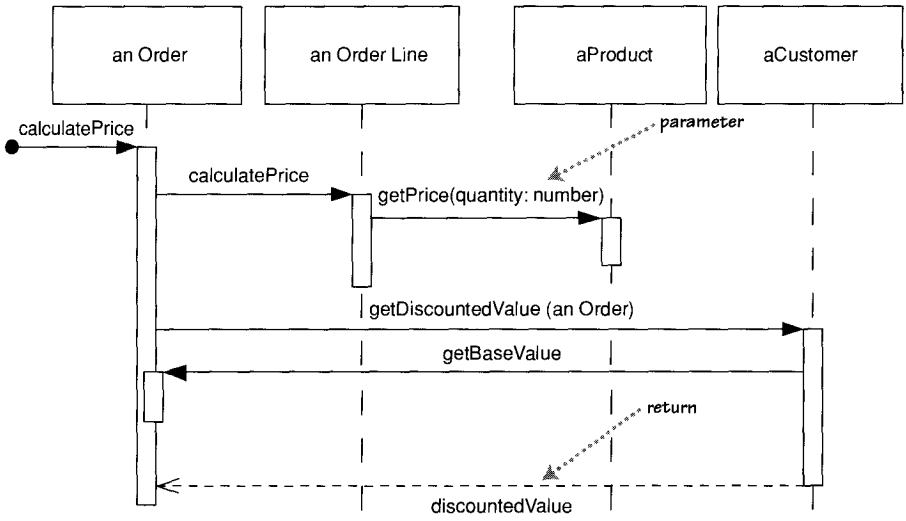

Figure 4.2 *A sequence diagram for distributed control*

The second thing to note is the clear difference in styles between the two interactions. Figure 4.1 is **centralized control**, with one participant pretty much doing all the processing and other participants there to supply data. Figure 4.2 uses **distributed control**, in which the processing is split among many participants, each one doing a little bit of the algorithm.

Both styles have their strengths and weaknesses. Most people, particularly those new to objects, are more used to centralized control. In many ways, it's simpler, as all the processing is in one place; with distributed control, in contrast, you have the sensation of chasing around the objects, trying to find the program.

Despite this, object bigots like me strongly prefer distributed control. One of the main goals of good design is to localize the effects of change. Data and behavior that accesses that data often change together. So putting the data and the behavior that uses it together in one place is the first rule of object-oriented design.

Furthermore, by distributing control, you create more opportunities for using polymorphism rather than using conditional logic. If the algorithms for product pricing are different for different types of product, the distributed control mechanism allows us to use subclasses of product to handle these variations.

In general the OO style is to use a lot of little objects with a lot of little methods that give us a lot of plug points for overriding and variation. This style is very confusing to people used to long procedures; indeed, this change is the heart of the **paradigm shift** of object orientation. It's something that's very difficult to teach. It seems that the only way to really understand it is to work in an OO environment with strongly distributed control for a while. Many people then say that they get a sudden "aha" when the style makes sense. At this point, their brains have been rewired, and they start thinking that decentralized control is actually easier.

Creating and Deleting Participants

Sequence diagrams show some extra notation for creating and deleting participants (Figure 4.3). To create a participant, you draw the message arrow directly into the participant box. A message name is optional here if you are using a constructor, but I usually mark it with "new" in any case. If the participant immediately does something once it's created, such as the query command, you start an activation right after the participant box.

Deletion of a participant is indicated by big X. A message arrow going into the X indicates one participant explicitly deleting another; an X at the end of a lifeline shows a participant deleting itself.

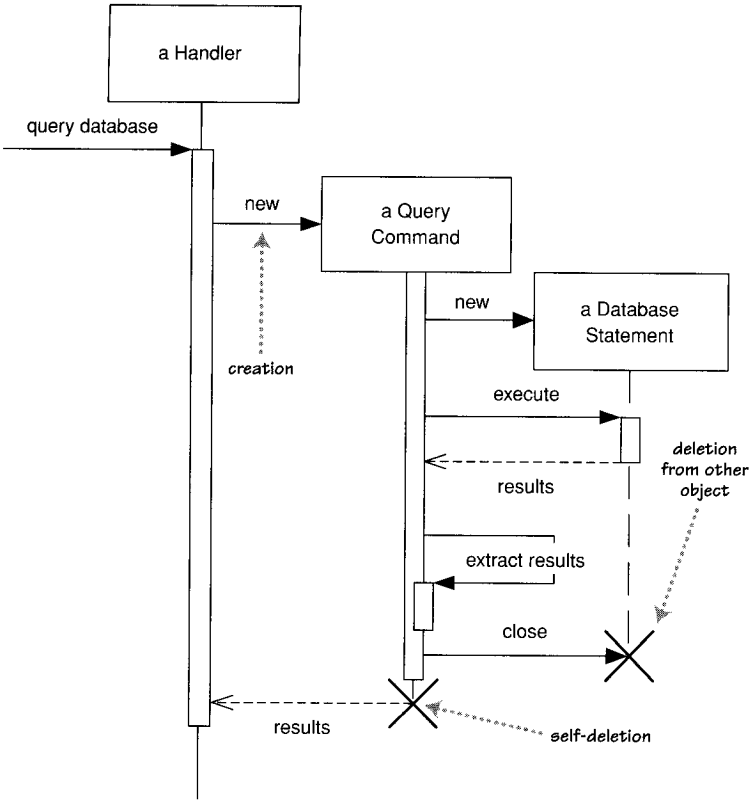

Figure 4.3 *Creation and deletion of participants*

In a garbage-collected environment, you don't delete objects directly, but it's still worth using the X to indicate when an object is no longer needed and is ready to be collected. It's also appropriate for close operations, indicating that the object isn't usable any more.

Loops, Conditionals, and the Like

A common issue with sequence diagrams is how to show looping and conditional behavior. The first thing to point out is that this isn't what sequence diagrams are good at. If you want to show control structures like this, you are better off with an activity diagram or indeed with code itself. Treat sequence

diagrams as a visualization of how objects interact rather than as a way of modeling control logic.

That said, here's the notation to use. Both loops and conditionals use **interaction frames,** which are ways of marking off a piece of a sequence diagram. Figure 4.4 shows a simple algorithm based on the following pseudocode:

```
procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confirm
end procedure
```

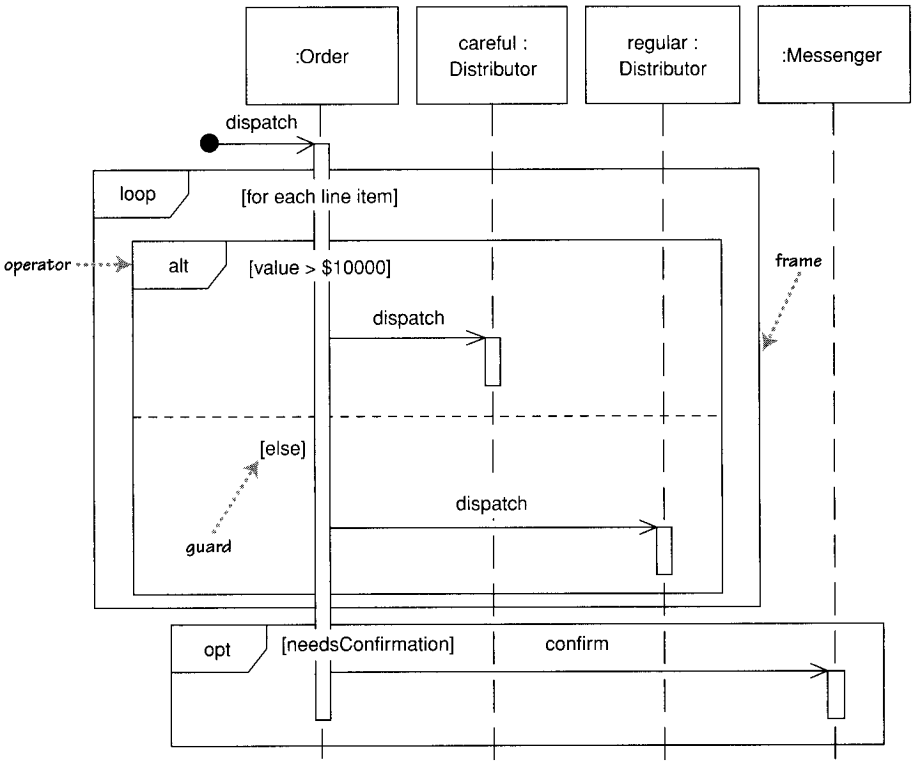

Figure 4.4 *Interaction frames*

In general, frames consist of some region of a sequence diagram that is divided into one or more fragments. Each frame has an operator and each fragment may have a guard. (Table 4.1 lists common operators for interaction frames.) To show a loop, you use the `loop` operand with a single fragment and put the basis of the iteration in the guard. For conditional logic, you can use an `alt` operator and put a condition on each fragment. Only the fragment whose guard is true will execute. If you have only one region, there is an `opt` operator.

Interaction frames are new in UML 2. As a result, you may see diagrams prepared before UML 2 and that use a different approach; also, some people don't like the frames and prefer some of the older conventions. Figure 4.5 shows some of these unofficial tweaks.

UML 1 used iteration markers and guards. An **iteration marker** is a * added to the message name. You can add some text in square brackets to indicate the basis of the iteration. **Guards** are a conditional expression placed in square brackets and indicate that the message is sent only if the guard is true. While these notations have been dropped from sequence diagrams in UML 2, they are still legal on communication diagrams.

Although iteration markers and guards can help, they do have weaknesses. The guards can't indicate that a set of guards are mutually exclusive, such as the

Table 4.1 *Common Operators for Interaction Frames*

Operator	Meaning
`alt`	Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4).
`opt`	Optional; the fragment executes only if the supplied condition is true. Equivalent to an `alt` with only one trace (Figure 4.4).
`par`	Parallel; each fragment is run in parallel.
`loop`	Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration (Figure 4.4).
`region`	Critical region; the fragment can have only one thread executing it at once.
`neg`	Negative; the fragment shows an invalid interaction.
`ref`	Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
`sd`	Sequence diagram; used to surround an entire sequence diagram, if you wish.

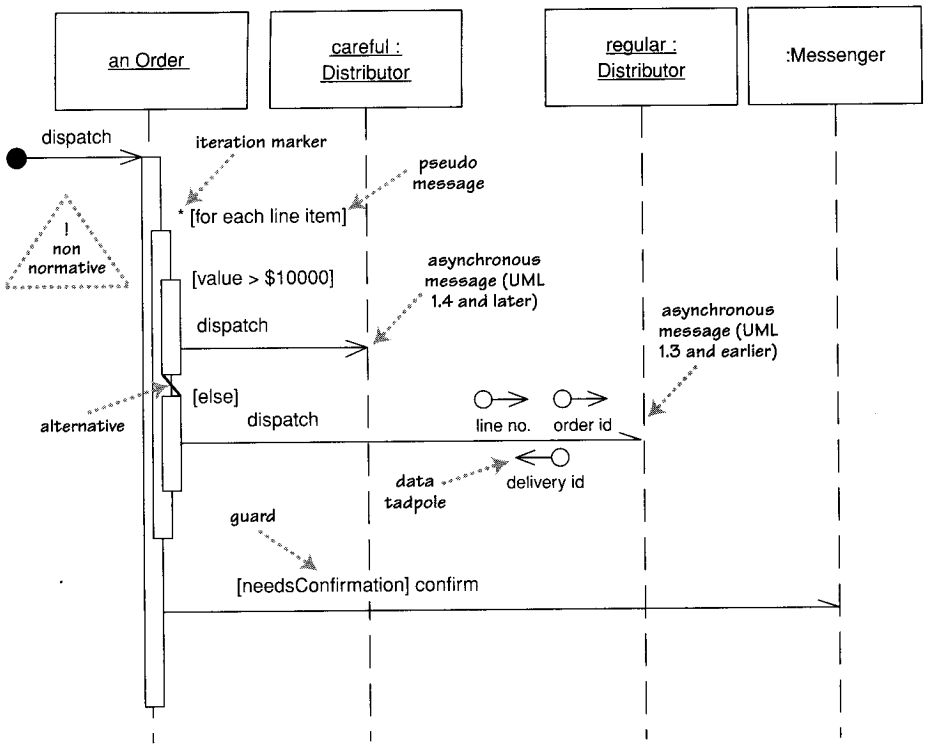

Figure 4.5 *Older conventions for control logic*

two on Figure 4.5. Both notations work only with a single message send and don't work well when several messages coming out of a single activation are within the same loop or conditional block.

To get around this last problem, an unofficial convention that's become popular is to use a **pseudomessage**, with the loop condition or the guard on a variation of the self-call notation. In Figure 4.5, I've shown this without a message arrow; some people include a message arrow, but leaving it out helps reinforce that this isn't a real call. Some also like to gray shade the pseudomessage's activation bar. If you have alterative behavior, you can show that with an alternative marker between the activations.

Although I find activations very helpful, they don't add much in the case of the `dispatch` method, whereby you send a message and nothing else happens within the receiver's activation. A common convention that I've shown on Figure 4.5 is to drop the activation for those simple calls.

The UML standard has no graphic device to show passing data; instead, it's shown by parameters in the message name and return arrows. **Data tadpoles** have been around in many methods to indicate the movement of data, and many people still like to use them with the UML.

All in all, although various schemes can add notation for conditional logic to sequence diagrams, I don't find that they work any better than code or at least pseudocode. In particular, I find the interaction frames very heavy, obscuring the main point of the diagram, so I prefer pseudomessages.

Synchronous and Asynchronous Calls

If you're exceptionally alert, you'll have noticed that the arrowheads in the last couple of diagrams are different from the arrowheads earlier on. That minor difference is quite important in UML 2. In UML 2, filled arrowheads show a synchronous message, while stick arrowheads show an asynchronous message.

If a caller sends a **synchronous message**, it must wait until the message is done, such as invoking a subroutine. If a caller sends an **asynchronous message**, it can continue processing and doesn't have to wait for a response. You see asynchronous calls in multithreaded applications and in message-oriented middleware. Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to debug.

The arrowhead difference is very subtle; indeed, rather too subtle. It's also a backward-incompatible change introduced in UML 1.4, before then an asynchronous message was shown with the half-stick arrowhead, as in Figure 4.5.

I think that this arrowhead distinction is too subtle. If you want to highlight asynchronous messages, I would recommend using the obsolete half-stick arrowhead, which draws the eye much better to an important distinction. If you're reading a sequence diagram, beware of making assumptions about synchrony from the arrowheads unless you're sure that the author is intentionally making the distinction.

When to Use Sequence Diagrams

You should use sequence diagrams when you want to look at the behavior of several objects within a single use case. Sequence diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.

If you want to look at the behavior of a single object across many use cases, use a state diagram (see Chapter 10). If you want to look at behavior across many use cases or many threads, consider an activity diagram (see Chapter 11).

If you want to explore multiple alternative interactions quickly, you may be better off with CRC cards, as that avoids a lot of drawing and erasing. It's often handy to have a CRC card session to explore design alternatives and then use sequence diagrams to capture any interactions that you want to refer to later.

Other useful forms of interaction diagrams are communication diagrams, for showing connections; and timing diagrams, for showing timing constraints.

CRC Cards

One of the most valuable techniques in coming up with a good OO design is to explore object interactions, because it focuses on behavior rather than data. CRC (Class-Responsibility-Collaboration) diagrams, invented by Ward Cunningham in the late 1980s, have stood the test of time as a highly effective way to do this (Figure 4.6). Although they aren't part of the UML, they are a very popular technique among skilled object designers.

To use CRC cards, you and your colleagues gather around a table. Take various scenarios and act them out with the cards, picking them up in the air when they are active and moving them to suggest how they send messages to each other and pass them around. This technique is almost impossible to describe in a book yet is easily demonstrated; the best way to learn it is to have someone who has done it show it to you.

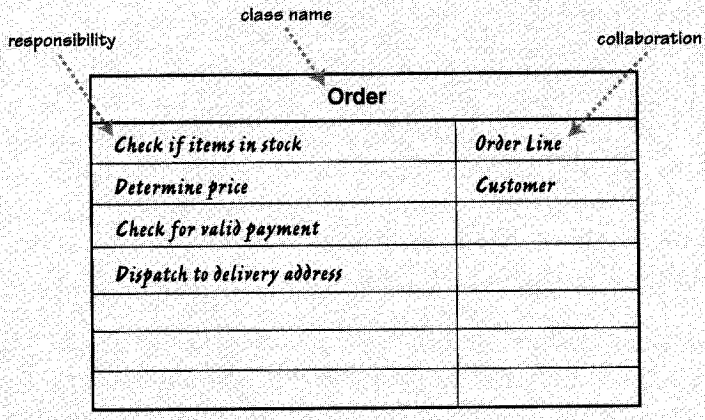

Figure 4.6 *A sample CRC card*

An important part of CRC thinking is identifying responsibilities. A **responsibility** is a short sentence that summarizes something that an object should do: an action the object performs, some knowledge the object maintains, or some important decisions the object makes. The idea is that you should be able to take any class and summarize it with a handful of responsibilities. Doing that can help you think more clearly about the design of your classes.

The second C refers to **collaborators**: the other classes that this class needs to work with. This gives you some idea of the links between classes—still at a high level.

One of the chief benefits of CRC cards is that they encourage animated discussion among the developers. When you are working through a use case to see how classes will implement it, the interaction diagrams in this chapter can be slow to draw. Usually, you need to consider alternatives; with diagrams, the alternatives can take too long to draw and rub out. With CRC cards, you model the interaction by picking up the cards and moving them around. This allows you to quickly consider alternatives.

As you do this, you form ideas about responsibilities and write them on the cards. Thinking about responsibilities is important, because it gets you away from the notion of classes as dumb data holders and eases the team members toward understanding the higher-level behavior of each class. A responsibility may correspond to an operation, to an attribute, or, more likely, to an undetermined clump of attributes and operations.

A common mistake I see people make is generating long lists of low-level responsibilities. But doing so misses the point. The responsibilities should easily fit on one card. Ask yourself whether the class should be split or whether the responsibilities would be better stated by rolling them up into higher-level statements.

Many people stress the importance of role playing, whereby each person on the team plays the role of one or more classes. I've never seen Ward Cunningham do that, and I find that role playing gets in the way.

Books have been written on CRC, but I've found that they never really get to the heart of the technique. The original paper on CRC, written with Kent Beck, is [Beck and Cunningham]. To learn more about both CRC cards and responsibilities in design, take a look at [Wirfs-Brock].

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).

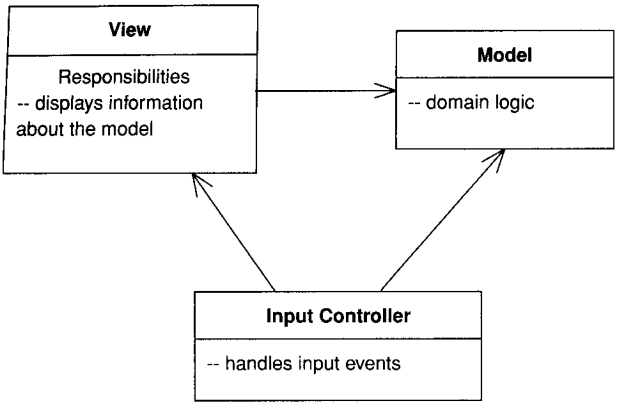

Figure 5.1 *Showing responsibilities in a class diagram*

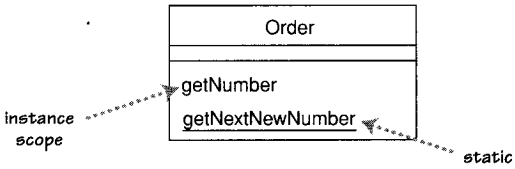

Figure 5.2 *Static notation*

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].

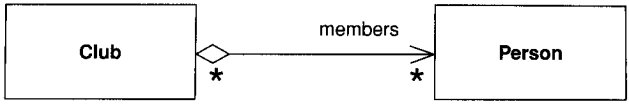

Figure 5.3 *Aggregation*

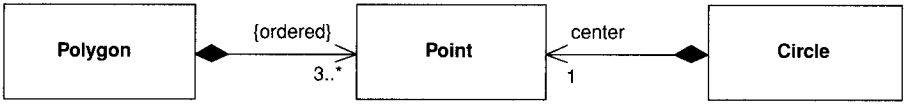

Figure 5.4 *Composition*

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,

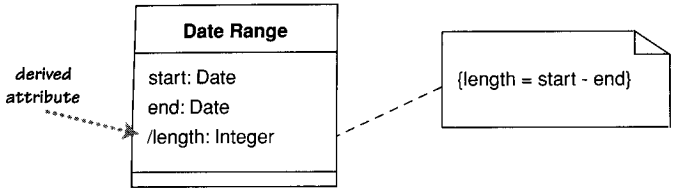

Figure 5.5 *Derived attribute in a time period*

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 whiteboards, 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,

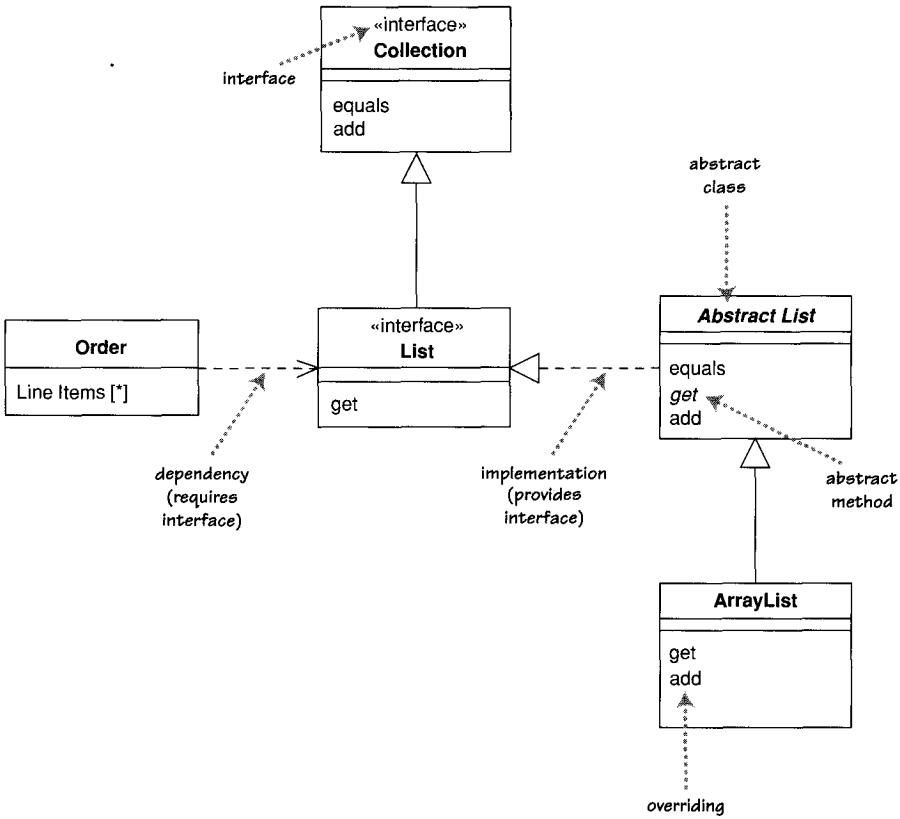

Figure 5.6 *A Java example of interfaces and an abstract class*

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.

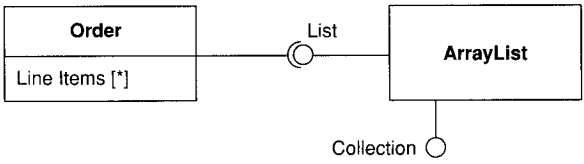

Figure 5.7 *Ball-and-socket notation*

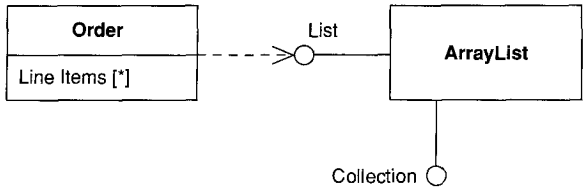

Figure 5.8 *Older dependencies with lollipops*

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.)

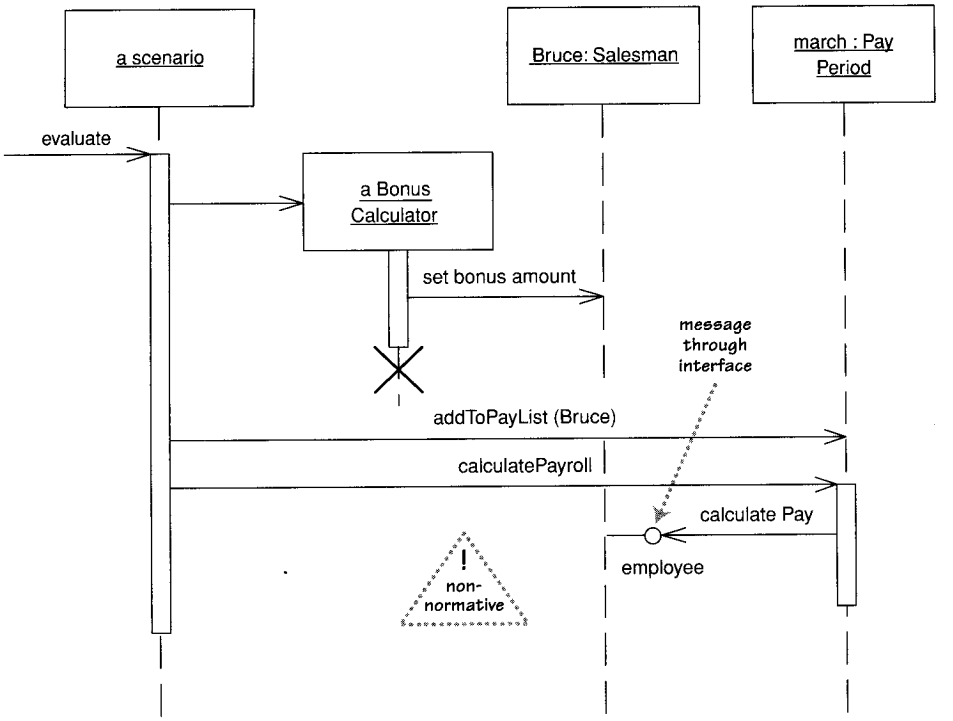

Figure 5.9 *Using a lollipop to show polymorphism in a sequence diagram*

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

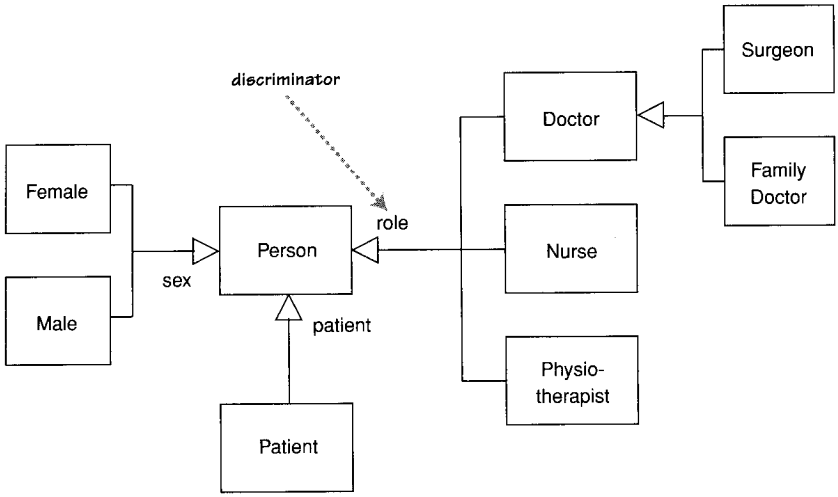

Figure 5.11 *Multiple classification*

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.

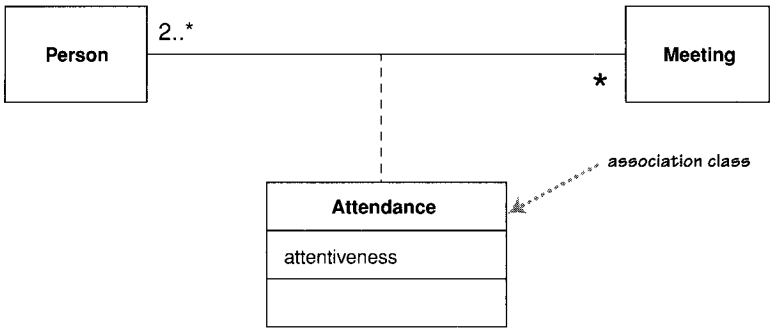

Figure 5.12 *Association class*

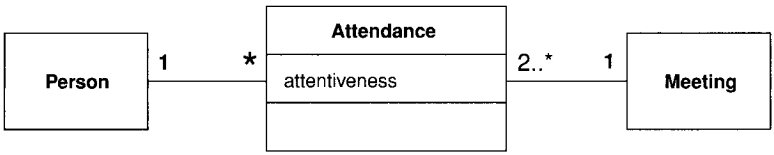

Figure 5.13 *Promoting an association class to a full class*

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 where 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()
```

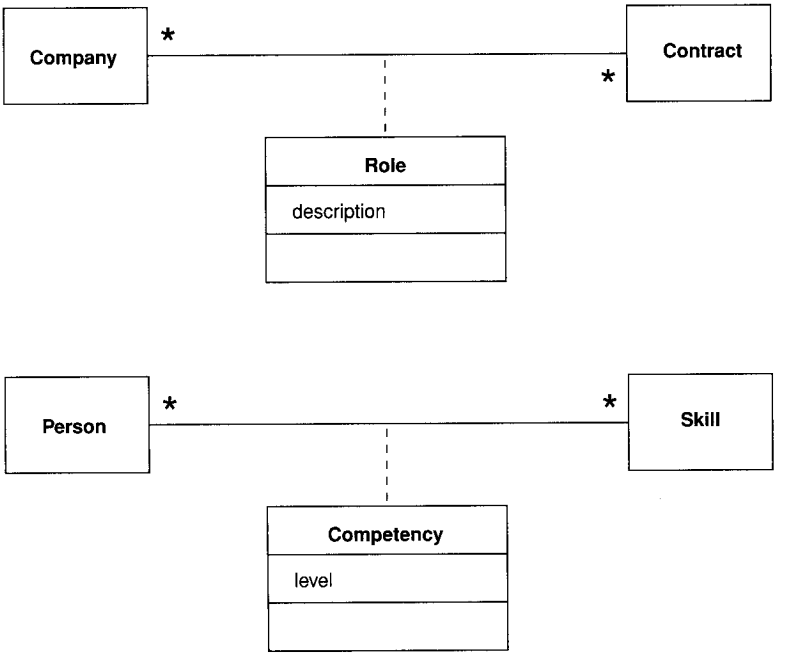

Figure 5.14 *Association class subtleties (Role should probably not be an association class)*

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.**

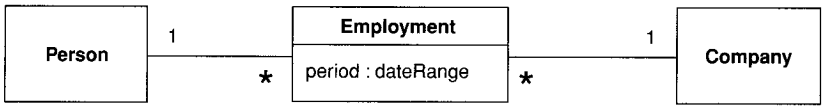

Figure 5.15 *Using a class for a temporal relationship*

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.

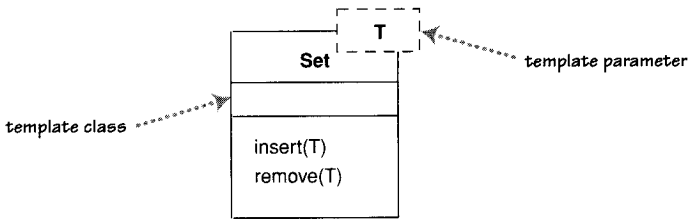

Figure 5.17 *Template class*

Figure 5.18 *Bound element (version 1)*

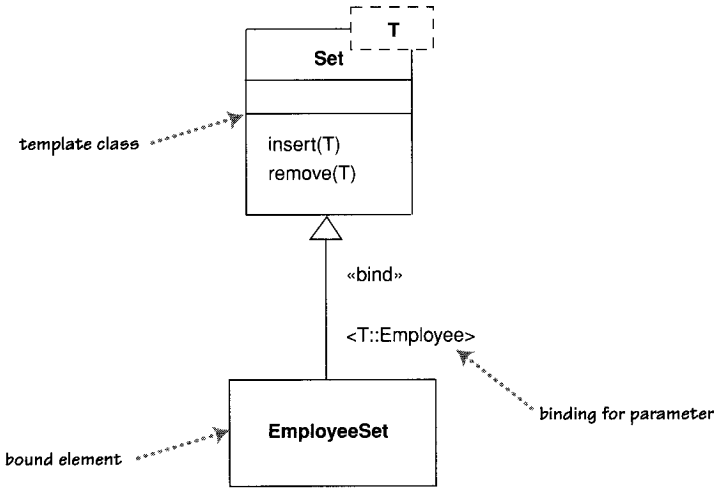

Figure 5.19 *Bound element (version 2)*

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.

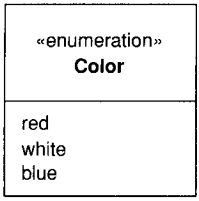

Figure 5.20 *Enumeration*

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.

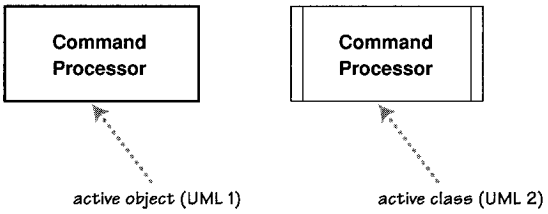

Figure 5.21 *Active class*

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

different languages. These differences are small, but they lead to confusion, especially for those of us who use more than one language.

The UML tries to address this without getting into a horrible tangle. Essentially, within the UML, you can tag any attribute or operation with a visibility indicator. You can use any marker you like, and its meaning is language dependent. However, the UML provides four abbreviations for visibility: + (public), – (private), ~ (package), and # (protected). These four levels are used within the UML meta-model and are defined within it, but their definitions vary subtly from those in other languages.

When you are using visibility, use the rules of the language in which you are working. When you are looking at a UML model from elsewhere, be wary of the meanings of the visibility markers, and be aware of how those meanings can change from language to language.

Most of the time, I don't draw visibility markers in diagrams; I use them only if I need to highlight the differences in visibility of certain features. Even then, I can mostly get away with + and –, which at least are easy to remember.

Messages

Standard UML does not show any information about message calls on class diagrams. However, I've sometimes seen conventional diagrams like Figure 5.22.

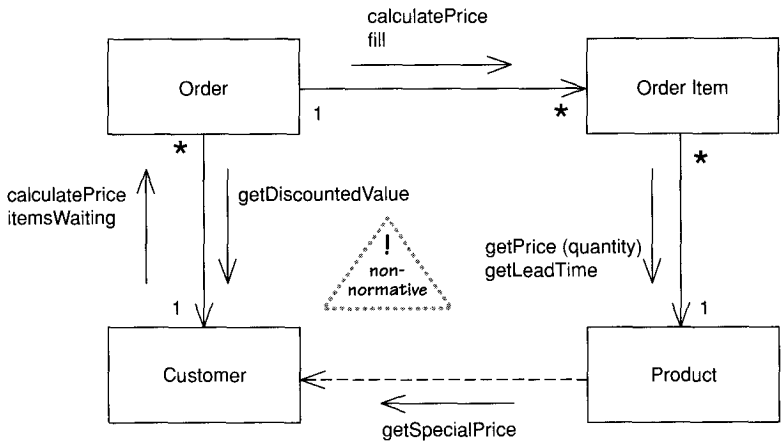

Figure 5.22 *Classes with messages*

These add arrows to the sides of associations. The arrows are labeled with the messages that one object sends to another. Because you don't need an association to a class to send a message to it, you may also need to add a dependency arrow to show messages between classes that aren't associated.

This message information spans multiple use cases, so they aren't numbered to show sequences, unlike communication diagrams.