8.1.3 Diagramming Collaborating Classes

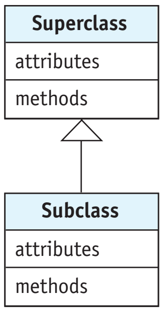
KEY IDEA

Class diagrams show the relationships between collaborating classes.

We have used class diagrams regularly to give an overview of an individual class. These diagrams can also be used to show the relationships between collaborating classes. In fact, we've already seen class diagrams showing such collaborating classes: when we extended one class to form a new one with additional capabilities (see Sections 2.2 and 3.5.3). In that situation, we generally place the superclass above the subclass and connect the two with a closed arrow pointing to the superclass. A generic example is shown in Figure 8-3.

(figure 8-3)

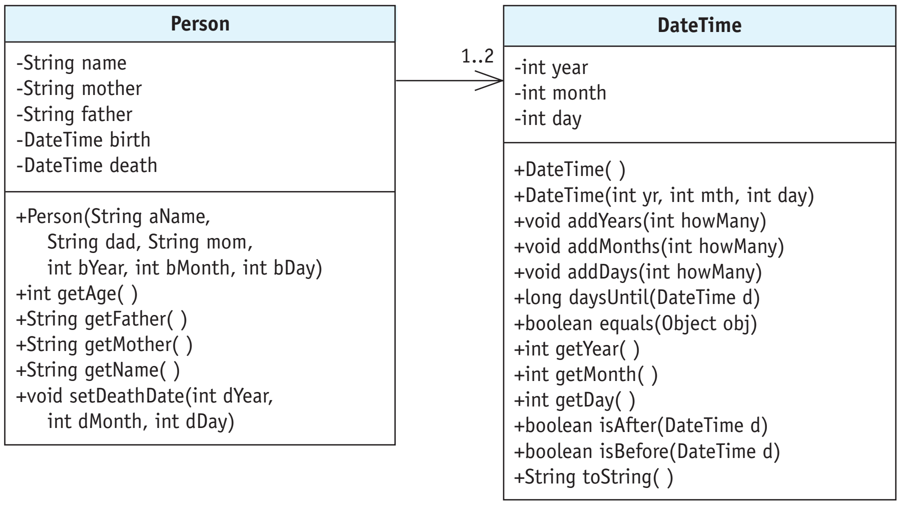
Class diagram showing two classes collaborating via inheritance



However, the `Person` class does not extend `DateTime` (nor is the reverse true), and so we use a different diagramming convention. This convention uses an open-headed arrow from one class to the other. The tail of the arrow is the class containing the instance variable and the head of the arrow is the class representing the variable's type. Usually the classes are drawn side by side, if possible. A class diagram for the `Person` class in Figure 8-4 serves as an example.

(figure 8-4)

Class diagram for the `Person` *class showing its collaboration with* `DateTime`


Person
-String name
-String mother
-String father
-DateTime birth
-DateTime death
+Person(String aName, String dad, String mom, int bYear, int bMonth, int bDay)
+int getAge( )
+String getFather( )
+String getMother( )
+String getName( )
+void setDeathDate(int dYear, int dMonth, int dDay)
1..2
DateTime
-int year
-int month
-int day
+DateTime( )
+DateTime(int yr, int mth, int day)
+void addYears(int howMany)
+void addMonths(int howMany)
+void addDays(int howMany)
+long daysUntil(DateTime d)
+boolean equals(Object obj)
+int getYear( )
+int getMonth( )
+int getDay( )
+boolean isAfter(DateTime d)
+boolean isBefore(DateTime d)
+String toString( )

Another feature of the diagram is the **multiplicity** near the arrowhead. The `1..2` in the diagram shows that each `Person` object uses at least one but no more than two `DateTime` objects. A class diagram will show each class only once, no matter how many objects are actually created using the classes. In general, the first number is the minimum number of objects that will be used, and the second number is the maximum number that will be used in the running program.

PATTERN

Has-a (Composition)

Other multiplicities are common. `1` is an abbreviation for `1..1` and means that exactly one object is used. An asterisk (`*`) is used to mean "many." An asterisk by itself is an abbreviation for `0..*` meaning "anywhere from none to many." If there will always be at least one but possibly many, use `1..*`. An arrow without an explicit multiplicity is assumed to be `1`.

The inheritance relationship, as shown in Figure 8-3, never includes a multiplicity.

Clients and Servers

In Section 1.1.2, we briefly discussed the terms **client** and **server**. Here we see those roles depicted graphically. The arrow goes from the client to the server. The client, `Person`, requests a service such as finding the days until another date. The server, `DateTime`, is the class or object that performs the service.

"Is-a" versus "Has-a"

How do you know which diagramming convention to use? If you already have the Java code, you examine the code. If the code says `public class X extends Y`, use the "**is-a**" relationship shown in Figure 8-3. If the class has an instance variable referring to an object, use the "**has-a**" relationship shown in Figure 8-4.

"Is-a" comes from the sentence "An `X` is a kind of `Y`." For example, "a `Harvester` robot is a kind of `Robot`" (see Listing 3-3) or "a `Lamp` is a kind of `Thing`" (see Listing 2-6). Other examples include "a `Circle` is a kind of `Shape`," "an `Employee` is a kind of `Person`," and "an `Automobile` is a kind of `Vehicle`." Given two classes, if a sentence like any one of these makes sense, then using `extends` and a diagram like Figure 8-3 is often the right thing to do.

LOOKING AHEAD

We'll examine is-a relationships more carefully in Chapter 12.

On the other hand, it's more often the case that "an `X` has a `Y`." In that case, we use the "has-a" relationship, also called **composition**. "A `Person` has a `birth date`" or "a `GasPump` has a `Meter`" or "an `Automobile` has an `Engine`." Has-a relationships are implemented by adding an instance variable in the class that "has" something and is diagrammed similar to Figure 8-4.