Chapter 3. Class and Object Diagrams

This chapter focuses on class and object diagrams, which depict the structure of a system in general and at a particular point in time, respectively. First, I introduce class and object diagrams and how they are used. Next, I discuss classes, objects, and their details for modeling the elements that make up a system. Then, I go over associations, links, and their details for modeling the relationships among the elements that make up a system. Finally, I discuss various other types of elements and relationships. Many details that were not fleshed out in Chapter 2 are more fully elaborated here, and throughout the chapter I include suggestions relating to class and object diagrams.

Class modeling is a specialized type of modeling concerned with the general structure of a system. Object modeling is a specialized type of modeling concerned with the structure of a system at a particular point in time. You usually apply class and object modeling during analysis and design activities to understand the requirements and determine how a system will satisfy its requirements. Object modeling is usually used in conjunction with class modeling to explore and refine class diagrams. Class and object modeling usually start after the requirements have matured enough (as determined by your system development process) and continue in parallel with interaction and collaboration modeling (Chapter 6) throughout the system development process, while focusing on the elements that make up the system and their relationships.

As an architecture-centric process focuses on the architecture of a system across iterations, it is important to understand what elements make up a system and how they are related to one another. Given that every project has limited resources, you can use this information to determine how best to develop a system. This allows architects, designers, and developers to consider technical trade-offs concerning the system, including which elements can be developed in parallel, which elements can be purchased rather than built, and which elements can be reused.

Classes and Objects

Class diagrams show classes that represent concepts, while object diagrams show objects that represent specific instances of those concepts. The next few sections talk in detail about the representation of classes and objects in class and object diagrams.

Classes

As discussed in Chapter 2, a *class* is a general concept. For example, the project management system involves various general concepts, including projects, managers, teams, work products, requirements, and systems.

A class defines a type of object and its characteristics, including structural features and behavioral features. *Structural features* define what objects of the class know, and *behavioral features* define what objects of the class can do. For example, in Chapter 2 you saw that individuals of the `Manager` class have names (something they know), and can initiate and

terminate projects (things they do). Structural features include attributes and associations. Behavioral features include operations and methods.

The most crucial aspect of a class is that it has semantics: some agreed upon meaning between whomever is communicating about it. For example, when I discuss a project, what does a project mean to my audience? Is it an effort that lasts one week or one year? Is it an effort that requires a manager and other human resources? And so forth. Such meaning is very specific to the audience and domain in which the class is used.

In a UML class diagram, a class is shown as a solid-outline rectangle with three standard compartments separated by horizontal lines. The required top compartment shows the class name, the optional second compartment shows a list of attributes, and the optional third compartment shows a list of operations. The second and third compartments need only show the specific information you want to communicate using a given diagram. You don't need to show all of a class's attributes and operations all the time.

Figure 3-1 shows various fundamental classes associated with the project management system in our case study, including `Worker`, `UnitOfWork`, and `WorkProduct`, using the most basic notation for classes.

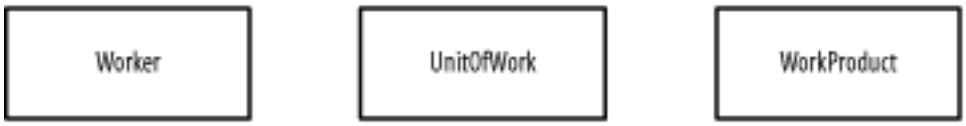

Figure 3-1. Classes

A worker is a person or group of people who perform work, including project managers, resource managers, human resources, and system administrators. A unit of work is a unit of effort, which includes capturing and analyzing requirements, as well as designing, implementing, testing, or deploying a system. A work product is anything produced and used within a project, including the requirements and the system. Notice that a class should be named using a noun phrase. Classes, as you should recall from Chapter 2, represent concepts that you discover by focusing on nouns.

Attributes

An *attribute* is what an object of a class knows. It's an element of data maintained by the object. For example, each object of the `Worker` class of the project management system may have a name, description, and so forth. These are all attributes.

In a class diagram, you list attributes in the second compartment for a class. The simplest approach is to just list attribute names, but the UML allows you to do much more than that. Consider an attribute for holding a worker's email address. You may start by defining it using the following basic syntax:

```
EmailAddress
```

As you go through the development process, you can add detail to this definition in each iteration by asking various questions and capturing more detail about the attribute based upon the answers.

For example, you may ask how many email addresses a worker has. Presuming that a worker may have up to five email addresses, you can update the attribute definition to the following:

```
EmailAddress [1..5]
```

Next, you may ask if these email addresses are ordered, perhaps by priority. Presuming that email addresses are not ordered, you can update the attribute definition as follows:

```
EmailAddress [1..5 unordered]
```

You may decide to ask the type of data an email address attribute needs to hold. You discover that an email address is a string of characters, and you update the attribute definition to indicate that:

```
EmailAddress [1..5 unordered] : String
```

You might then ask if there should be a default value for a worker's email address. Your client suggests using a default value of "`No email address`", so you update the attribute definition to the following:

```
EmailAddress [1..5 unordered] : String = "No email address"
```

Finally, you may ask whether other objects are allowed to access a `Worker` object's email address. Presuming the answer is that a `Worker` object's email address is not accessible by other objects, you can update the attribute definition one last time by preceding it with minus sign (`-`), as follows:

```
- EmailAddress [1..5 unordered] : String = "No email address"
```

To summarize, this final attribute definition communicates the following information:

The - symbol

> Indicates that the email address attribute is private to an object and thus inaccessible by other objects.

`1..5`

> Indicates that the email address attribute may have from one to five values.

`unordered`

Indicates that the email address attribute values are not ordered based on any specific criteria.

`String`

Indicates that email addresses are strings of characters.

`"No email address"`

Is the initial value of each email address.

Figure 3-2 shows the `Worker` class from Figure 3-1 with its attributes. A worker has an identification number (`ID`), a next identification number (`NextID`), a name, up to five email addresses that are unordered, any number of phone numbers that are ordered, and a password. `NextID` is underlined to indicate that it is one value, defined at the class level, shared by all objects of the class. The system uses this shared value to ensure that every `Worker` object has a unique ID number.

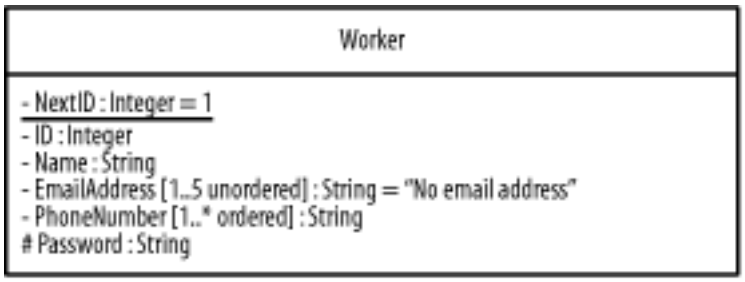

Figure 3-2. Worker class with its attributes

In Figure 3-2, you'll see some syntax I haven't described yet: a number sign (`#`) at the front of the `Password` attribute and the keyword `ordered` in the `PhoneNumber` attribute. This syntax is described in the next section, Section 3.1.1.2.

Figure 3-3 shows the `UnitOfWork` class of Figure 3-1 with its attributes. A unit of work has a name and description.

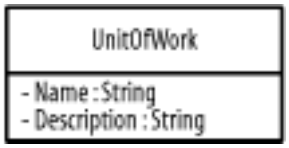

Figure 3-3. UnitOfWork class with its attributes

Figure 3-4 shows the `WorkProduct` class of Figure 3-1 with its attributes. A work product has a name, description, and a percentage of completion.

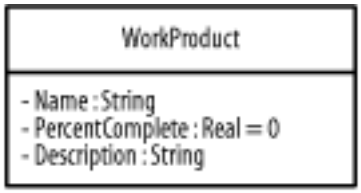

Figure 3-4. WorkProduct class with its attributes

Attribute syntax

In the UML, an attribute is described in a class's second compartment expressed using the following UML syntax:

```
visibility name [multiplicity ordering] : type = initial_value
```

in which:

visibility

Is optional, has no default value, and indicates whether the attribute is accessible from outside the class. It may be one of the following:

+

Public visibility; the attribute is accessible from outside its class.

-

Private visibility; the attribute is inaccessible from outside its class.

#

Protected visibility; the attribute is accessible by classes that have a generalization relationship (as discussed in Chapter 2) to its class, but is otherwise inaccessible from outside its class.

Figure 3-2 through Figure 3-4 show that most attributes are private, except a worker's password is protected so that more specific types of workers may use it in whatever manner in which they handle security.

name

Is the name of the attribute you are describing.

multiplicity

Is optional, has a default value of 1, and indicates the number of values an attribute may hold. If an attribute has only one value, the multiplicity, ordering, and square brackets are not shown. Otherwise, the multiplicity is shown as a `lower-bound .. upper-bound` string in which a single asterisk indicates an unlimited range; for example, `0..*` allows from zero up to an infinite number of values. Figure 3-2 through Figure 3-4 show that all attributes except for `EmailAddress` and `PhoneNumber` have one value only. A worker may have up to five email addresses and any number of phone numbers.

`ordering`

Is optional, has a default value of `unordered`, and is used where the multiplicity is greater than one to indicate whether the values of an attribute are ordered or unordered. Use one of the following keywords:

`unordered`

Indicates that the values are unordered.

`ordered`

Indicates that the values are ordered.

Figure 3-2 shows that a worker's five email addresses are unordered and that a worker's phone numbers are ordered.

`type`

Is optional, has no default value, and indicates the type of data an attribute may hold. If you don't show a type for an attribute, you should omit the colon. The type of an attribute may be another class. In addition, the UML provides the following data types:

`Boolean`

A true or false value

`Integer`

An integer number

`Real`

A real number

`String`

A sequence of characters

Figure 3-2 through Figure 3-4 show that most of the attributes are strings while a worker's identification number (`ID`) and next identification number (`NextID`) are integers, and a work product's `PercentComplete` attribute is a real number.

`initial_value`

Is optional, and indicates the initial value of an attribute. By default, an attribute has no initial value. If you do not show an initial value, you should omit the equal symbol (=). Figure 3-2 through Figure 3-4 show that most of the attributes have no initial value. However, a worker's next identification number (`NextID`) has an initial value of 1, a work product's percent complete has an initial value of 0, and email addresses have an initial value of `"No email address`".

If you prefer, the UML also allows you to show an attribute using pseudocode or another language. For example, you can use the syntax of Java, C++, C#, or some other programming language.

If an attribute's value is specific to an object, it is known as *instance scoped* or *object scoped*. If an attribute is shared by all objects of a class, it is known as *class scoped*. To indicate that an attribute is class scoped, underline it. Figure 3-2 through Figure 3-4 show that all the attributes are object scoped, except for the worker's next identification number (`NextID`), which is class scoped.

Operations

Recall from Chapter 2 that an operation is what an object of a class can do. It is a specification of a service provided by the object. Recall also that a method is how an object of a class does its processing. It is an implementation of a service provided by the object. For example, each class of the project management system may provide getter and setter operations for its attributes. These getter and setter operations retrieve and set the values for the attributes of a worker, unit of work, work product, and so forth.

Consider an operation for adding an email address to a worker. You may start by defining it using the following basic syntax:

```
addEmailAddress
```

As you go through iterations of the development process, you can add detail to this definition by asking questions and capturing additional detail about the operation from the answers to those questions.

For example, you may ask if the `addEmailaddress` operation requires any parameters: data that is input to or output from the operation. Presuming that the operation requires an email address as input, you can update the operation definition to the following:

```
addEmailAddress (theEmailAddress)
```

Next, you may ask what type of data may the email address hold? Presuming that the email address is a string of characters, you can update the operation definition to the following:

```
addEmailAddress (theEmailAddress : String)
```

Next, you may ask if there is a default value for the email address. Presuming that the email address has a default value of an empty string, or no characters, you can update the operation definition to the following:

```
addEmailAddress (theEmailAddress : String = "")
```

You might then ask whether the email address is simply an input to the operation, an output from the operation, or both. Presuming that the email address is only input to the operation, and thus not modified by the operation, you can add the `in` keyword preceding the parameter name:

```
addEmailAddress (in theEmailAddress : String = "")
```

You may then ask whether the operation returns any type of data. Presuming that the operation returns a Boolean true or false indicating whether the operation was successful in adding the email address to the worker, you can update the operation definition to the following:

```
addEmailAddress (in theEmailAddress : String = "") : Boolean
```

Finally, you may ask whether other objects are allowed to access an object's `addEmailAddress` operation. Presuming that an object's `addEmailAddress` operation is accessible by other objects, you can precede the operation name in the definition with a plus sign (+):

```
+ addEmailAddress (in theEmailAddress : String = "") : Boolean
```

To summarize, this operation definition communicates the following:

The + symbol

> Indicates that the `addEmailAddress` operation is public and is accessible by other objects. The expression inside the parentheses indicates the parameters that hold the values passed to the operation.

The `in` keyword

> Indicates that the parameter is input and may not be modified by the operation.

theEmailAddress : *String* = ""

> Indicates that an email address, which is a string with a default value of an empty string, is passed to the operation.

The `Boolean` keyword

> Indicates that the `addEmailAddress` operation returns a value of `true` or `false`, perhaps indicating whether there is room for the email address to be added to the email address attribute (which holds a maximum of five email addresses).

Figure 3-5 shows the `Worker` class from Figure 3-1 and Figure 3-2 with its various attribute getter and setter operations. Notice that the getter and setter operations for phone numbers are based on the priority of the phone number such that you specify the priority and set or get the corresponding phone number. Also, notice that the getter operation for email addresses retrieves all of a worker's email addresses, while the setter operation for email addresses simply adds one email address. The `create` and `destroy` operations create and destroy worker objects, respectively.

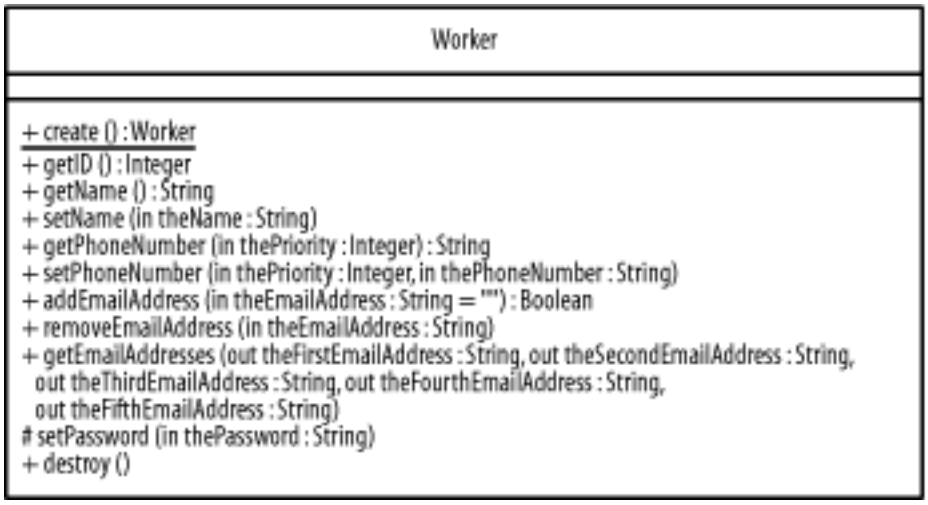

Figure 3-5. Worker class with its operations

Figure 3-6 shows the `UnitOfWork` class from Figure 3-1 and Figure 3-3 with its various getter and setter operations to set and retrieve attribute values, and with operations to create and destroy `UnitOfWork` objects.

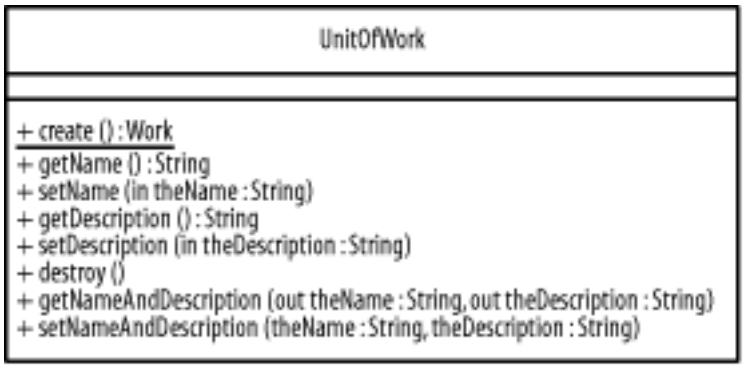

Figure 3-6. UnitOfWork class with its operations

Figure 3-7 shows the `WorkProduct` class from Figure 3-1 and Figure 3-4 with its various getter and setter operations for its attributes and operations to create and destroy work objects. Notice that Figure 3-7 also shows the attributes of the class.

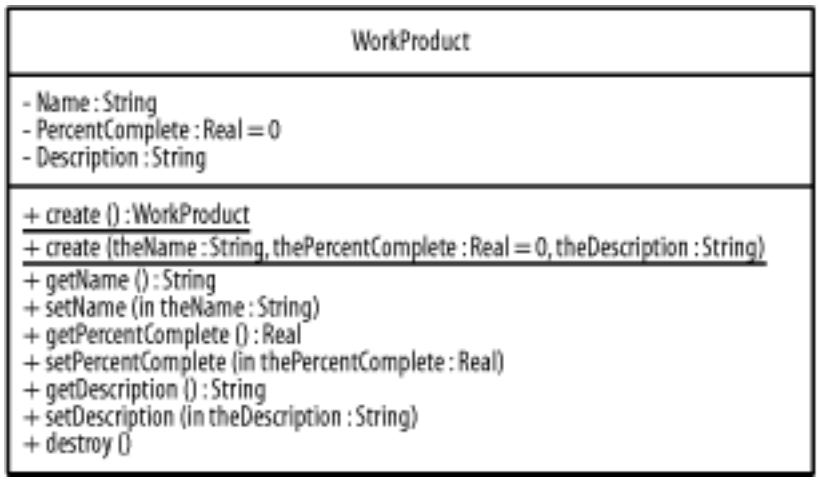

Figure 3-7. WorkProduct class with its operations

Operation syntax

In the UML, an operation is described in a class's third compartment using the following UML syntax:

```
visibility operation_name (parameter_list) : return_type
```

in which:

`visibility`

Is optional, has no default value, and indicates whether the operation is accessible from outside of the class.

It may be one of the following:

+

Public visibility; the operation is accessible from outside its class.

-

Private visibility; the operation is inaccessible from outside its class.

#

Protected visibility; the operation is accessible by classes that have a generalization relationship (as discussed in Chapter 2) to its class, but is otherwise inaccessible from outside its class.

Figure 3-5, Figure 3-6, and Figure 3-7 show that all the operations are public, except that the operation to set a worker's password is protected so that more specific types of workers or subclasses of the `Worker` class may use it in whatever manner in which they handle security.

`operation_name`

Is the name of the operation you are describing.

`parameter_list`

Is optional, has no default value, and is a comma-separated list indicating the parameters that hold the values passed to or received from the operation. Each parameter is shown as a text string having the following syntax:

`kind name : type = default_value`

`kind`

Is optional, has a default value of `in`, and may be one of the following:

`in`

Indicates the parameter is input-only, and may not be modified by the operation.

`out`

Indicates the parameter is output-only, and may be modified by the operation to communicate information to the client that invoked the operation.

`inout`

Indicates that the parameter is input and may in turn be modified by the operation to communicate information to the client that invoked the operation.

The type and default value are the same as for an attribute's type and initial value, described in the previous section.

`type`

Is optional, has no default value, and indicates the type of data a parameter may hold. If you don't show a type for a parameter, you should omit the colon. The type of a parameter may be another class. In addition, the UML provides the following data types:

`Boolean`

A true or false value.

`Integer`

An integer number.

`Real`

A real number.

`String`

A sequence of characters.

`default_value`

Is optional, and indicates the initial value of a parameter. By default, a parameter has no initial value. If you do not show an initial value, you should omit the equal symbol (`=`).

Figure 3-5 shows an initial value for the parameter to the `addEmailAddress` method.

`return_type`

Is optional, has no default value, and indicates the type of data the operation returns to its caller. If you choose not to show the return type of an operation, you should also omit the colon. Your choices for return type are the same as for a parameter type. Many of the operations shown in Figure 3-5 through Figure 3-7 show a return type.

If you prefer, the UML also allows you to show an operation using pseudocode or another language. For example, you can use the syntax of Java, C++, C#, or some other programming language.

If an operation applies to a specific object, it is known as *instance scoped* or *object scoped*. If an operation applies to the class itself, it is known as *class scoped*. Figure 3-5 through Figure 3-7 show that most of the operations are object scoped. The exceptions are the `create` operations, which are class scoped. The `create` operations are used to create objects of a class and are known as *constructors*. The `destroy` operations are used to destroy objects of a class and are known as *destructors*. The `create` operations are class scoped, because a class is used to create objects of the class; if `create` were instance scoped, you'd need to somehow create an object before invoking its `create` operation to create it, which makes no sense. The `destroy` operation, on the other hand, is object scoped, because it is applied to a specific object that is to be destroyed.

We can combine Figure 3-2 and Figure 3-5 or Figure 3-3 and Figure 3-6 much the way that Figure 3-7 combines the attributes shown in Figure 3-4 with the class's operations. In this way, we can show any combination of attributes and operations based upon what we want to communicate.

TIP

Methods, the actual implementations of operations, are not shown on a class, but may be described using other UML modeling techniques.

Objects

As discussed in Chapter 2, an *object* is a specific concept, or instance of a class, having the characteristics defined by its class, including structural features and behavioral features. For example, the project management system involves various specific concepts, including specific projects, managers, teams, work products, requirements, systems, and so forth. Recall that structural features define what the object knows, and that behavioral features define what the object can do. Structural features included attribute values and links. Behavioral features include operations and methods, which are shared by all the objects of a class. The most crucial aspect of an object is that it has its own identity. No two objects are the same, even if they have the same values for their structural features. For example, even if two worker objects have the same values for their attributes, the objects are unique and have their own identities.

In a UML object diagram, an object is shown as a solid-outline rectangle with two standard compartments separated by horizontal lines. The required top compartment shows the object name followed by a colon followed by the object's class name, and the entire string is fully underlined. Both names are optional, and the colon should only be present if the class name is specified. The optional second compartment shows a list of attributes. The second compartment need only show the specific information you want to communicate using a given diagram; you need not show all of an object's attribute values all the time.

Figure 3-8 shows various objects associated with the classes shown in the previous figures. These objects include the following:

- Nora and Phillip who are workers
- Testing that is a unit of work
- Test that is a work product
- An anonymous unit of work with the name attribute, `Implementation`
- An anonymous work product with the name attribute, `System`
- XYZ that is an unspecified object.

Figure 3-8 introduces notation you haven't seen before: some objects have no names and others have no class specified. Objects in a UML diagram that do not have names are referred to as *anonymous objects*. Objects for which a class has not been specified are referred to as *unspecified objects*. You may encounter such notation depending on the specific methodology being used for a project.

Note that the object names in Figure 3-8 are all fully underlined. You'll also see specific attribute values. I'll discuss attribute values further in the next section.

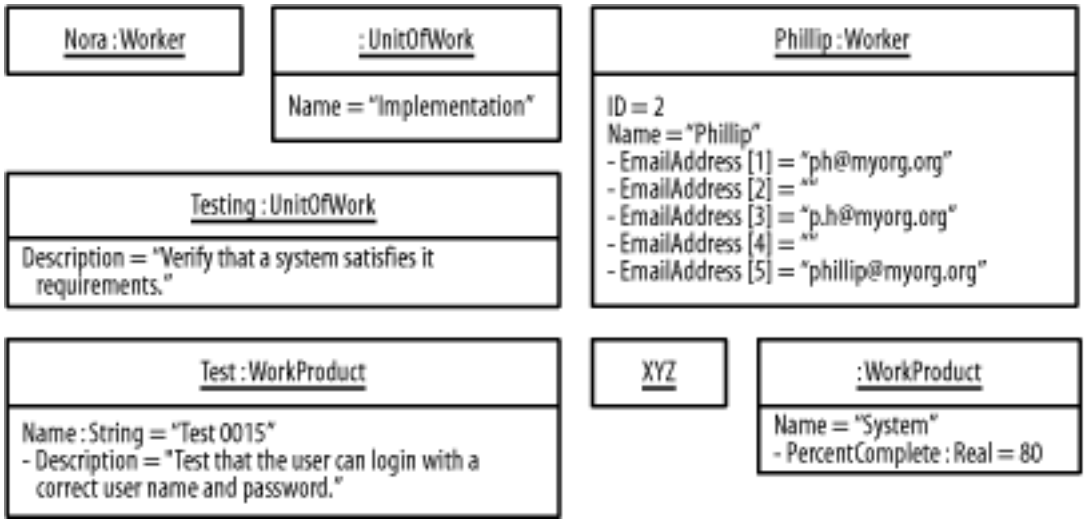

Figure 3-8. Objects

Attribute values

An *attribute value* is the value an object of a class knows for an attribute. In the UML, an attribute value is described in an object's second compartment. For example, each `Worker` object of the project management system may have a value for its name.

Consider an attribute value for holding a worker's email addresses. You may define it using the following syntax:

```
- EmailAddress [1] : String = "ph@myorg.org"
```

This definition communicates the following:

The - symbol

: Indicates that the email address attribute is private and accessible only by the object.

`[1]`

: Indicates that this is the first email address value, because there are multiple email address values.

`String`

: Indicates that the email address is a string of characters.

`"ph@myorg.org"`

: Indicates the value of the email address attribute.

Following is the general syntax to use for defining attribute values:

```
visibility name [index] : type = value
```

The syntax elements are the same as for the attributes of a class.

Operations

Because the operations and methods of a class are shared by all the objects of the class, operations are not shown on each object. For example, Figure 3-8 shows that both Nora and Phillip are workers, and therefore they share the operations and methods of the `Worker` class. There is no need to show the operations on each object, as the operations will be unnecessarily repeated each time. To determine the operations of an object, refer to that object's class.

Associations and Links

Class diagrams contain associations, and object diagrams contain links. Both associations and links represent relationships. Associations represent relationships between classes; links represent relationships between objects. The next few sections discuss the UML's representation of associations and links in detail.

Associations

As discussed in Chapter 2, an association defines a type of link and is a general relationship between classes. For example, the project management system involves various general relationships, including manage, lead, execute, input, and output between projects, managers, teams, work products, requirements, and systems. Consider, for example, how a project manager leads a team.

Binary associations

A *binary association* relates two classes. For example, one binary relationship in the project management system is between individual workers and their units of work, and another binary relationship is between individual workers and their work products.

In a UML class diagram, a binary association is shown as a solid-line path connecting the two related classes. A binary association may be labeled with a name. The name is usually read from left to right and top to bottom; otherwise, it may have a small black solid triangle next to it where the point of the triangle indicates the direction in which to read the name, but the arrow is purely descriptive, and the name of the association should be understood by the classes it relates.

Figure 3-9 shows various associations within the project management system using the most basic notation for binary associations. The associations in the figure are as follows:

- A worker is responsible for work products and performs units of work
- Units of work consume work products as input and produce work products as output.

Notice that a binary association should be named using a verb phrase. Recall from Chapter 2 that you discover associations by focusing on verbs.

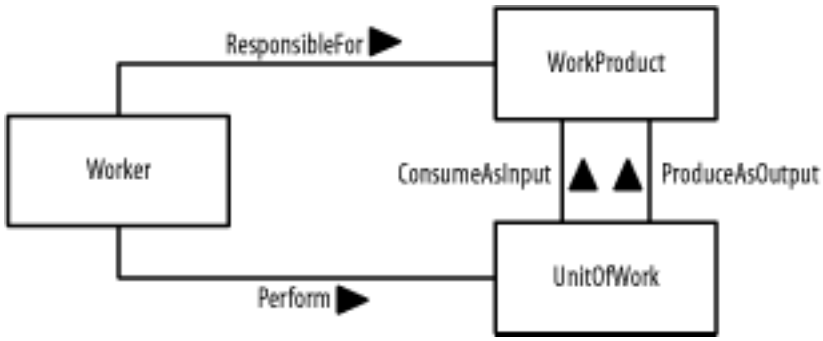

Figure 3-9. Binary associations

N-ary associations

An *n-ary association* relates three or more classes. For example, in the project management system, the use of a worker involves the worker, her units of work, and her associated work products.

In a UML class diagram, an n-ary association is shown as a large diamond with solid-line paths from the diamond to each class. An n-ary association may be labeled with a name. The name is read in the same manner as for binary associations, described in the previous section.

Figure 3-10 shows an n-ary association associated with the project management system using the most basic notation for n-ary associations. This association states that utilization involves workers, units of work, and work products. As with a binary association, an n-ary association is also commonly named using a verb phrase. However, this is not always the case — for example, the n-ary `Utilization` association shown in Figure 3-10 is described using a noun rather than a verb, because it is named from our perspective rather than the perspective of one of the classes. That is, from our perspective, we want to understand a worker's utilization relative to the other classes. From the worker's perspective, a worker is responsible for work products and performs units of work.

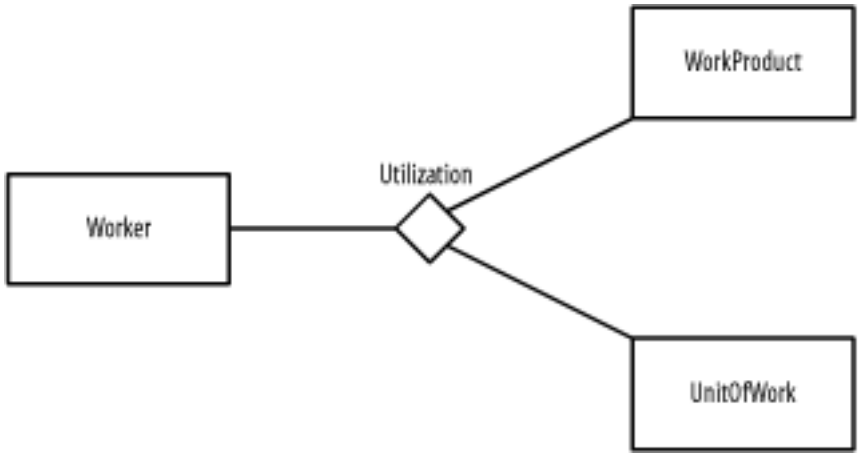

Figure 3-10. N-ary association

Association Classes

Association classes may be applied to both binary and n-ary associations. Similar to how a class defines the characteristics of its objects, including their structural features and behavioral features, an *association class* may be used to define the characteristics of its links, including their structural features and behavioral features. These types of classes are used when you need to maintain information about the relationship itself.

In a UML class diagram, an association class is shown as a class attached by a dashed-line path to its association path in a binary association or to its association diamond in an n-ary association. The name of the association class must match the name of the association.

Figure 3-11 shows association classes for the binary associations in Figure 3-9 using the most basic notation for binary association classes. The association classes track the following information:

- The reason a worker is responsible for a work product
- The reason a worker performs a unit of work
- A description of how a unit of work consumes a work product
- A description of how a unit of work produces a work product.

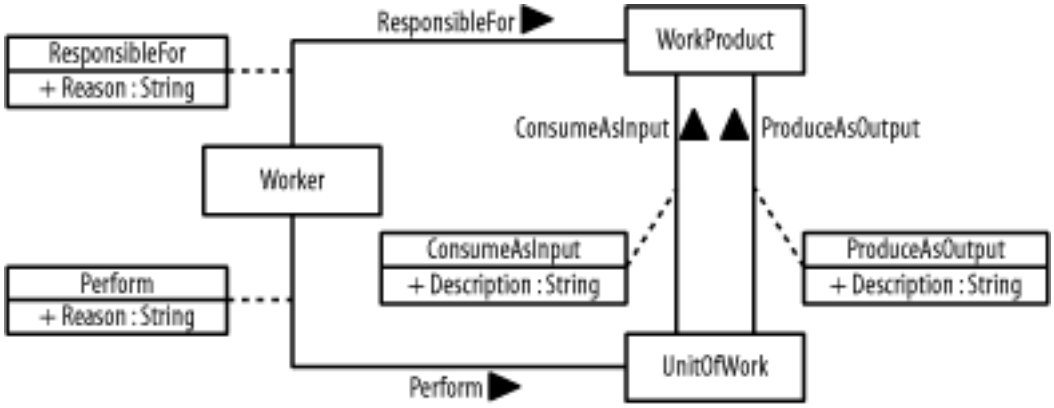

Figure 3-11. Binary association classes

Figure 3-12 shows an association class for the n-ary association in Figure 3-10 using the most basic notation for n-ary association classes. The association class tracks a utilization percentage for workers, their units of work, and their associated work products.

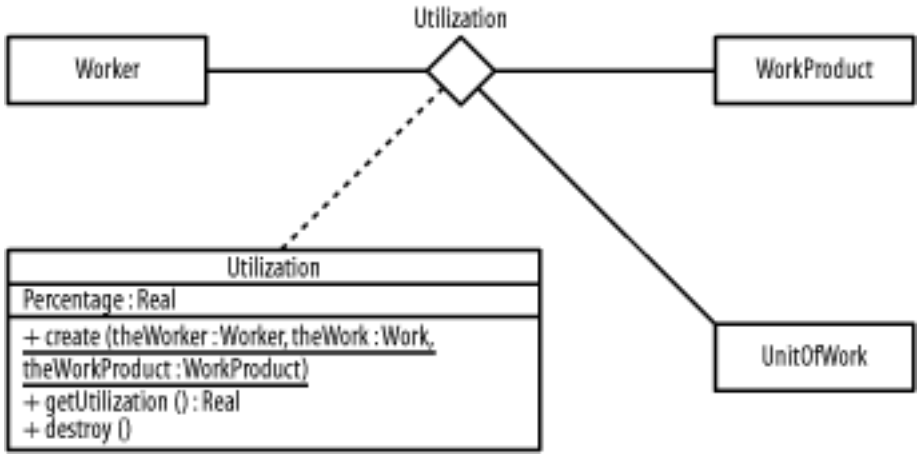

Figure 3-12. N-ary association class

Association Ends

An *association end* is an endpoint of the line drawn for an association, and it connects the association to a class. An association end may include any of the following items to express more detail about how the class relates to the other class or classes in the association:

- Rolename
- Navigation arrow
- Multiplicity specification
- Aggregation or composition symbol
- Qualifier

Rolenames

A *rolename* is optional and indicates the role a class plays relative to the other classes in an association, how the other classes "see" the class or what "face" the class projects to the other classes in the relationship. A rolename is shown near the end of an association attached to a class.

For example, a work product is seen as input by a unit of work where the unit of work is seen as a consumer by the work product; a work product is seen as output by a unit of work where the unit of work is seen as a producer by the work product, as shown in Figure 3-13. I will continue to discuss this figure in the next sections. I particularly captured significant detail in one figure so that you can see how much the UMC enables you to communicate in a figure such as this.

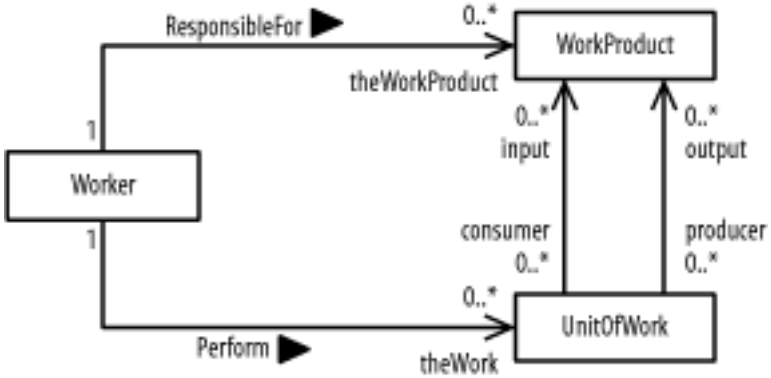

Figure 3-13. Binary association ends

Navigation

Navigation is optional and indicates whether a class may be referenced from the other classes in an association. Navigation is shown as an arrow attached to an association end pointing toward the class in question. If no arrows are present, associations are assumed to be navigable in all directions, and all classes involved in the association may reference one another.

For example, given a worker, you can determine his work products and units of work. Thus, Figure 3-13 shows arrows pointing towards work product and units of work. Given a unit of work, you can determine its input and output work products; but given a work product, you are unable to identify which worker is responsible for it or which units of work reference it as input or output (as shown in Figure 3-13 by the lack of arrows pointing to the `Worker` class). Figure 3-14 shows navigation arrows applied to an n-ary association. Given a worker, you can reference his work products and units of work to determine his utilization, but given a work product or unit of work, you are unable to determine its utilization by a worker.

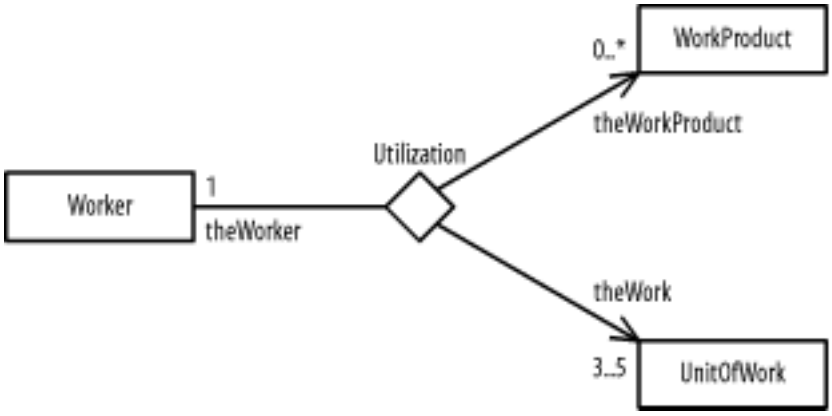

Figure 3-14. N-ary association ends

Multiplicity

Multiplicity (which is optional) indicates how many objects of a class may relate to the other classes in an association. Multiplicity is shown as a comma-separated sequence of the following:

- Integer intervals
- Literal integer values

Intervals are shown as a *`lower-bound .. upper-bound`* string in which a single asterisk indicates an unlimited range. No asterisks indicate a closed range. For example, `1` means one, `1..5` means one to five, `1, 4` means one or four, `0..*` and `*` mean zero or more (or many), and `0..1` and `0, 1` mean zero or one. There is no default multiplicity for association ends. Multiplicity is simply undefined, unless you specify it. For example:

- A single worker is responsible for zero or more work products.
- A single work product is the responsibility of exactly one worker.
- A single worker performs zero or more units of work.
- A unit of work is performed by exactly one worker.
- A unit of work may input as a consumer zero or more work products and output as a producer zero or more work products.
- A work product may be consumed as input by zero or more units of work and produced as output by zero or more units of work.

All this is shown in Figure 3-13. Continuing the example, utilization may be determined for a single worker who must have three to five units of work and zero or more work products, as shown in Figure 3-14.

To determine the multiplicity of a class, ask yourself how many objects may relate to a single object of the class. The answer determines the multiplicity on the other end of the association. For example, using Figure 3-13, if you have a single worker object, how many work products can a single worker object be responsible for? The answer is zero or more, and that is the multiplicity shown on the diagram next to the `WorkProduct` class. Using Figure 3-14, if you have

a single worker, to how many work products can the single worker be related to determine the worker's utilization? The answer is zero or more, and that is the multiplicity shown on the diagram next to the `WorkProduct` class.

Another way to determine multiplicity is to ask how many objects of a class may relate to a single object of the class on the other end of an association, or to a single object of each class on the other ends of an n-ary association. The answer determines the multiplicity for the class. For example, using Figure 3-13, how many work products is a single worker responsible for? The answer is zero or more; that is the multiplicity shown on the diagram next to the `WorkProduct` class. Also, using Figure 3-14, to how many work products is a single worker and a single unit of work related to determine the worker's utilization? The answer is zero or more; that is the multiplicity shown on the diagram next to the `WorkProduct` class.

Aggregation

Aggregation is whole-part relationship between an *aggregate*, the whole, and its *parts*. This relationship is often known as a *has-a* relationship, because the whole *has* its parts. For example, when you think of workers making up a team, you can say that a team has workers. Aggregation is shown using a hollow diamond attached to the class that represents the whole. This relationship that I've just mentioned between a team and its workers is shown in Figure 3-15. Look for the hollow diamond to the right of the `Team` class. The filled-in diamonds represent composition, which I'll discuss next. As a UML rule, aggregation is used only with binary associations.

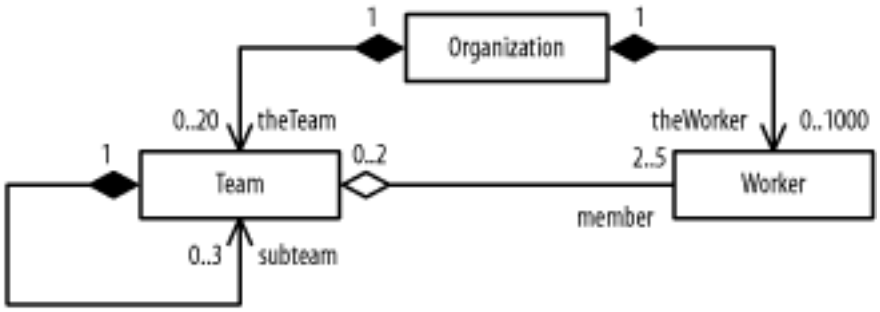

Figure 3-15. Aggregation and composition for associations

Notice in Figure 3-15 that I've done something different with `Team`. I've created a circular relationship to allow for subteams. Such a circular relationship is known as a *reflexive relationship*, because it relates two objects of the same class.

Composition

Composition, also known as *composite aggregation*, is a whole-part relationship between a *composite* (the whole) and its *parts*, in which the parts must belong only to one whole and the whole is responsible for creating and destroying its parts when it is created or destroyed. This relationship is often known as a *contains-a* relationship, because the whole *contains* its parts. For example, an organization contains teams and workers, and if the organization ceases to exist, its

teams and workers also cease to exist. The specific individuals who represent the workers would still exist, but they would no longer be workers of the organization, because the organization would no longer exist. Composition is shown using a filled diamond attached to the class that represents the whole. The relationships between a team, its workers, and an organization are shown in Figure 3-15. The filled-in diamond at the endpoint of the subteam relationship in Figure 3-15 indicates that teams contain their subteams. As a UML rule, composition is used only with binary associations.

Notice how much information is being communicated in Figure 3-15. It shows that an organization may contain 0 to 20 teams and 0 to 1,000 workers. Furthermore, each team has 2 to 5 workers and each worker may be a member of 0 to 2 teams. In addition, a team may contain 0 to 3 subteams.

To determine if you should use an aggregation or composition, ask yourself a few questions. First, if the classes are related to one another, use an association. Next, if one class is part of the other class, which is the whole, use aggregation; otherwise, use an association. For example, Figure 3-15 shows that workers are part of a team and organization, teams are part of an organization, and subteams are part of teams. Finally, if the part must belong to one whole only, and the whole is responsible for creating and destroying its parts, use composition; otherwise, simply use aggregation. For example, Figure 3-15 shows that a team and worker must belong to one organization only, and the organization is responsible for determining (or creating and destroying) its teams and workers. It also shows that a subteam must belong to one team only, and the team is responsible for determining (or creating and destroying) its subteams. If this is unclear, keep things simple and use associations without aggregation or composition.

Composition also may be shown by graphically nesting classes, in which a nested class's multiplicity is shown in its upper-right corner and its rolename is indicated in front of its class name. Separate the rolename from the class name using a colon. Figure 3-16 uses the graphical nesting of teams and workers in organizations to communicate the same information as shown in Figure 3-15.

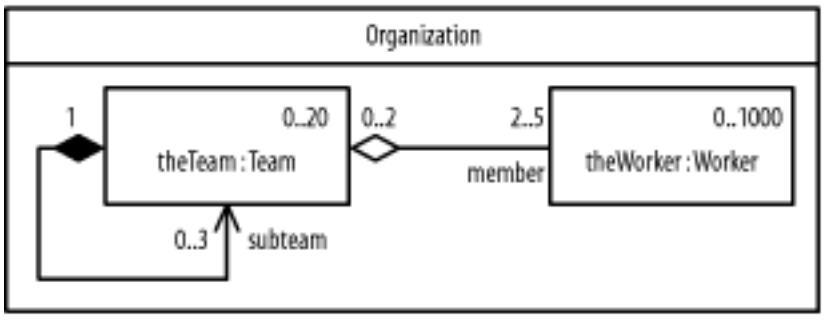

Figure 3-16. Alternate form of composition for associations

Figure 3-17 uses the graphical nesting of subteams within teams to communicate the same information as Figure 3-15 and Figure 3-16. The result is a nested class inside a nested class.

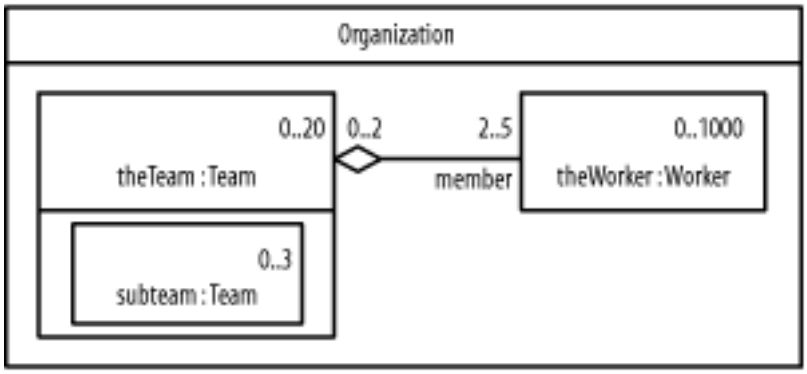

Figure 3-17. Doubly nested composition for associations

Qualifiers

A *qualifier* is an attribute of an association class that reduces the multiplicity across an association. For example, Figure 3-13 shows that multiplicity between work products and units of work is zero or more for both associations; that is, there may be many work products associated with a single unit of work and there may be many units of work associated with a single work product. Rather than simply say that there are "many" objects involved in the relationship, you can communicate a more finite number.

You can reduce the multiplicity between work products and units of work by asking yourself what you need to know about a unit of work so that you can define a more specific multiplicity — one that isn't unbounded on the high-end. Likewise, you can ask yourself the same question about the association between work product and units of work. If you have a work product and the name of a unit of work, you can determine whether a relationship exists between the two; likewise, if you have a unit of work and the name of a work product, you can determine whether a relationship exists between those two. The trick is to document precisely what information is needed so you can identify the objects on the other end of the relationship. This is where the qualifier comes into play.

Essentially, a qualifier is a piece of information used as an index to find the objects on the other end of an association. A qualifier is shown as a small rectangle attached to a class where an object of the class, together with a value for the qualifier, reduces the multiplicity on the other end of the association. Qualifiers have the same notation as attributes, have no initial values, and must be attributes of the association or the class on the other end of the association.

The relationships between work products and units of work and their qualifiers are shown in Figure 3-18. The qualifiers indicate that a work product with the name of a unit of work may identify a unit of work, and that a unit of work with the name of a work product may identify a work product. Notice that I've reduced the multiplicity of 0..* shown in Figure 3-13 to 0..1 in Figure 3-18. The qualifier enables me to do this. As a UML rule, qualifiers are used only with binary associations.

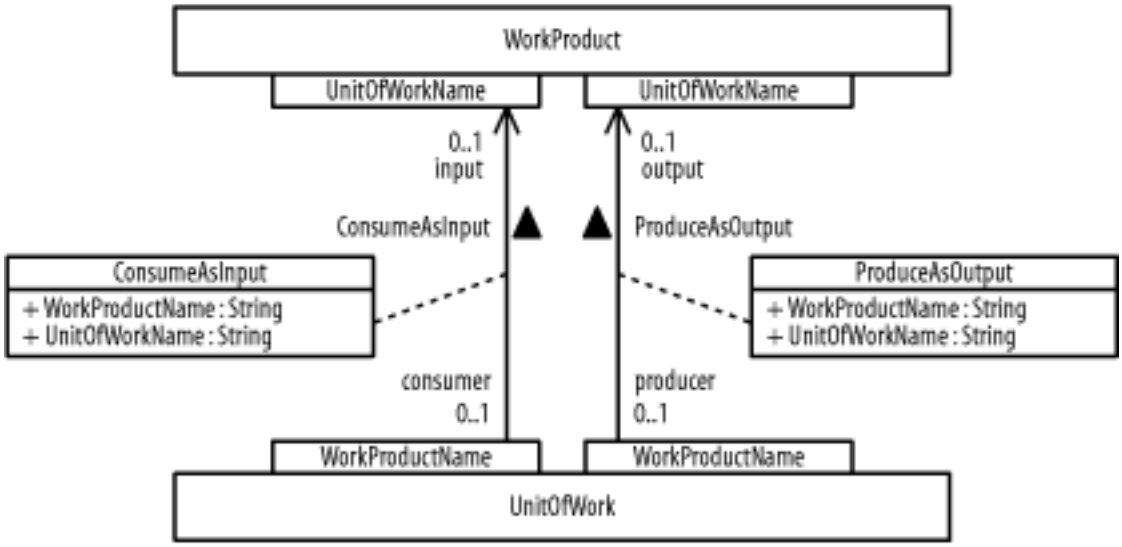

Figure 3-18. Qualifiers for associations

Links

As discussed in Chapter 2, a *link* is a specific relationship between objects. For example, the project management system involves various specific relationships, including specific manage, lead, execute, input, output, and other relationships between specific projects, managers, teams, work products, requirements, systems, and so forth. A link is an instance of an association, and the UML supports different types of links that correspond to the different types of associations.

The general rules for representing links in a UML diagram are as follows:

- Label links with their association names, and underline the names to show that they are specific instances of their respective associations.
- Ensure that link ends are consistent with their corresponding association ends.
- Translate association multiplicity into one or more specific links between specific objects.

The next few sections show how to apply these rules to the various link types.

Binary links

A *binary link*, which is a specific relationship between two objects, is shown as a solid-line path connecting the two objects in a UML object diagram. For example, a specific worker is related to specific units of work and work products in the project management system. A link may have its association name shown near the path (fully underlined), but links do not have instance names.

Figure 3-19 shows various objects associated with the classes shown in Figure 3-13 and the association classes shown in Figure 3-11. Additionally, Figure 3-19 includes several link objects. Figure 3-19 describes an anonymous worker that performs a project—a unit of work—

that consumes a `Problem Statement` work product and produces a system work product. `ResponsibleFor` and `Performed` are two links in Figure 3-19.

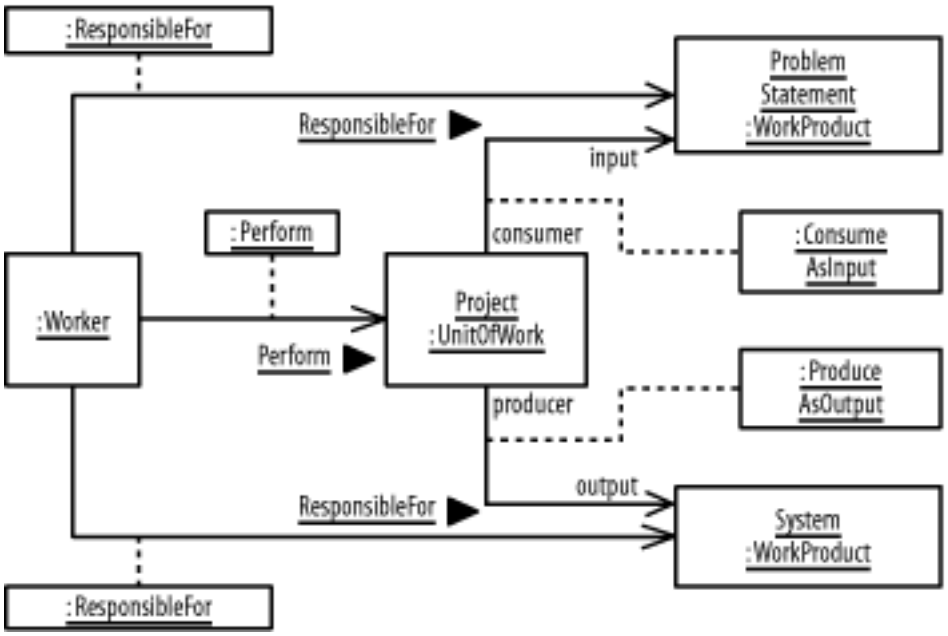

Figure 3-19. Binary links

The only difference between a binary link and a binary association is that the binary link is specific, and thus has its association name underlined.

N-ary links

An *n-ary link*, a relationship between three or more objects, is shown as a large diamond with solid-line paths from the diamond to each object in a UML object diagram. For example, the utilization of a specific worker involves the worker, the worker's specific units of work, and the worker's specific work products in the project management system. A link may have its association name shown near the path, and because a link is specific, its association name should be fully underlined. However, links do not have instance names. As a UML rule, aggregation, composition, and qualifiers may not be used with n-ary links.

Figure 3-20 shows various objects associated with the classes shown in Figure 3-14 and the association classes shown in Figure 3-12. Additionally, Figure 3-20 includes a link object named `Utilization`. Figure 3-20 describes the utilization of an anonymous team, its work, and work products.

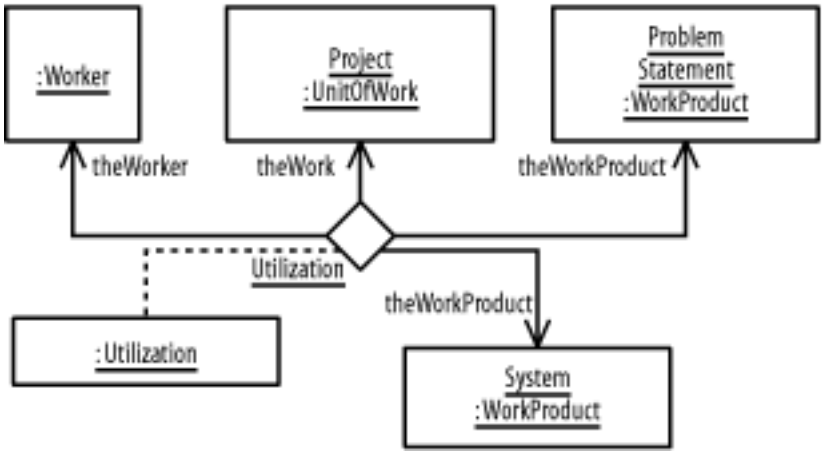

Figure 3-20. N-ary link

Link Objects

A *link object* is a specific instance of an association class, and thus has all the characteristics, including structural and behavioral features, defined by the association class. Structural features include attribute values and perhaps other links. Behavioral features include operations and methods, which are shared by all the links of an association class. Whenever an association has a related association class, each of its links has a corresponding link object. This link object defines attribute values for the link's structural features. In addition, the behavioral features defined by the link's association class apply to the link objects. In a UML object diagram, a link object is shown as an object rectangle attached by a dashed-line path to its link path in a binary link, or attached to its link diamond in an n-ary link. As with all UML elements representing specific objects or links, link object names should be fully underlined.

Figure 3-19 shows link objects for the binary associations in Figure 3-13, and Figure 3-20 shows a link object for the n-ary association in Figure 3-14.

Link Ends

A *link end*, similar to an association end, is an endpoint of a link and connects the link to an object. A link end may show its association end's rolename, navigation arrow, aggregation or composition symbol, and values for its association end's qualifiers.

Rolenames

A link end's rolename must match its association end's rolename. For example, Figure 3-13 shows that a `Worker` is responsible for a `WorkProduct`. The specific association used is `ResponsibleFor`; this same association name is used again in Figure 3-19 to describe the specific links between a specific `Worker` and the two specific work products: `ProblemStatement` and `System`.

Navigation

Likewise, a link end's navigation must match its association end's navigation. For example, the arrows on the two `ResponsibleFor` links in Figure 3-19 both point to instances of `WorkProduct`. This is consistent with Figure 3-13, which shows the arrow for the `ResponsibleFor` association pointing to the `WorkProduct` class.

Multiplicity

Multiplicity is shown only on association ends. This is because an association describes the multiplicity between two or more classes of objects. A link however, is between specific objects. Thus, in an object diagram, multiplicity manifests itself in terms of a specific number of links pointing to a specific number of discrete objects. For example, the multiplicity shown in Figure 3-13 indicates that a `Worker` object may be responsible for zero to many (0..*) `WorkProduct` objects. In Figure 3-19, two specific `WorkProduct` objects are shown. Figure 3-19 is specific, and the specific multiplicity in this case is two: the specific `Worker` object is responsible for two specific `WorkProduct` objects.

Aggregation

Aggregation is shown using a hollow diamond, as shown in Figure 3-21 through Figure 3-23. Figure 3-21 shows three teams named Eagle, Falcon, and Hawk. Jonathan, Andy, Nora, and Phillip are on the Eagle team, while Nora and Phillip are also on the Hawk team.

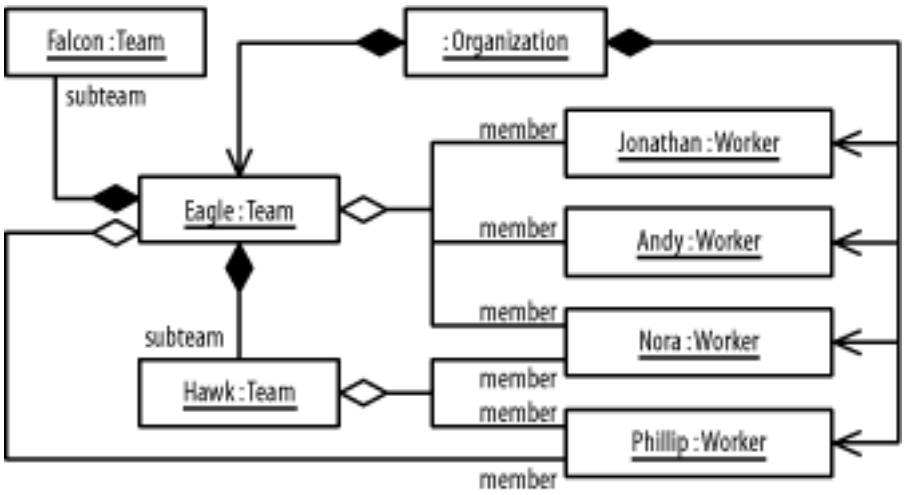

Figure 3-21. Aggregation and composition for links

Composition

Composition may be shown using a filled diamond or graphical nesting, as in Figure 3-21. Figure 3-21 shows that the two teams, Falcon and Hawk, are subteams of the Eagle team. In addition, the filled-in diamond next to the `Organization` class indicates that all the individuals on

these teams belong to the same organization, and that the Eagle team itself belongs to the organization.

Figure 3-22 uses the graphical nesting of teams and workers in organizations to communicate the same information as Figure 3-21.

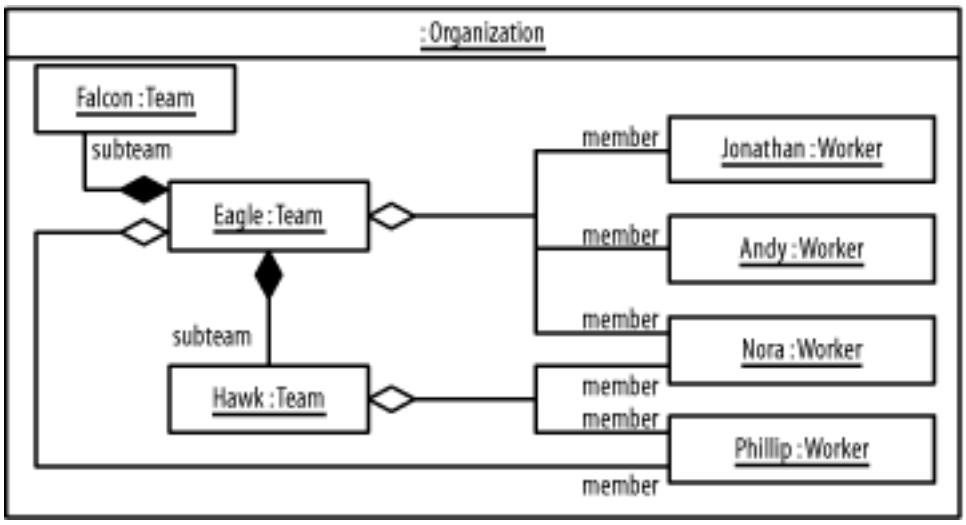

Figure 3-22. Alternate form of composition for links

Figure 3-23 uses the graphical nesting of subteams in teams to communicate the same information as Figure 3-21 and Figure 3-22.

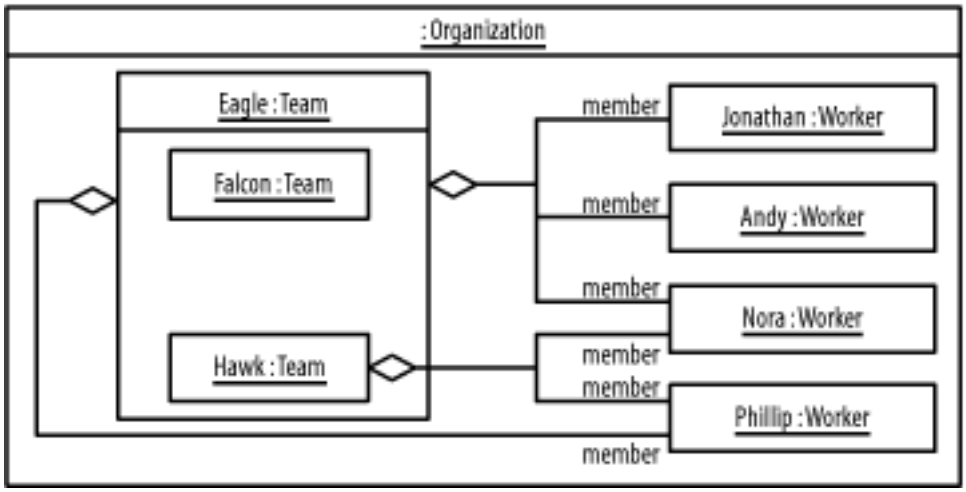

Figure 3-23. Doubly nested composition for links

Qualifiers

Values for link qualifiers have the same notation as for object attribute values. Figure 3-24 shows how qualifier values associate a project with its problem statement (named `Problem Statement`) and system (named `PM-System`).

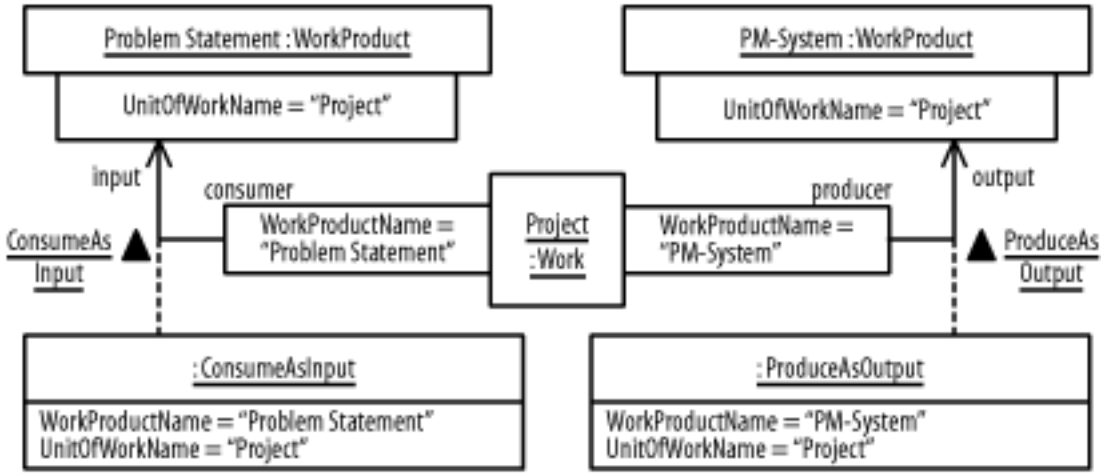

Figure 3-24. Qualifier values for links

Types, Implementation Classes, and Interfaces

The kinds of classes discussed thus far are known as *undifferentiated classes*, and are commonly used during design activities within a development process. You can also differentiate between three different kinds of classes, called *differentiated classes*. These include:

- Types
- Implementation classes
- Interfaces

These differentiated classes closely relate to different activities in the development process. The next few sections discuss these differentiated classes.

Types

A *type* is a class that may have attributes, associations, and operations, but does not have any methods. A type defines a role an object may play relative to other objects, similar to how a rolename indicates the role a class plays relative to other classes in an association. For example, a `Worker` object may play the role of a project manager, resource manager, human resource, or system administrator. A type is shown as a class marked with the `type` keyword. These types of workers are shown in Figure 3-25.

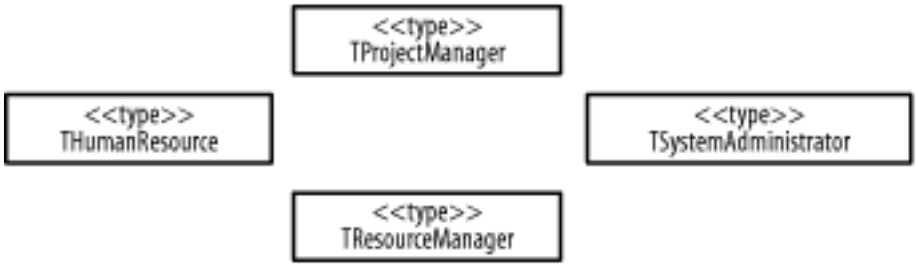

Figure 3-25. Types

Types are commonly used during analysis activities within a development process to identify the kinds of objects a system may require. You can think of types as conceptual classes, because they are ideas for possible classes. Also, because types do not have methods and represent roles only, they do not have instances.

Types may be used with binary and n-ary association and link ends. A comma-separated list of one or more type names may be placed following a rolename to indicate the roles a class or object plays in the relationship. Separate the rolename from the list of types using a colon. If no rolename is used, the type names are placed following a colon.

Figure 3-26 uses the types from Figure 3-25 to update Figure 3-13. It shows the various roles a worker may play relative to work products and units of work. A worker may be a project manager, resource manager, human resource, and system administrator.

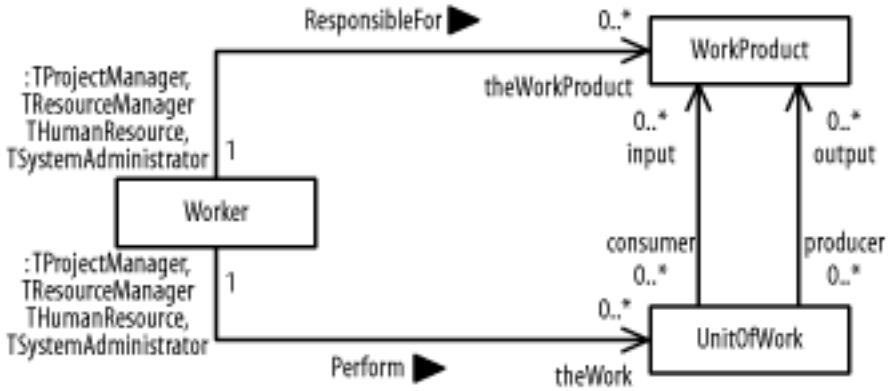

Figure 3-26. Types with association ends

Implementation Classes

An *implementation class* is a class that may have attributes, associations, operations, and methods. An implementation class defines the physical implementation of objects of a class. For example, if you were to implement our classes in a database management system, the `Worker` class might be implemented as an employee table, the `WorkProduct` class might be implemented as an artifact table, and the `UnitOfWork` class might be implemented as work order table. An implementation class is shown as a class marked

with the `implementationClass` keyword. The three implementation classes just mentioned are shown in Figure 3-27.

Figure 3-27. Implementation classes

Implementation classes are commonly used during the later part of design and during implementation activities within a development process to identify how objects are implemented for a system. You can think about implementation classes as physical "code" classes because, they are physical implementations of classes.

Interfaces

An *interface* is a class that may have operations but may not have attributes, associations, or methods. An interface defines a service or contract as a collection of public operations. For example, a project manager must be able to initiate and terminate a project, plan and manage a project while leading a team, and so forth. A resource manager must be able to assign and unassign human resources to and from a team. A work product must be producible and consumable; it is produced by being created or written, and consumed by being read or destroyed. Interfaces can be used to define these collections of operations.

An interface is shown as a class marked with the `interface` keyword, and because interfaces don't have attributes, the second compartment is always empty and, therefore, not shown. An interface also may be shown as a small circle with the interface name placed near the symbol and the operations of the interface not shown. Examples of both interface representations are shown in Figure 3-28.

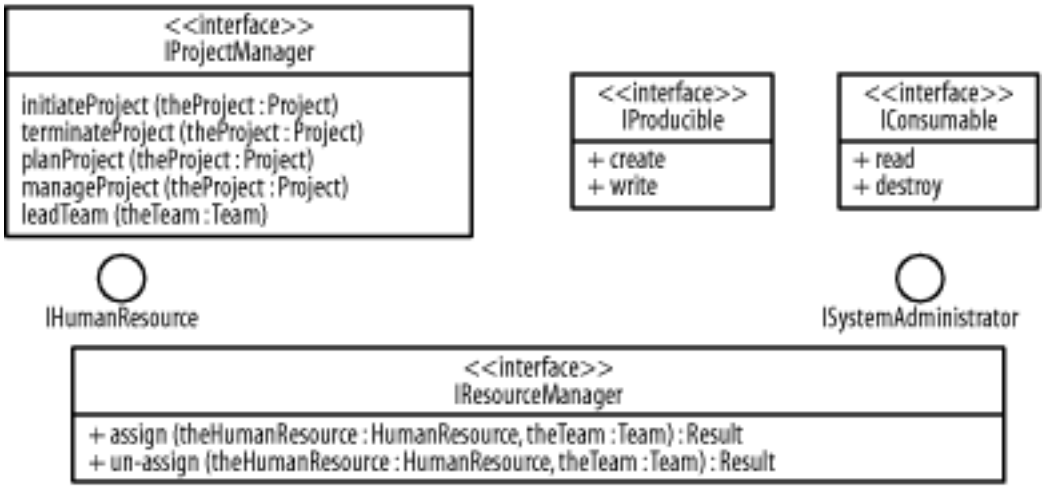

Figure 3-28. Interfaces

Interfaces are commonly used during analysis and design activities within a development process to identify services that classes and their objects provide. You can think of interfaces as application programming interfaces (APIs), because they define a collection of operations that are commonly used together and thus define a more general service. Because interfaces do not have methods but merely represent services, they do not have instances.

Interfaces may be used with binary and n-ary association and link ends to indicate the services (or interfaces) that a class provides in the relationship. Begin with a rolename followed by a colon, then add a comma-separated list of one or more interface names, as shown in Figure 3-29.

Figure 3-29 uses the interfaces from Figure 3-28 to update Figure 3-26. It shows the various services a work product provides to units of work, including an interface for consumption of the work product and an interface for production of the work product. It also shows the various interfaces a worker provides in association with work products and units of work. Interfaces and types may be listed together, but types are more commonly used during analysis activities, while interfaces are more commonly used during analysis and design activities. Because both interfaces and types may be used during analysis and design, it is very important to have a standard naming convention. For example, one convention is to prefix interfaces with an `I` and to prefix types with a `T`.

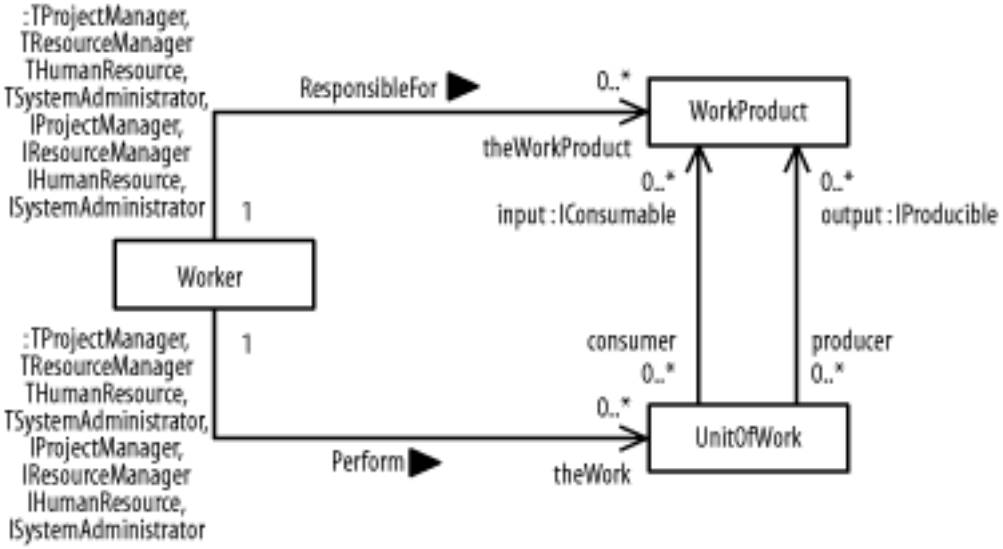

Figure 3-29. Interfaces with association ends

Generalizations, Realizations, and Dependencies

While it is common to use types during analysis activities, interfaces during analysis activities and design activities, undifferentiated classes during design activities, and implementation classes during the later part of design and during implementation activities, how are all these elements related? Generalizations, realizations, and dependencies, called *specialized relationships*, address the question of how undifferentiated and differentiated classes are related. The next few sections discuss these relationships.

Generalizations

A generalization between a more general element and a more specific element of the same kind indicates that the more specific element receives the attributes, associations and other relationships, operations, and methods from the more general element. The two elements must be of the same kind. For example, a generalization relationship can be between two types but not between a type and an interface. Generalization relationships may exist for the following kinds of elements:

- Types
- Undifferentiated classes
- Implementation classes
- Interfaces

A generalization is shown as a solid-line path from the more specific element to the more general element, with a large hollow triangle at the end of the path connected to the more general element. You'll see examples of this as I discuss each specific type of generalization in the following sections.

Types

The project manager, human resource, and system administrator types shown earlier in Figure 3-25 are specific types of human resources. You can model a generalization of these three types to factor out structure and behavior common to all. A generalization between types allows us to reuse a type's attributes, associations, and operations to define a new type. Figure 3-30 shows that the `THumanResource` type is a generalization of `TProjectManager`, `TSystemAdministrator`, and `TResourceManager`.

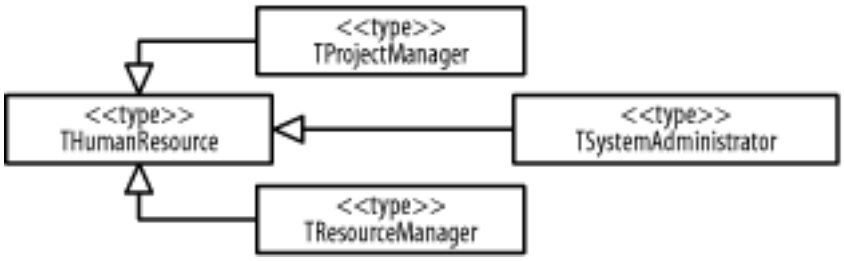

Figure 3-30. Generalizations between types

Because a generalization between a more general type and a more specific type indicates that the more specific type is a specialized form of the more general type, those classes that may play the roles of the more specific type may also play the roles of the more general type. Therefore, Figure 3-30 shows that those objects that play the specific roles of project manager, resource manager, and system administrator may also each play the more general role of a human resource.

Undifferentiated classes

The `Worker` class shown in Figure 3-26 may have a more specialized undifferentiated class of human resource, which itself has more specialized undifferentiated classes, including project managers, resource managers, and system administrators. You can nest generalization relationships for undifferentiated classes as well as types, interfaces, and differentiated classes. For example, project managers, resource managers, and system administrators all could be specializations of a human resource. A human resource, on the other hand, could be a specialization of a worker, or a worker could be a generalization of a human resource. A generalization between undifferentiated classes allows us to reuse a class's attributes, associations, operations, and methods to define a new undifferentiated class. The relationships among the undifferentiated classes just mentioned are shown in Figure 3-31.

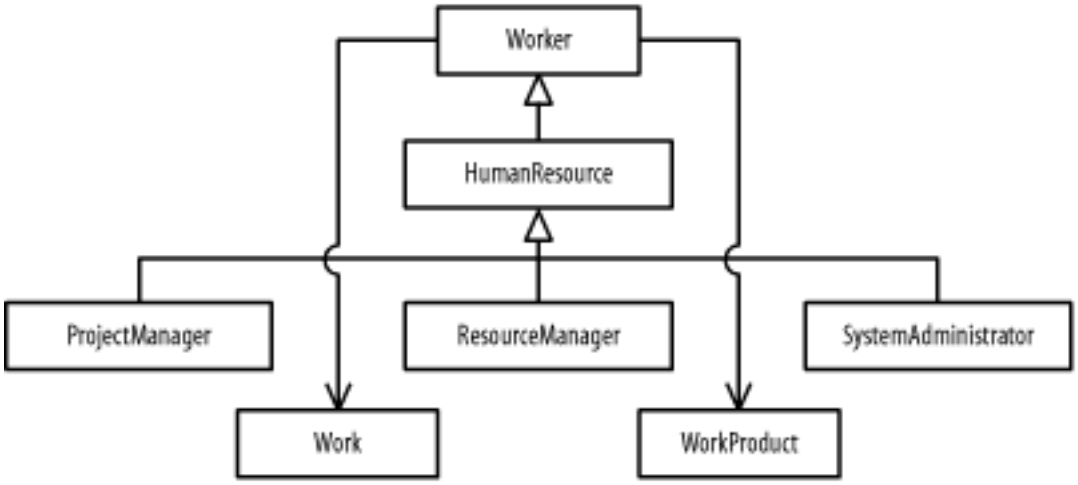

Figure 3-31. Generalizations between classes

Figure 3-30 shows three generalization paths, while Figure 3-31 combines three generalization paths from the `ProjectManager`, `ResourceManager`, and `SystemAdministrator` classes into one path that connects to the `HumanResource` class. Whenever several paths of the same kind connect to a single element, the UML allows you to combine those paths into a single path as shown in this figure. Also, when paths cross but do not connect, the UML allows you to show this with a small, semicircular jog by one of the paths, as shown for the associations between the `Worker` class and the `UnitOfWork` and `WorkProduct` classes. The jog indicates that the line with the jog does not connect in any way with the other line passing through the jog.

Given a specific class, any immediately more general classes are called *parents* or *super-classes*. Any immediately more specific classes are called *children* or *subclasses*. General classes that are not parents (i.e., not immediately more general) are called *ancestors*. More specific classes that are not children (i.e., not immediately more specific) are called *descendants*. Therefore, Figure 3-31 shows that the `Worker` class is the parent of the `HumanResource` class, and the `ProjectManager`, `ResourceManager`, and `SystemAdministrator` classes are the children of the `HumanResource` class. It also shows that the `Worker` class is an ancestor of the `ProjectManager`, `ResourceManager`, and `SystemAdministrator` classes, and that these classes are descendants of the `Worker` class.

Because a generalization between two undifferentiated classes indicates that objects of the more specific undifferentiated class are more specialized objects of the more general undifferentiated class, objects of the more specific undifferentiated class may be substituted for objects of the more general undifferentiated class. Therefore, Figure 3-31 shows that project manager, resource manager, and system administrator objects may be substituted for human resource objects.

Implementation classes

A generalization between implementation classes allows us to reuse an implementation class's attributes, associations, operations, and methods to define a new implementation class. Earlier, I raised the possibility of implementing the classes shown in Figure 3-27 as database tables. In

such a case, the three implementation classes in Figure 3-27 could all be specialized classes of a database table class. Figure 3-32 shows this by making `Employee`, `WorkOrder`, and `Artifact` into specializations of a more general `DatabaseTable` class.

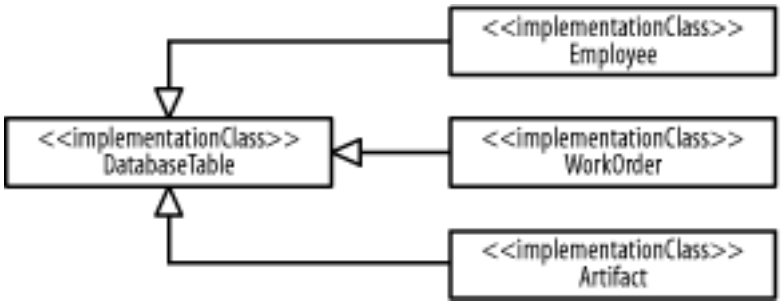

Figure 3-32. Generalizations between implementation classes

Because a generalization between a more general implementation class and a more specific implementation class indicates that objects of the more specific implementation class are specialized objects of the general implementation class, objects of the specific implementation class may be substituted for objects of the general implementation class. Therefore, Figure 3-32 shows that employee, work order, and artifact objects may be substituted for database table objects.

Interfaces

The project manager, human resource, and system administrator interfaces shown in Figure 3-28 are more specific versions of the human resource interface. A generalization between interfaces allows us to reuse an interface's operations to define a new interface. The relationships among the interfaces just mentioned are shown in Figure 3-33.

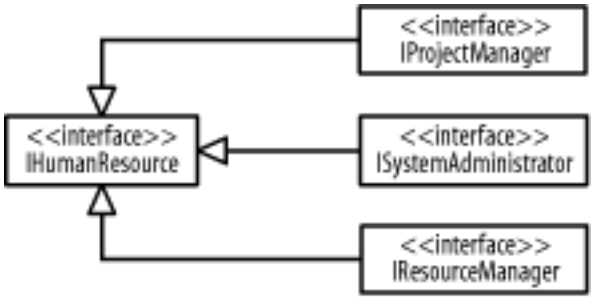

Figure 3-33. Generalizations between interfaces

Because a generalization between a more general interface and a more specific interface indicates that the more specific interface is a specialized form of the more general interface, those classes that provide the service defined by the more specific interface may also provide the service defined by the more general interface. Therefore, Figure 3-33 shows that those objects that provide the project manager, human resource, and system administrator interfaces also provide the human resource interface.

Realizations

A *realization* from a source element (called the *realization element*) to a target element (called the *specification element*) indicates that the source element supports at least all the operations of the target element without necessarily having to support any attributes or associations of the target element. For example, an undifferentiated class or implementation class may play the role defined by a type and may provide the service defined by an interface, if the class supports the operations defined by the type and interface. A realization allows us to reuse the operations of types and interfaces where a realization element is said to *realize* its specification elements.

A realization is shown as a dashed-line path from the source element to the target element, with a large hollow triangle at the end of the path connected to the target element. When the target element is an interface shown as a small circle, the realization is shown as a solid-line path connecting the source and interface.

Undifferentiated classes

Figure 3-29 shows a list of types and interfaces that the `Worker` class supports. Based on Figure 3-29, Figure 3-34 shows that the `Worker` class realizes those types and interfaces. The source element is the `Worker` class, and the other elements are the targets. Figure 3-29 shows how interfaces and types are used in the various associations between the `Worker` class and other classes, while Figure 3-34 shows that the `Worker` class explicitly realizes these interfaces and types independent of how they are used in relationships.

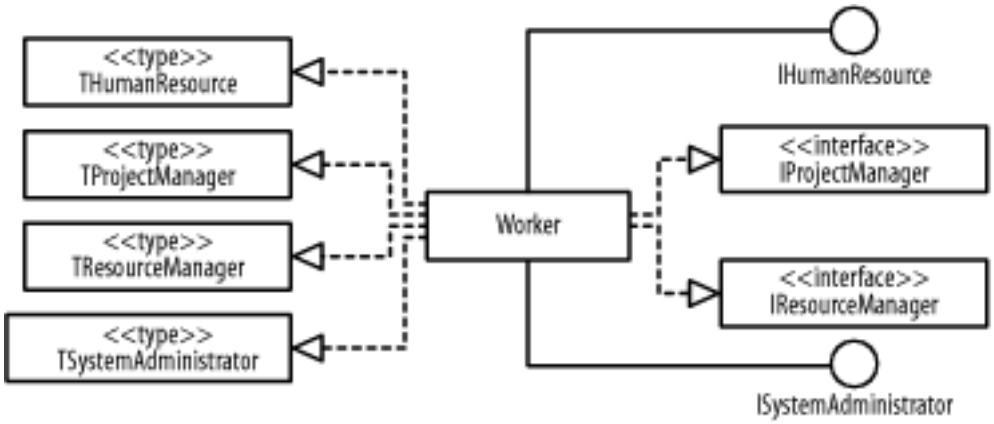

Figure 3-34. Realizations for the Worker class

Based on Figure 3-29, Figure 3-35 shows the interfaces work products realize. The source element is the `WorkProduct` class and the other elements are the targets.

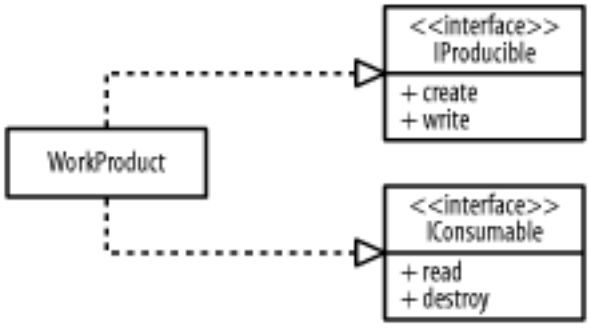

Figure 3-35. Realizations for the WorkProduct class

Because a realization from a source class to a target element indicates that objects of the source class support all the operations of the target element, objects of the source class may be substituted for objects of other classes that also realize the same target element. Therefore, Figure 3-34 shows that a worker object may be substituted for objects of other classes that realize the same types and interfaces as the worker object, and objects of other classes that realize the same types and interfaces as the worker object may be substituted for worker objects. That is, if two objects realize the same type or interface, they may be substituted for one another. Figure 3-35 illustrates this.

Implementation classes

Based on Figure 3-27, Figure 3-36 shows that the `Worker` class may be implemented as an employee table, the `WorkProduct` class may be implemented as an artifact table, and the `UnitOfWork` class may be implemented as work order table, if you are to implement your classes in a database management system. This is indicated with the realization relationships between the `Employee` implementation class realizing the `Worker` class, the `WorkOrder` implementation class realizing the `UnitOfWork` class, and the `Artifact` implementation class realizing the `WorkProduct` class.

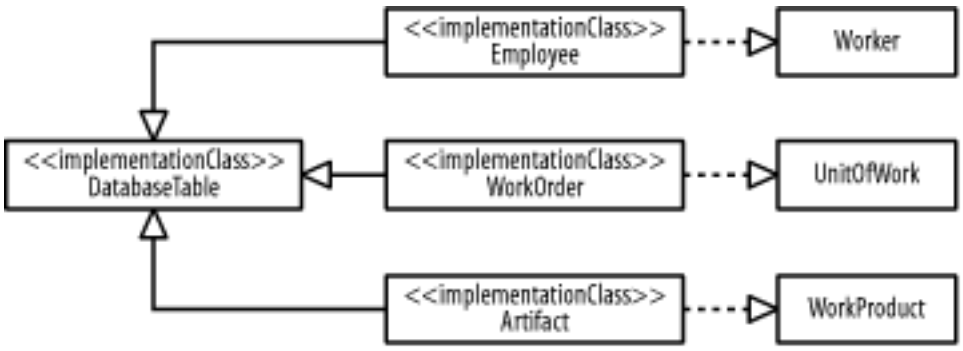

Figure 3-36. Realizations of undifferentiated classes by implementation classes

When an implementation class realizes an undifferentiated class, it must also realize the types and interfaces that the undifferentiated class realizes; otherwise, it could not play the roles

defined by the undifferentiated class's types and provide the services defined by the undifferentiated class's interfaces.

Based on Figure 3-36 and Figure 3-34, Figure 3-37 shows the types and interfaces the `Employee` implementation class realizes.

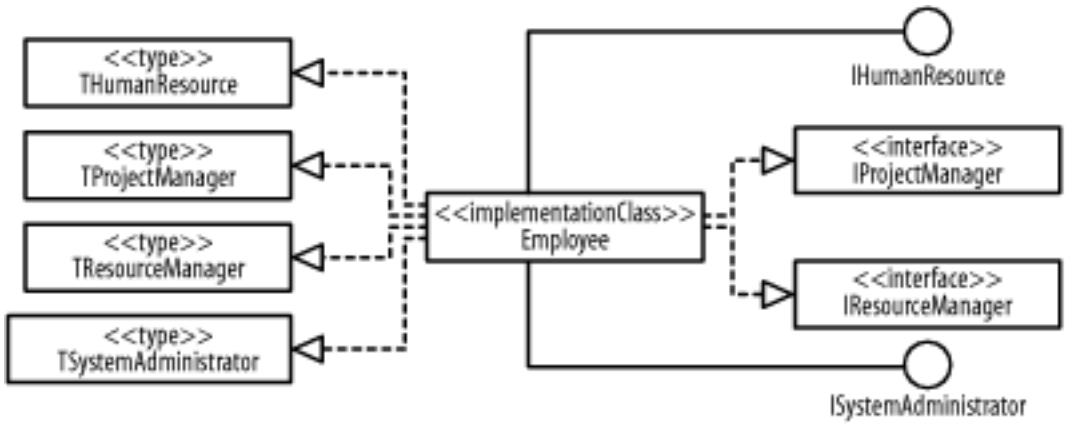

Figure 3-37. Realizations for the Employee implementation class

Based on Figure 3-36 and Figure 3-35, Figure 3-38 shows the interfaces the `Artifact` implementation class realizes.

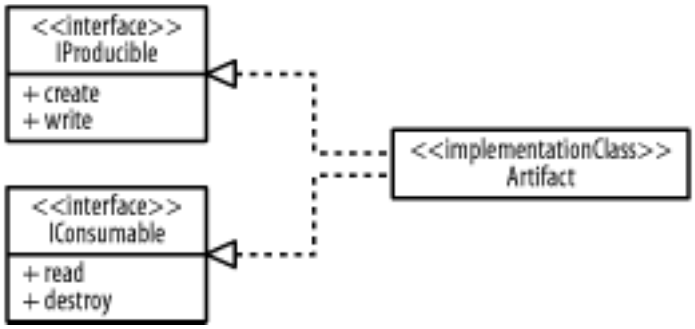

Figure 3-38. Realizations for the Artifact implementation class

Because a realization from a source class to a target element indicates that objects of the source class support all the operations of the target element, objects of the source class may be substituted for objects of other classes that also realize the same target element. Therefore, Figure 3-37 shows that an employee object may be substituted for objects of other classes that realize the same types and interfaces as the employee object, and objects of other classes that realize the same types and interfaces as the employee object may be substituted for employee objects. Figure 3-38 shows the same for an artifact object.

Dependencies

A *dependency* from a source element (called the *client*) to a target element (called the *supplier*) indicates that the source element uses or depends on the target element; if the target element changes, the source element may require a change. For example, a `UnitOfWork` uses the `IConsumable` interface as a consumer and uses the `IProducible` interface as a producer; if either interface changes, the `UnitOfWork` may require a change. Figure 3-29 shows the interfaces used by `UnitOfWork`.

A dependency is shown as a dashed-line path from the source element to the target element. The dependency may be marked with the `use` keyword; however, the keyword is often omitted because the meaning is evident from how the dependency is used. Also, notice that a dependency does not have a large hollow triangle at the end of the path, but has an open arrow.

Based on Figure 3-29, Figure 3-39 shows the dependencies between units of work and work products. Notice that a realization may be shown as a dependency marked with the `realize` keyword, as shown in Figure 3-39 between the `WorkProduct` class and the `IProducible` interface.

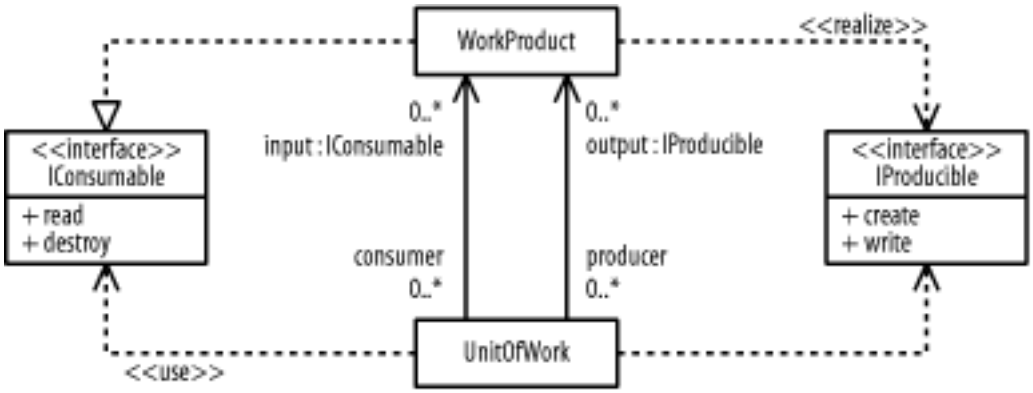

Figure 3-39. Realizations and dependencies

Figure 3-40 shows the dependencies between the interfaces discussed in this chapter and the parameter and return types for their operations. For example, `IProjectManager` must depend on `Project`, because many of its operations take a `Project` object as a parameter.

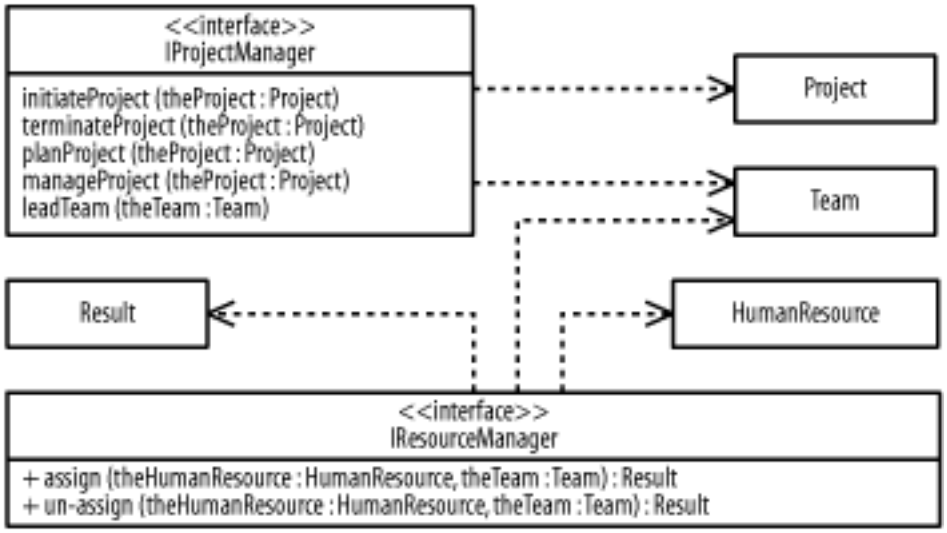

Figure 3-40. Dependencies between interfaces and return types

Packages and Subsystems

As a model may have hundreds (if not thousands) of model elements, how do you organize the elements that make up a system and their relationships? And how do you use this information to determine how best to develop the system while considering technical trade-offs concerning the system, including which elements may be developed in parallel, which elements may be purchased rather than built, and which elements may be reused? Packages and subsystems, called *model management elements*, address these questions.

Packages

A *package* is a grouping and organizing element in which other elements reside, which must be uniquely named. In the UML, packages are used in a manner similar to the way directories and folders in an operating system group and organize files. For example, the project management system may be decomposed into a collection of classes organized into packages as follows:

`Utility`

: Date, time, and other utility classes

`Workers`

: The `Worker` class and any other worker-related classes in which the `Worker` class is contained inside of a package named `Generic`

`Generic`

: Generic classes such as the `Worker` class and any other worker-related classes

`Work Units`

: The `UnitOfWork` class and any other work-related classes

`Work Products`

: The `WorkProduct` class and any other work product-related classes

`User Interface`

: A package housing classes responsible for providing a user interface through which users may interact with the system

`Business Processing`

: A package housing classes responsible for implementing business functionality

`Data`

: A package housing classes responsible for implementing data storage functionality

Packages allow us to partition our system into logical groups and then to relate these logical groups; otherwise, it could become overwhelming to work with every class and its relationship at once. A package is shown as a large rectangle with a small rectangle or "tab" attached to its top, left side. The packages in the previous list, together with their relationships, are shown in Figure 3-41.

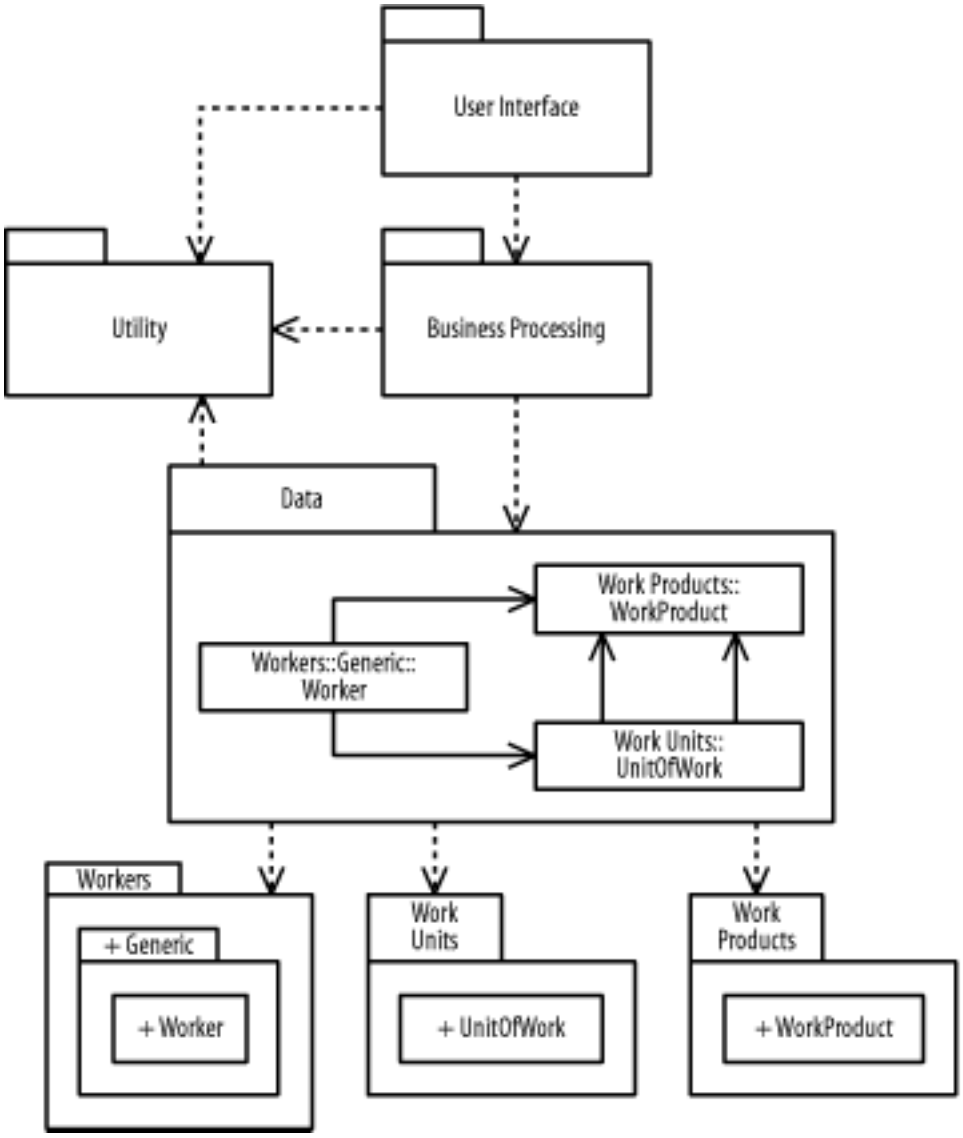

Figure 3-41. Packages

You can show the name of a package inside of its large rectangle when the package contents are not shown; otherwise, when the package contents are shown in the large rectangle, you should show the name of a package inside its tab. In Figure 3-41, you see that the `User Interface`, `Utility`, and `Business Processing` packages don't show their contents, while all the other packages do show some of their contents. Thus, the `User Interface`, `Utility`, and `Business Processing` packages have their package names within the large rectangle, whereas the other packages have their names in their respective tabs. Each element inside a package may have a visibility symbol indicating whether the element is accessible from outside its package (i.e., is public). Figure 3-41 shows that the `Worker`, `UnitOfWork`, and `WorkProduct` classes are public. The `Generic` package located inside of the `Workers` package is also public.

A dependency from a source package to a target package indicates that the contents of the source package use the contents of the target package. Figure 3-41 shows that the `Data` package uses

the Workers, Work Units, and Work Products packages. It also shows that the User Interface, Business Processing, and Data packages use the Utility package, while the User Interface package uses the Business Processing package, which in turn uses the Data package.

A package defines a *namespace*, a part of a model in which a name must be unique. An element shown in a package is most likely defined in that package if it is simply named. For example, the Work Units package in Figure 3-41 defines one public class named UnitOfWork. You can show the UnitOfWork class in other packages to indicate that those packages use the class, in which case you qualify the class name with the path to the package in which the class is defined. Such a fully qualified name is referred to as a pathname. The *pathname* of an element is a sequence of package names linked together and separated by double colons (::), followed by the name of the element. The sequence of package names starts from the outermost package level and works its way down to the package containing the element in question. Figure 3-41 shows that the Data package uses the following:

`Workers::Generic::Worker`

: The Worker class located inside the Generic package, which is nested inside the Workers package

`Work Units::UnitOfWork`

: The UnitOfWork class located inside the Work Units package

`Work Products::WorkProduct`

: The WorkProduct class located inside the Work Products package

Subsystems

Recall that a system is an organized collection of elements that may be recursively decomposed into smaller subsystems and eventually into nondecomposable primitive elements. For example, the project management system may be decomposed into the following:

- A user interface subsystem responsible for providing a user interface through which users may interact with the system
- A business processing subsystem responsible for implementing business functionality
- A data subsystem responsible for implementing data storage functionality

The primitive elements would be the various classes that are used in these subsystems and ultimately in the whole system. While a package simply groups elements, a subsystem groups elements that together provide services such that other elements may access only those services and none of the elements themselves. And while packages allow us to partition our system into logical groups and relate these logical groups, subsystems allow us to consider what services these logical groups provide to one another.

A subsystem is shown as a package marked with the `subsystem` keyword. The large package rectangle may have three standard compartments shown by dividing the rectangle with a vertical line and then dividing the area to the left of this line into two compartments with a horizontal line. Figure 3-42 shows how a `Data` subsystem for our project management system might look. The subsystem's operations, specification elements, and interfaces describe the services the subsystem provides, and are the only services accessible by other elements outside the subsystem.

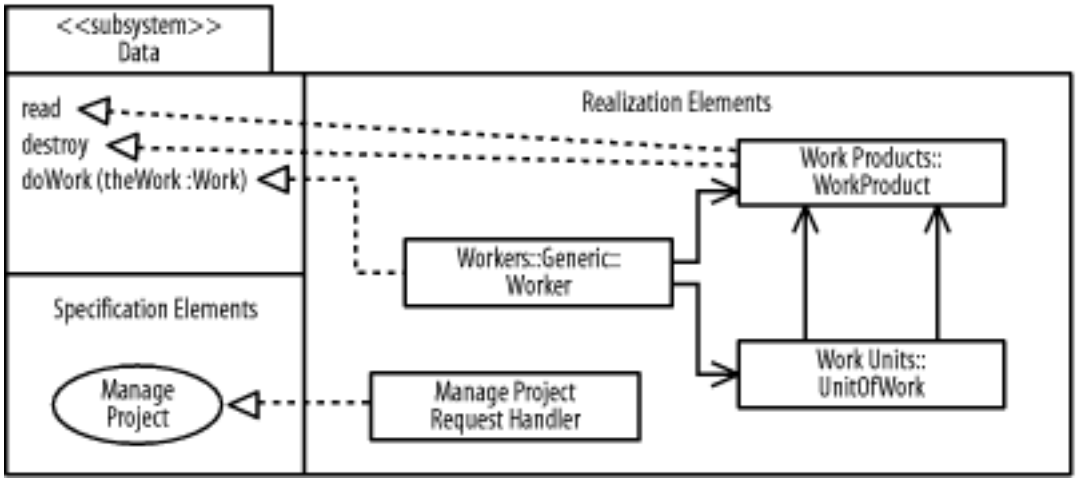

Figure 3-42. A subsystem's representation in the UML

The upper-left compartment shows a list of operations that the subsystem realizes. The lower-left compartment may be labeled "Specification Elements" and shows specification elements that the subsystem realizes. For example, any use cases that the subsystem provides are specification elements that the subsystem must realize. The right compartment may be labeled "Realization Elements" and shows elements inside the subsystem that realize the subsystem's operations and specification elements as well as any interfaces that the subsystem provides. You can modify this general notation by rearranging compartments, combining compartments, or completely suppressing one or more compartments. Any element may be used as a specification or realization element, because a realization simply indicates that the realization element supports at least all the operations of the specification element without necessarily having to support any attributes or associations of the specification element.

Figure 3-43 uses subsystems to refine Figure 3-41. The `Business Processing` and `Data` packages from Figure 3-41 are now subsystems. The `Business Processing` subsystem provides an interface that is used by the `User Interface` package. The `Business Processing` subsystem itself uses the `Data` subsystem and the `IProducible` interface provided by the `Data` subsystem. The `Data` subsystem realizes the `IProducible` interface, which is outside the subsystem itself, various operations, and the `Manage Project` use case that was discussed in Chapter 2. The use case is the oval in the specification element's compartment. The realization elements of the `Data` subsystem realize the `read`, `destroy`, and `doWork` operations, the use case, and the operations of the `IProducible` interface.

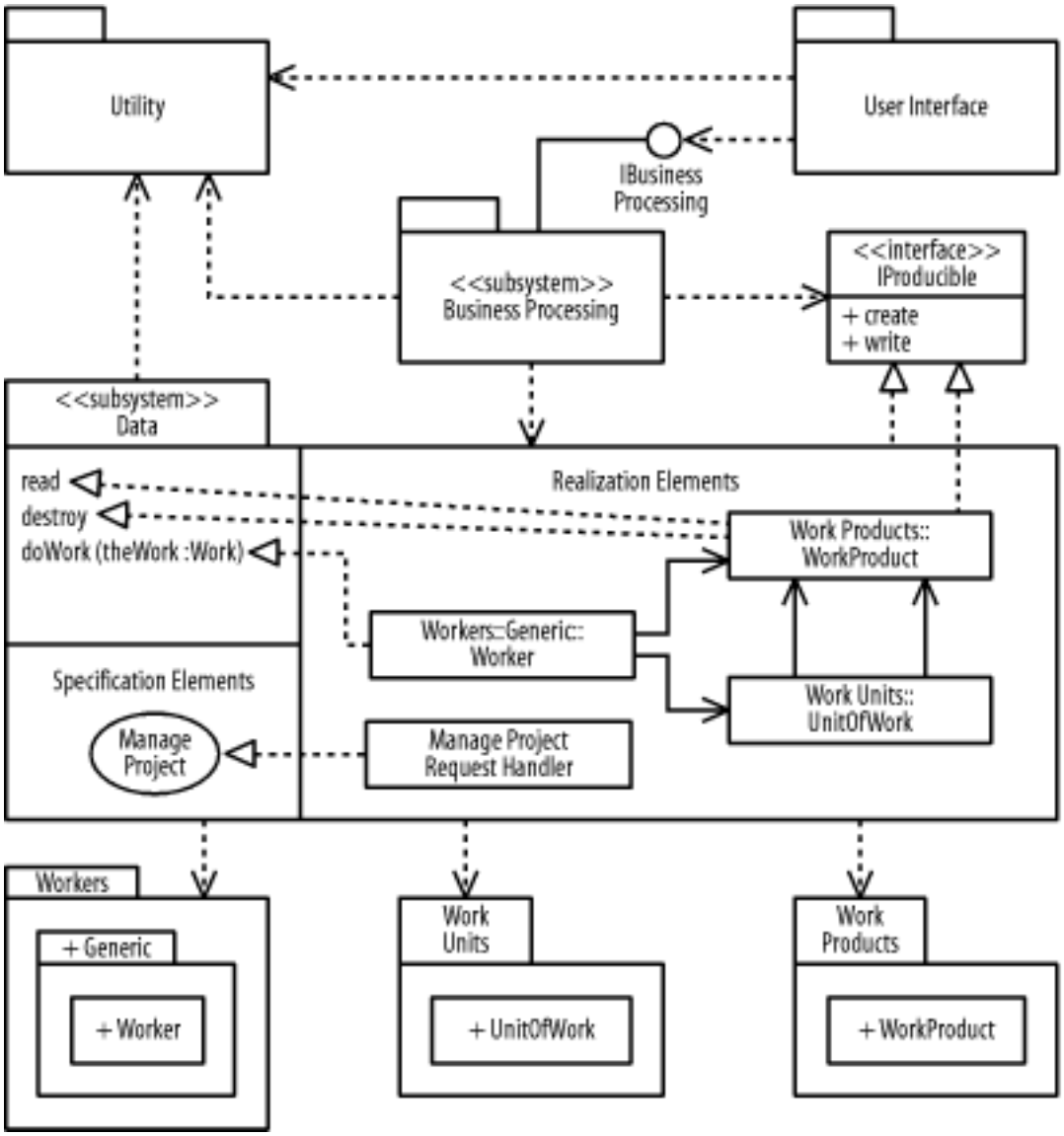

Figure 3-43. Subsystems

Notice in Figure 3-43 that the `User Interface` package does not use the `Business Processing` package as shown in Figure 3-41, but instead uses the `IBusinessProcessing` interface provided by the `Business Processing` subsystem. This use of an interface allows us to focus on the service that the `User Interface` package uses rather than on the package providing the service. If you focus on the fact that the `User Interface` package uses the `Business Processing` package, you are unaware of the exact operations the `User Interface` package requires from the `Business Processing` package; if there is a change to the `Business Processing` package, you would have to expend significant effort to determine whether the change requires modifying the `User Interface` package. By focusing on the service that the `User Interface` package uses rather than the package providing the service, you are aware of the exact operations used from the package providing the service; only if there is a change to the service or to those operations do you have to expend effort to determine whether

the change requires modifying the `User Interface` package. Rather than having to consider all the operations available inside the `Business Processing` package to determine whether changes to that package impact the `User Interface` package, you need only look at a subset of those operations: the subset defined by the `IBusinessProcess` interface. Similarly, notice how the `Business Processing` package uses the `Data` package in Figure 3-41, but the `Business Processing` subsystem uses the operations, specification elements, and `IProducible` interface provided by the `Data` subsystem in Figure 3-43.

Figure 3-43 shows the major elements that make up the project management system and the relationships between them. Using packages, subsystems, interfaces, and their relationships, you can more readily consider which elements may be developed in parallel, which elements may be purchased rather than built, and which elements may be reused. It is possible to address these issues with classes and their relationships, but because a system may have many classes, it can easily become overwhelming to work with such granularity. You could also address these issues by using packages and their dependencies, but packages don't offer services. Packages simply capture the major elements that make up a system and not the services that are being used from a package. Thus, you must focus on all the contents of a package rather than on the services used by elements that depend on the package. However, by using packages, subsystems, interfaces, and their relationships, you can more readily address the issues listed earlier, because you capture the major elements making up the system, as well as the services that are provided and required for these elements to work together to provide the functionality of the system.

Because a subsystem's operations, specification elements, and interfaces describe the services the subsystem provides, which are the only services accessible by other elements outside the subsystem, any collection of subsystems may be developed in parallel, because any interdependencies between them rely only on their services. For example, you may develop the `Data` subsystem and `Business Processing` subsystem in parallel, because any elements that use these subsystems always use the defined services.

Because a subsystem's services are fully specified, you can attempt to search for and purchase (rather than build) a subsystem that provides the same services. For example, you may not have enough funding to build the `Data` subsystem, but because you know the services of the subsystem, you can attempt to search and purchase a similar subsystem that offers the same services.

Because a subsystem's services are fully specified, you can reuse the subsystem whenever you require similar services. For example, whenever you require services defined by the `IBusinessProcessing` interface, you can reuse any subsystem that provides the interface; whenever you require services defined by the `IProducible` interface, the `read` operation, the `destroy` operation, the `doWork` operation, or the `Manage Project` use case, you can reuse any subsystem that provides any of these services.