CHAPTER 4

Modeling a System's Logical Structure: Introducing Classes and Class Diagrams

Classes are at the heart of any object-oriented system; therefore, it follows that the most popular UML diagram is the class diagram. A system's structure is made up of a collection of pieces often referred to as *objects*. Classes describe the different types of objects that your system can have, and class diagrams show these classes and their relationships. Class relationships are covered in Chapter 5.

Use cases describe the behavior of your system as a set of concerns. Classes describe the different types of objects that are needed within your system to meet those concerns. Classes form part of your model's logical view, as shown in Figure 4-1.

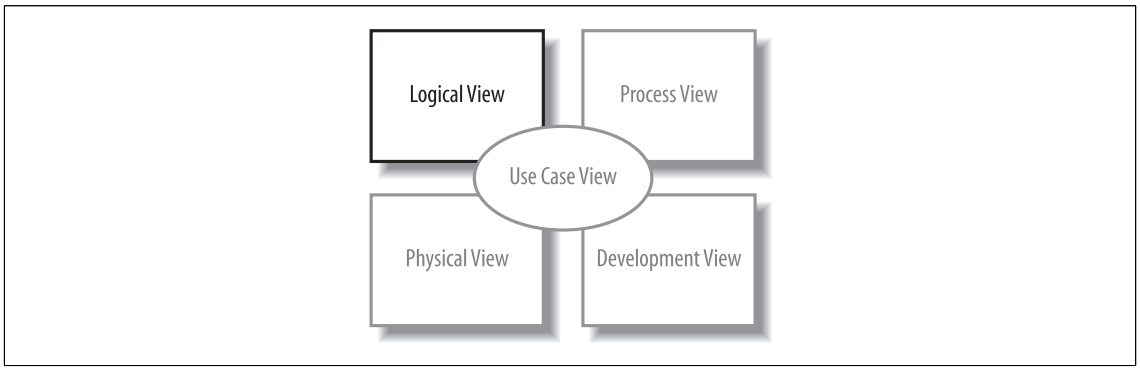


Figure 4-1. The Logical View on your model contains the abstract descriptions of your system's parts, including classes

What Is a Class?

Like any new concept, when first coming to grips with what classes are, it's usually helpful to start with an analogy. The analogy we'll use here is that of guitars, and my favorite guitar is the Burns Brian May Signature (BMS) guitar, shown in Figure 4-2.

The guitar in Figure 4-2 is an example of an object. It has an identity: it's the one I own. However, I'm not going to pretend that Burns made only one of this type of

Figure 4-2. One of my guitars: a good example of an object

guitar and that it was just for me—I'm not that good a guitarist! Burns as a company will make hundreds of this *type* of guitar or, to put it another way, this *class* of guitar.

A class is a type of something. You can think of a class as being the blueprint out of which objects can be constructed, as shown in Figure 4-3.

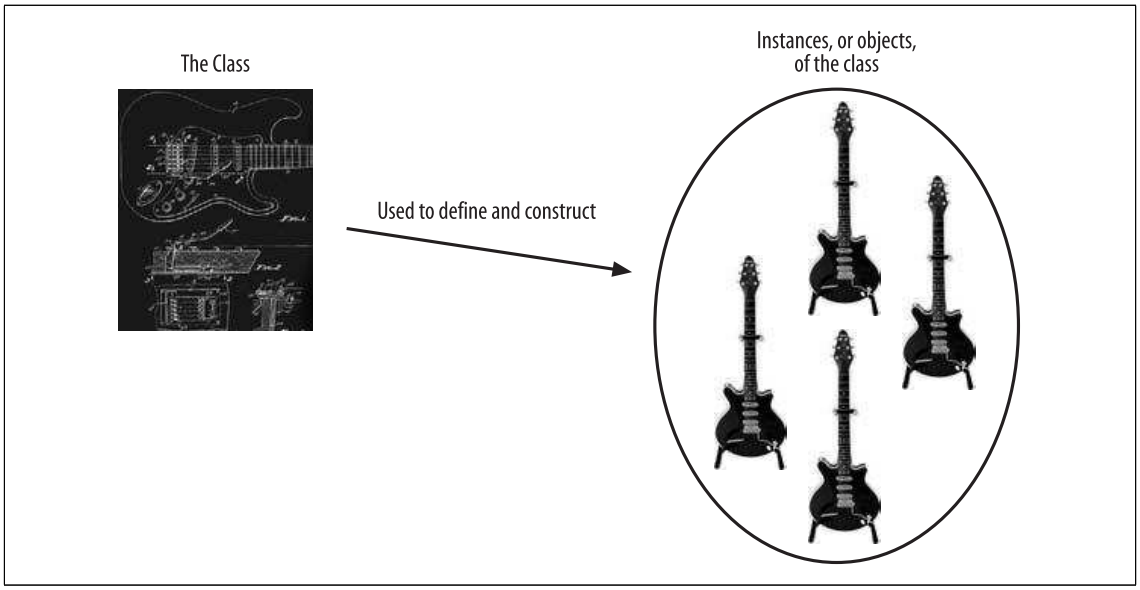


Figure 4-3. The class defines the main characteristics of the guitar; using the class, any number of guitar objects can be constructed

In this analogy, the BMS guitar that Burns manufactures is an example of a class of guitar. Burns know how to build this type of guitar from scratch based on its blueprints. Each guitar constructed from the class can be referred to as an *instance* or

object of the class, and so my guitar in Figure 4-2 is an instance of the `Burns BMS Guitar` class.

At its simplest, a class's description will include two pieces of information: the state information that objects of the class will contain and the behavior that they will support. This is what differentiates OO from other forms of system development. In OO, closely related state and behavior are combined into class definitions, which are then used as the blueprints from which objects can be created.

In the case of the `Burns BMS Guitar` class, the class's state could include information about how many strings the guitar has and what condition the guitar is in. Those pieces of information are the class's attributes.

To complete the description, we need to know what the guitar can do. This includes behavior such as tuning and playing the guitar. A class's behavior is described as the different operations that it supports.

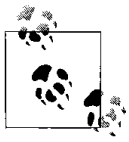
Attributes and operations are the mainstays of a class's description (see "Class State: Attributes"). Together, they enable a class to describe a group of parts within your system that share common characteristics such as state—represented by the class's attributes—and behavior—represented by the class's operations (see "Class Behavior: Operations" later in this chapter).

Abstraction

A class's definition contains the details about that class that are important to you and the system you are modeling. For example, my BMS guitar might have a scratch on the back—or several—but if I am creating a class that will represent BMS guitars, do I need to add attributes that contain details about scratches? I might if the class were to be used in a repair shop; however, if the class were to be used only in the factory system, then scratches are one detail that I can hopefully ignore. Discarding irrelevant details within a given context is called *abstraction*.

Let's have a look at an example of how a class's abstraction changes depending on its context. If Burns were creating a model of its guitar production system, then it would probably be interested in creating a `Burns BMS Guitar` class that models how one is constructed, what materials are to be used, and how the guitar is to be tested. In contrast, if a Guitar World store were creating a model of its sales system, then the `Burns BMS Guitar` class might contain only relevant information, such as a serial number, price, and possibly any special handling instructions.

Getting the right level of abstraction for your model, or even just for a class, is often a real challenge. Focus on the information that your system needs to know rather than becoming bogged down with details that may be irrelevant to your system. You will then have a good starting point when designing your system's classes.



Abstraction is key not only to class diagrams but to modeling in general. A model, by definition, is an abstraction of the system that it represents. The actual system *is* the real thing; the model contains only enough information to be an *accurate representation* of the actual system. In most cases, the model abstracts away details that are not important to the accuracy of the representation.

Encapsulation

Before we take a more detailed look at attributes, operations, and how classes can work together, it's worth focusing on what is the most important characteristic of classes and object orientation: *encapsulation*.

According to the object-oriented approach to system development, for an object to be an object, it needs to contain both data—attributes—and the instructions that affect the data—operations. This is *the* big difference between object orientation and other approaches to system development: in OO, there is the concept of an object that contains, or encapsulates, both the data *and* the operations that work on that data.

Referring back to the guitar analogy, the `Burns BMS Guitar` class could encapsulate its strings, its body, its neck, and probably some neat electrics that no one should mess around with. These parts of the guitar are effectively its attributes, and some of the attributes, such as the strings, are accessible to the outside world and others, such as electrics, are hidden away. In addition to these attributes, the `Burns BMS Guitar` class will contain some operations that will allow the outside world to work with the guitar's attributes. At a minimum, the guitar class should at least have an operation called `play` so that the guitar objects can be played, but other operations such as `clean` and possibly even `serviceElectrics` may also be encapsulated and offered by the class.

Encapsulation of operations and data within an object is probably the single most powerful and useful part of the object-oriented approach to system design. Encapsulation enables a class to hide the inner details of how it works from the outside world—like the electrics from the example guitar class—and only expose the operations and data that it chooses to make accessible.

Encapsulation is very important because with it, a class can change the way it works internally and as long as those internals are not visible to the rest of the system, those changes will have no effect on how the class is interacted with. This is a useful feature of the object-oriented approach because with the right classes, small changes to how those classes work internally shouldn't cause your system to break.

Getting Started with Classes in UML

So far we've been looking at what a class is and how it enables the key benefits of the object-oriented approach of system development: abstraction and encapsulation. Now it's time to take a look at how classes are represented in UML.

At its simplest, a class in UML is drawn as a rectangle split into up to three sections. The top section contains the name of the class, the middle section contains the attributes or information that the class contains, and the final section contains the operations that represent the behavior that the class exhibits. The attributes and operations sections are optional, as shown in Figure 4-4. If the attributes and operations sections are not shown, it does not necessarily imply that they are empty, just that the diagram is perhaps easier to understand with that information hidden.

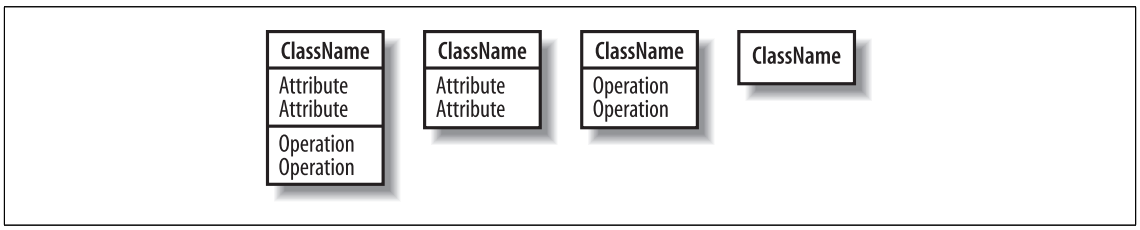


Figure 4-4. Four different ways of showing a class using UML notation

A class's name establishes a type for the objects that will be instantiated based on it. Figure 4-5 shows a couple of classes from the CMS in Chapter 2: the `BlogAccount` class defines the information that the system will hold relating to each of the user's accounts, and the `BlogEntry` class defines the information contained within an entry made by a user into her blog.

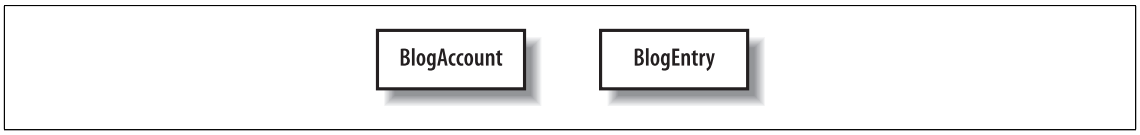


Figure 4-5. Two classes of objects have been identified in the CMS

The interaction diagrams covered in Chapters 7 through 10 are used to show how class instances, or objects, work together when a system is running.

Visibility

How does a class selectively reveal its operations and data to other classes? By using *visibility*. Once visibility characteristics are applied, you can control access to attributes, operations, and even entire classes to effectively enforce encapsulation. See "Encapsulation" earlier in this chapter for more information on why encapsulation is such a useful aspect of object-oriented system design.

There are four different types of visibility that can be applied to the elements of a UML model, as shown in Figure 4-6. Typically these visibility characteristics will be used to control access to both attributes, operations, and sometimes even classes (see the "Packages" section in Chapter 13 for more information on class visibility).

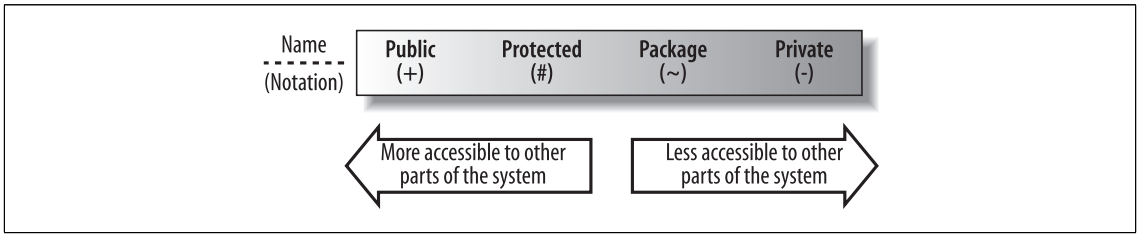


Figure 4-6. UML's four different visibility classifications

Public Visibility

Starting with the most accessible of visibility characteristics, *public visibility* is specified using the plus (+) symbol before the associated attribute or operation (see Figure 4-7). Declare an attribute or operation public if you want it to be accessible directly by any other class.

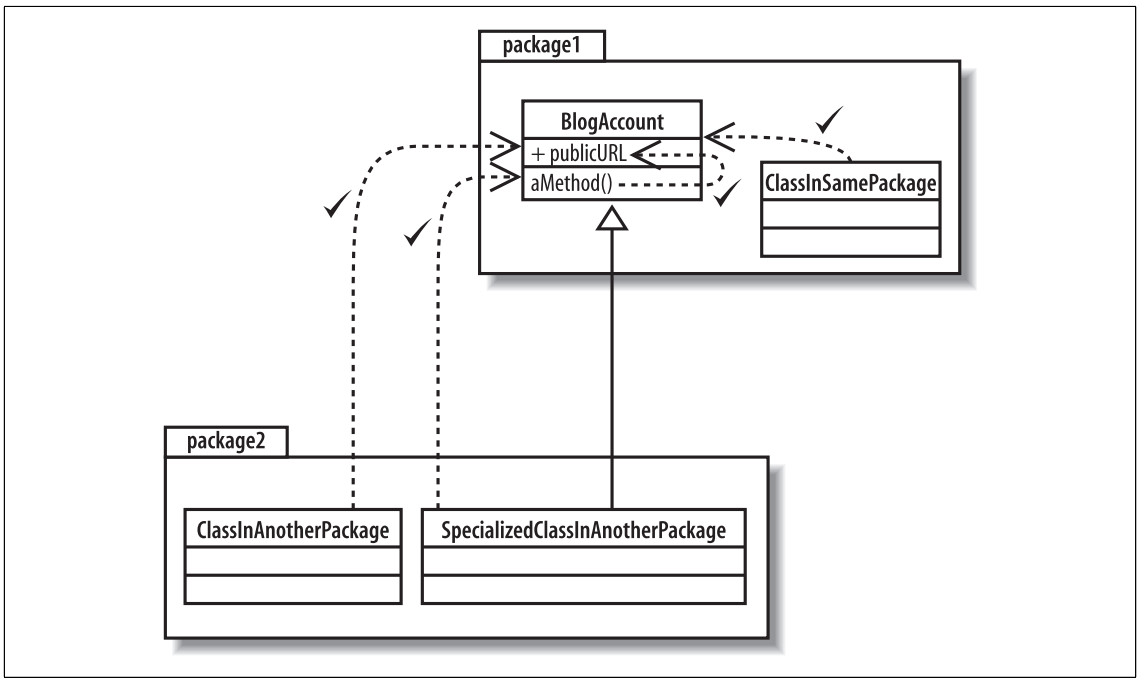


Figure 4-7. Using public visibility, any class within the model can access the publicURL attribute

The collection of attributes and operations that are declared public on a class create that class's public interface. The *public interface* of a class consists of the attributes and operations that can be accessed and used by other classes. This means the public interface is the part of your class that other classes will depend on the most. It is

Public Attributes

To have public attributes or to not have public attributes? That is the question. Many object-oriented designers groan at the use of public attributes: opening a class's attributes to the rest of the system is like exposing your house to any person off the street without requiring him to check with you before entering. There is just as much potential for abuse.

It's usually best to avoid public attributes, but there are always exceptions to the rule. One example where it is generally accepted to use a public attribute is when the attribute is a constant that may be used by a number of different classes. Attributes that act as constants, i.e., to be given an initial unchangeable value are given the property of `readOnly` (see "Attribute Properties"). In this situation, exposing the attribute to the rest of your system is not quite so dangerous since its value cannot be changed.

important that the public interface to your classes changes as little as possible to prevent unnecessary changes wherever your class is used.

Protected Visibility

Protected attributes and operations are specified using the hash (#) symbol and are more visible to the rest of your system than private attributes and operations, but are less visible than public. Declared protected elements on classes can be accessed by methods that are part of your class and also by methods that are declared on any class that inherits from your class. Protected elements cannot be accessed by a class that does not inherit from your class whether it's in the same package or not, as shown in Figure 4-8. See Chapter 5 for more information on inheritance relationships between classes.

Protected visibility is crucial if you want to allow specialized classes to access an attribute or operation in the base class without opening that attribute or operation to the entire system. Using protected visibility is like saying, "This attribute or operation is useful inside my class and classes extending my class, but no one else should be using it."

Java confuses the matter a little further by allowing access to protected parts of a class to any other class in the same package. This is like combining the accessibility of protected and package visibility, which is covered in the next section.

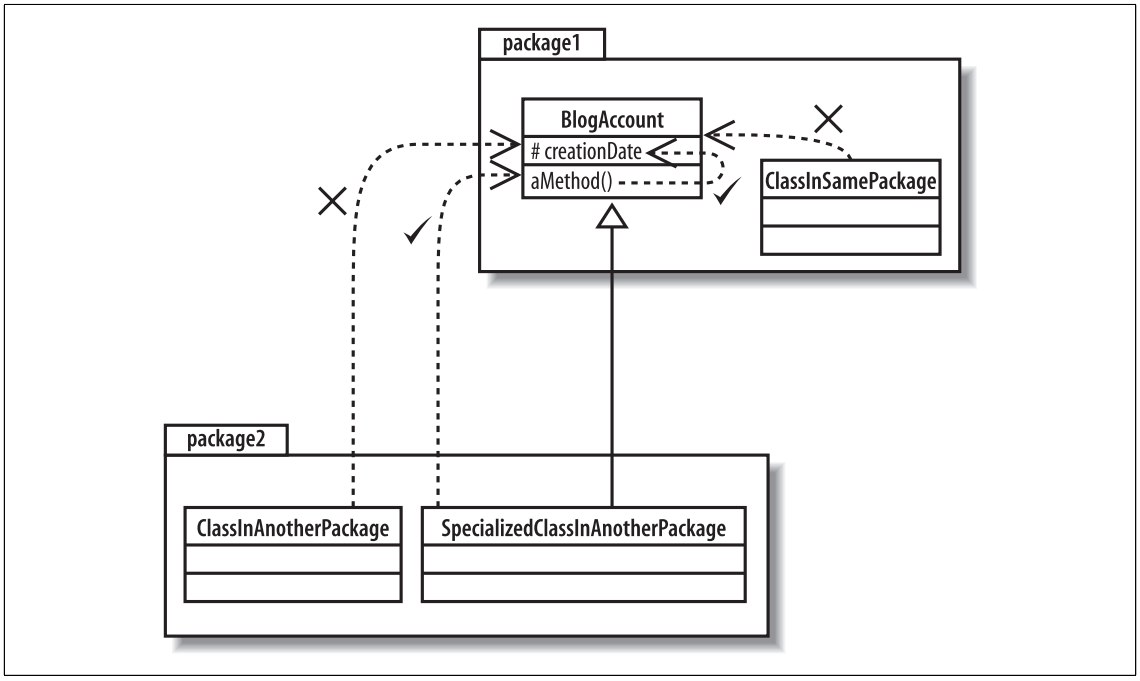


Figure 4-8. Any methods in the BlogAccount class or classes that inherit from the BlogAccount class can access the protected creationDate attribute

Package Visibility

Package visibility, specified with a tilde (~), when applied to attributes and operations, sits in between protected and private. As you'd expect, packages are the key factor in determining which classes can see an attribute or operation that is declared with package visibility.

The rule is fairly simple: if you add an attribute or operation that is declared with package visibility to your class, then any class in the same package can directly access that attribute or operation, as shown in Figure 4-9. Classes outside the package cannot access protected attributes or operations even if it's an inheriting class.In practice, package visibility is most useful when you want to declare a collection of methods and attributes across your classes that can only be used within your package.

For example, if you were designing a package of utility classes and wanted to reuse behavior between those classes, but not expose the rest of the system to that behavior, then you would declare package visibility to those particular operations internally to the package. Any functionality of utility classes that you wanted to expose to the rest of the application could then be declared with public visibility.

See "Package Diagrams" in Chapter 13 for more on how packages control visibility of elements such as classes.

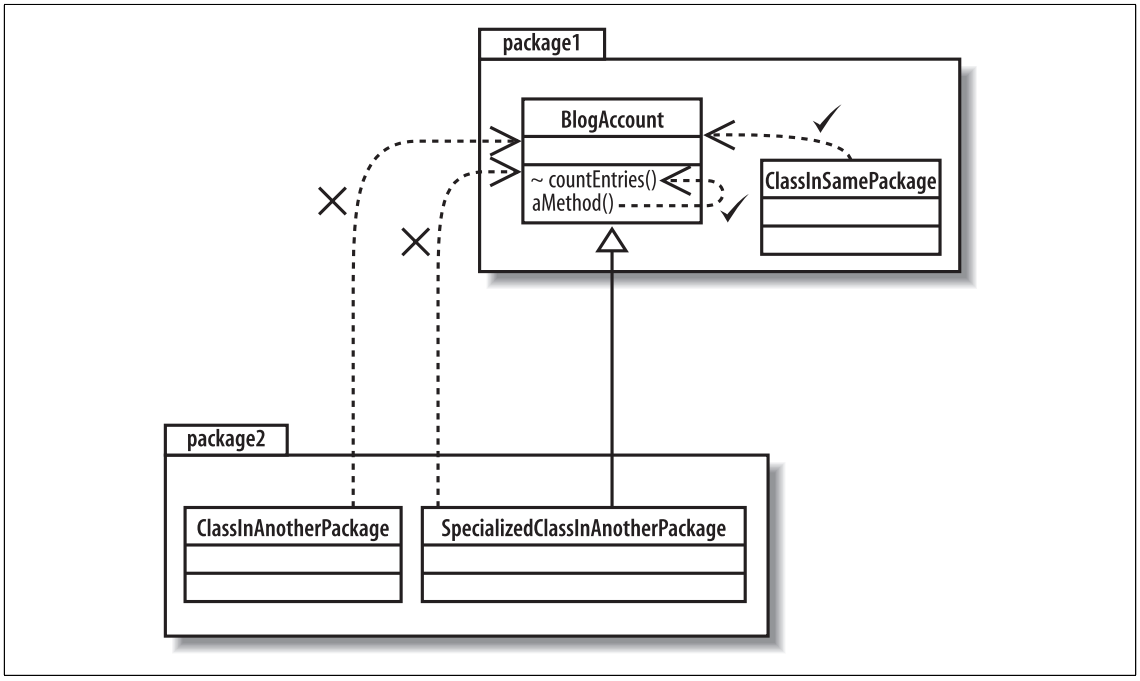


Figure 4-9. The countEntries operation can be called by any class in the same package as the BlogAccount class or by methods within the BlogAccount class itself

Private Visibility

Last in line in the UML visibility scale is private visibility. *Private visibility* is the most tightly constrained type of visibility classification, and it is shown by adding a minus (-) symbol before the attribute or operation. Only the class that contains the private element can see or work with the data stored in a private attribute or make a call to a private operation, as shown in Figure 4-10.

Private visibility is most useful if you have an attribute or operation that you want no other part of the system to depend on. This might be the case if you intend to change an attribute or operation at a later time but don't want other classes with access to that element to be changed.

It's a commonly accepted rule of thumb that attributes should always be private and only in extreme cases opened to direct access by using something more visible. The exception to this rule is when you need to share your class's attribute with classes that inherit from your class. In this case, it is common to use `protected`. In well-designed OO systems, attributes are usually `private` or `protected`, but very rarely `public`.

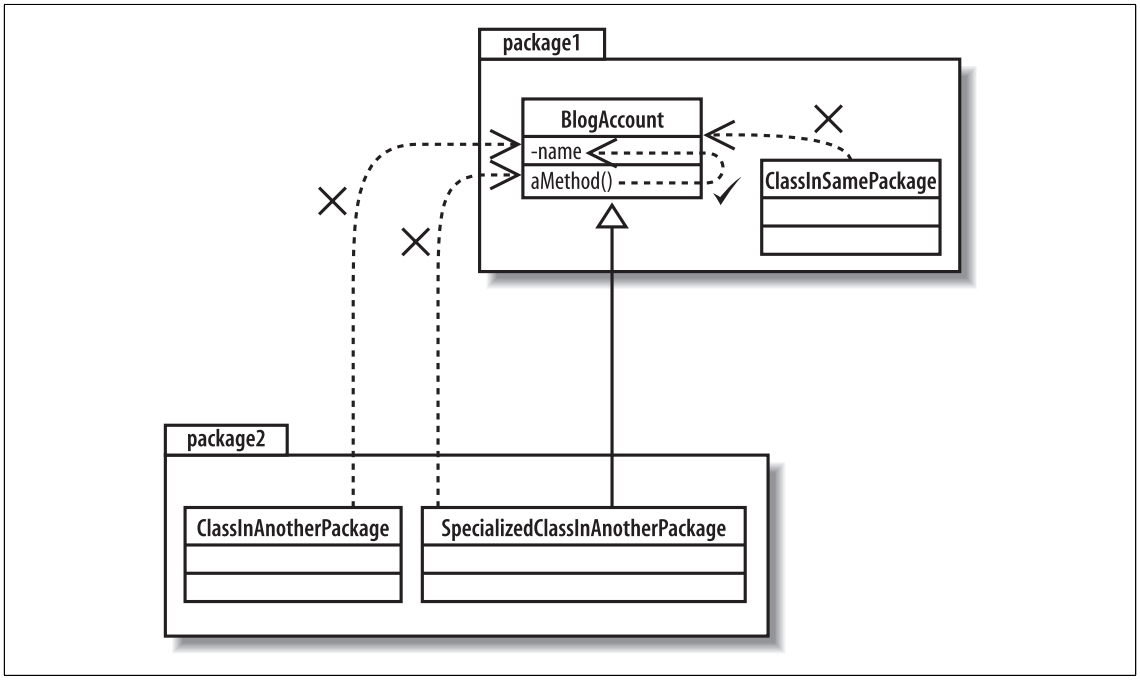


Figure 4-10. aMethod is part of the BlogAccount class, so it can access the private name attribute; no other class's methods can see the name attribute

Class State: Attributes

A class's *attributes* are the pieces of information that represent the state of an object. These attributes can be represented on a class diagram either by placing them inside their section of the class box—known as inline attributes—or by association with another class, as shown in Figure 4-11. Associations are covered in more detail in Chapter 5.

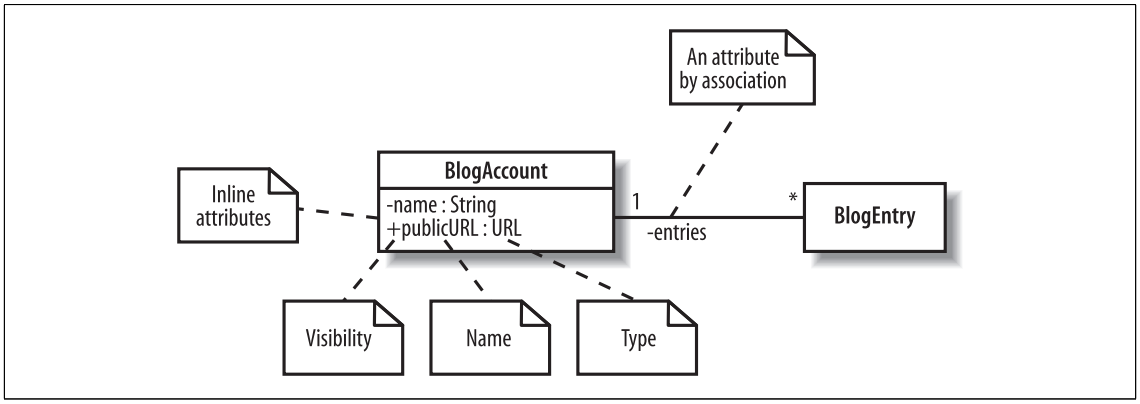


Figure 4-11. The BlogAccount class contains two inlined attributes, name and publicURL, as well as an attribute that is introduced by the association between the BlogAccount and BlogEntry classes

It doesn't matter if you are declaring an inline or associated attribute. At a minimum, your attribute will usually have a signature that contains a visibility property, a

name, and a type, although the attribute's name is the only part of its signature that absolutely must be present for the class to be valid.

Name and Type

An attribute's name can be any set of characters, but no two attributes in the same class can have the same name. The type of attribute can vary depending on how the class will be implemented in your system but it is usually either a class, such as String, or a primitive type, such as an `int` in Java.

Choosing Attribute Names

Remember, one of the primary aims of modeling your system is to communicate your design to others. When picking names of attributes, operations, classes, and packages, make sure that the name accurately describes what is being named. When naming attributes, it's worth trying to come up with a name that describes the information that the attribute represents.

Also, if your class is to be implemented in a specific software language, check to make sure that the name meets the conventions of that language. In Java, it is common to use an uppercase character for each word in your class's names, e.g., BlogAccount, while Java packages are usually named all in lowercase (see Chapter 13).

In Figure 4-11, the `name` attribute is declared as private (indicated by the minus (-) sign at the beginning of the signature) and after the colon, the type is specified as being of the class `String`. The associated `entries` attribute is also private, and because of that association, it represents a number of instances of the `BlogEntry` class.

If the `BlogAccount` class in Figure 4-11 was going to be implemented as a Java class in software, then the source code would look something like that shown in Example 4-1.

Example 4-1. Java inline and by-association attributes

```
public class BlogAccount
{
   // The two inline attributes from Figure 4-11.
   private String name;
   private URL publicURL;

   // The single attribute by association, given the name 'entries'
   BlogEntries[] entries;

   // ...

}
```

It's pretty clear how the two inline attributes are implemented in the `BlogAccount` Java class; the `name` attribute is just a Java String and the `publicURL` attribute is a Java URL object. The `entries` attribute is a bit more interesting since it is introduced by association. Associations and relationships between classes are covered in Chapter 5.

Multiplicity

Sometimes an attribute will represent more than one object. In fact, an attribute could represent any number of objects of its type; in software, this is like declaring that an attribute is an array. Multiplicity allows you to specify that an attribute actually represents a collection of objects, and it can be applied to both inline and attributes by association, as shown in Figure 4-12.

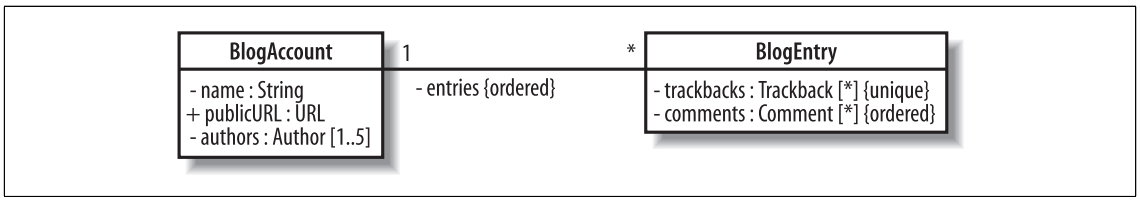


Figure 4-12. Applying several flavors of attribute multiplicity to the attributes of the BlogAccount and BlogEntry classes

In Figure 4-12, the `trackbacks`, `comments`, and `authors` attributes all represent collections of objects. The * at the end of the `trackbacks` and `comments` attributes specifies that they could contain any number of objects of the `Trackback` and `Comment` class, respectively. The `authors` attribute is a little more constrained since it specifies that it contains between one and five authors.

The `entries` attribute that is introduced using an association between the `BlogAccount` class and the `BlogEntry` class has two multiplicity properties specified at either end of the association. A * at the `BlogEntry` class end of the association indicates that any number of `BlogEntry` objects will be stored in the `entries` attribute within the `BlogAccount` class. The `1` specified at the other end of the association indicates that each `BlogEntry` object in the `entries` attribute is associated with one and only one `BlogAccount` object.

Those with a keen eye will have also noticed that the `trackbacks`, `comments`, and `entries` attributes also have extra properties to describe in even more detail what the multiplicity on the attributes means. The `trackbacks` attribute represents any number of objects of the `Trackback` class, but it also has the `unique` multiplicity property applied to it. The `unique` property dictates that no two `Trackback` objects within the array should be the same. This is a reasonable constraint since we don't want an entry in another blog cross-referencing one of our entries more than once; otherwise the list of `trackbacks` will get messy.

By default, all attributes with multiplicity are unique. This means that, as well as the `trackbacks` attribute in the `BlogEntry` class, no two objects in the `authors` attributes collection in the `BlogAccount` class should be the same because they are also declared unique. This makes sense since it specifies that a `BlogAccount` can have up to five *different* authors; however, it wouldn't make sense to specify that the same author represents two of the possible five authors that work on a blog! If you want to specify that duplicates are allowed, then you need to use the `not unique` property, as used on the `comments` attribute in the `BlogEntry` class.

The final property that an attribute can have that is related to multiplicity is the `ordered` property. As well as not having to be unique, the objects represented by the `comments` attribute on the `BlogEntry` class need to be ordered. The `ordered` property is used in this case to indicate that each of the `Comment` objects is stored in a set order, most likely in order of addition to the `BlogEntry`. If you don't care about the order in which objects are stored within an attribute that has multiplicity, then simply leave out the ordered property.

Attribute Properties

As well as visibility, a unique name, and a type, there is also a set of properties that can be applied to attributes to completely describe an attribute's characteristics.

Although a complete description of the different types attribute properties is probably a bit beyond this book—also, some of the properties are rarely used in practice—it is worth looking at what is probably the most popular attribute property: the `readOnly` property.

Other properties supported by attributes in UML include union, subsets, redefines, and composite. For a neat description of all of the different properties that can be applied to attributes, check out *UML 2.0 in a Nutshell* (O'Reilly).

If an attribute has the `readOnly` property applied, as shown in Figure 4-13, then the value of the attribute cannot be changed once its initial value has been set.

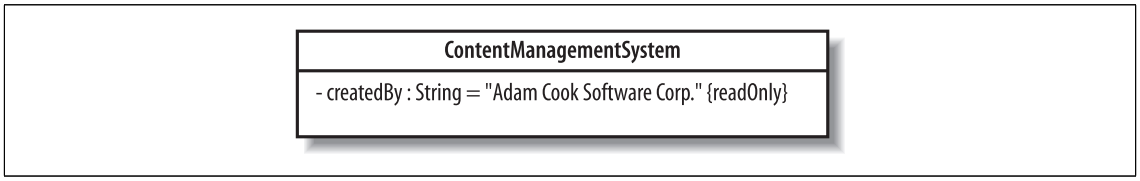


Figure 4-13. The createdBy attribute in the ContentManagementSystem class is given a default initial value and a property of readOnly so that the attribute cannot be changed throughout the lifetime of the system

If the `ContentManagementSystem` class were to be implemented in Java source code, then the `createdBy` attribute would be translated into a `final` attribute, as shown in Example 4-2.

Example 4-2. Final attributes in Java are often referred to as constants since they keep the same constant value that they are initially set up with for their entire lifetime

```
public class ContentManagementSystem
{
   private final String createdBy = "Adam Cook Software Corp.";
}
```

Inline Attributes Versus Attributes by Association

So, why confuse things with two ways of showing a class's attributes? Consider the classes and associations shown in Figure 4-14.

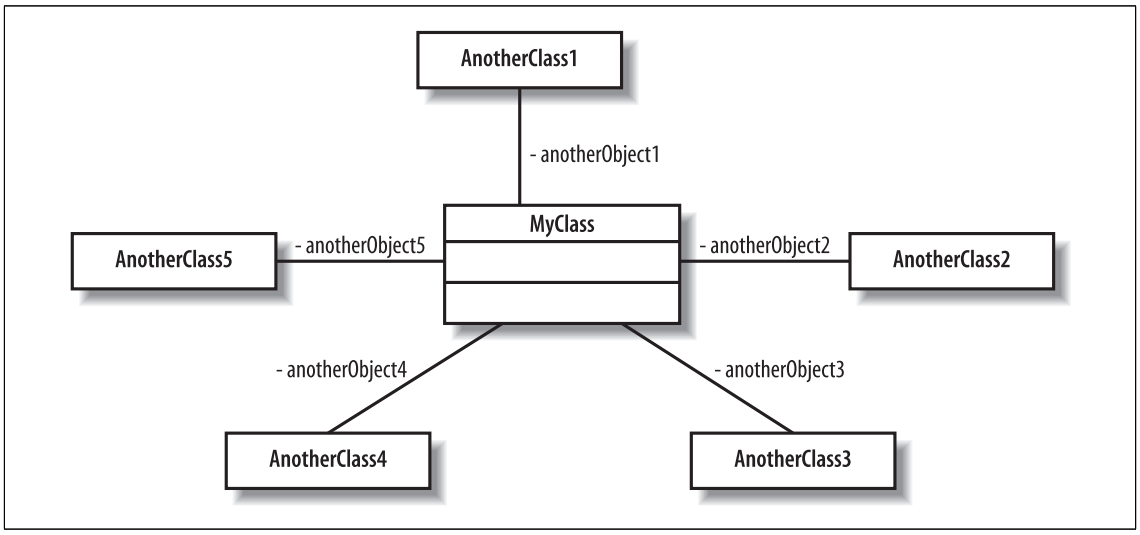


Figure 4-14. The MyClass class has five attributes, and they are all shown using associations

When attributes are shown as associations, as is the case in Figure 4-14, the diagram quickly becomes busy—and that's just to show the associations, nevermind all of the other relationships that classes can have (see Chapter 5). The diagram is neater and easier to manage with more room for other information when the attributes are specified inline with the class box, as shown in Figure 4-15.

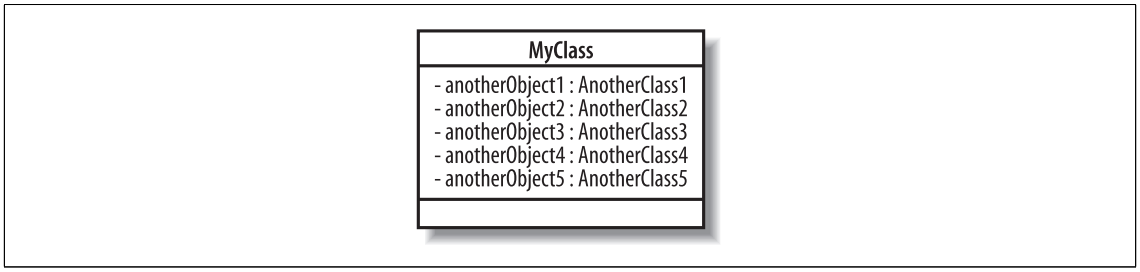
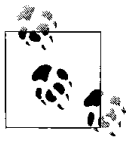


Figure 4-15. The MyClass class's five attributes shown inline within the class box

Choosing whether an attribute should be shown inline or as an association is really a question of what the focus of the diagram should be. Using inline attributes takes the spotlight away from the associations between `MyClass` and the other classes, but is a much more efficient use of space. Associations show relationships between classes very clearly on a diagram but they can get in the way of other relationships, such as inheritance, that are more important for the purpose of a specific diagram.



One useful rule of thumb: "simple" classes, such as the `String` class in Java, or even standard library classes, such as the `File` class in Java's io package, are generally best shown as inline attributes.

Class Behavior: Operations

A class's operations describe *what* a class can do but not necessarily *how* it is going to do it. An operation is more like a promise or a minimal contract that declares that a class will contain some behavior that does what the operation says it will do. The collection of all the operations that a class contains should totally encompass all of the behavior that the class contains, including all the work that maintains the class's attributes and possibly some additional behavior that is closely associated with the class.

Operations in UML are specified on a class diagram with a signature that is at minimum made up of a visibility property, a name, a pair of parentheses in which any parameters that are needed for the operation to do its job can be supplied, and a return type, as shown in Figure 4-16.

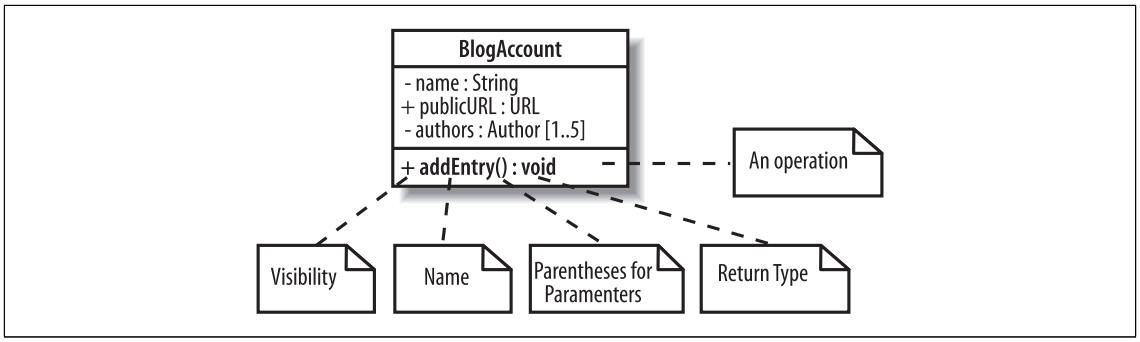


Figure 4-16. Adding a new operation to a class allows other classes to add a BlogEntry to a BlogAccount

In Figure 4-16, the `addEntry` operation is declared as public; it does not require any parameters to be passed to it (yet), and it does not return any values. Although this is a perfectly valid operation in UML, it is not even close to being finished yet. The operation is supposed to add a new `BlogEntry` to a `BlogAccount`, but at the moment, there is no way of knowing what entry to actually add.

Parameters

Parameters are used to specify the information provided to an operation to allow it to complete its job. For example, the `addEntry(..)` operation needs to be supplied with the `BlogEntry` that is to be added to the account, as shown in Figure 4-17.

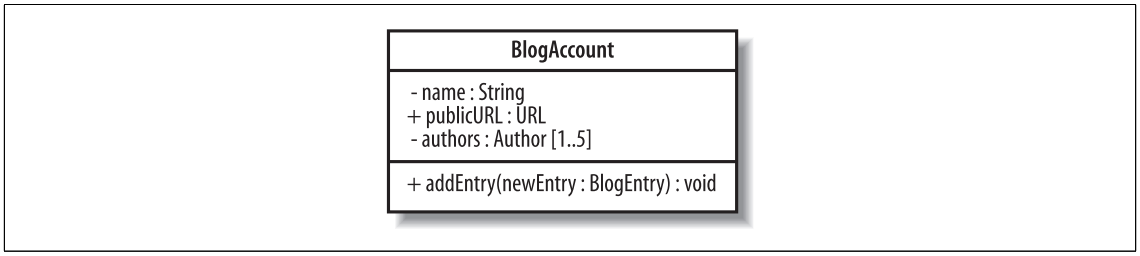


Figure 4-17. Adding a new parameter to the addEntry operation saves a bit of embarrassment when it comes to implementing this class; at least the addEntry operation will now know which entry to add to the blog!

The `newEntry` parameter that is passed to the `addEntry` operation in Figure 4-17 shows a simple example of a parameter being passed to an operation. At a minimum, a parameter needs to have its type specified—in this case, `BlogEntry` class. More than one parameter can be passed to an operation by splitting the parameters with a comma, as shown in Figure 4-18. For more information on all the nuances of parameter notation, see *UML 2.0 in a Nutshell* (O'Reilly).

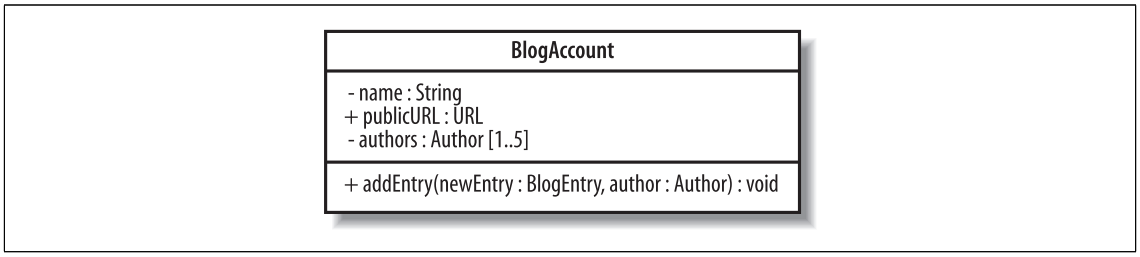


Figure 4-18. As well as passing the new blog entry that is to be added, by adding another parameter, we can also indicate which author wrote the entry

Return Types

As well as a name and parameters, an operation's signature also contains a return type. A return type is specified after a colon at the end of an operation's signature and specifies the type of object that will be returned by the operation, as shown in Figure 4-19.

There is one exception where you don't need to specify a return type: when you are declaring a class's constructor. A constructor creates and returns a new instance of the class that it is specified in, therefore, it does not need to explicitly declare any return type, as shown in Figure 4-20.

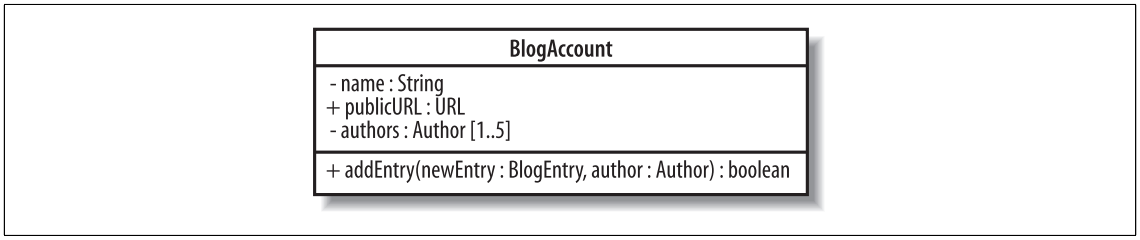


Figure 4-19. The addEntry(..) operation now returns a Boolean indicating whether the entry was successfully added

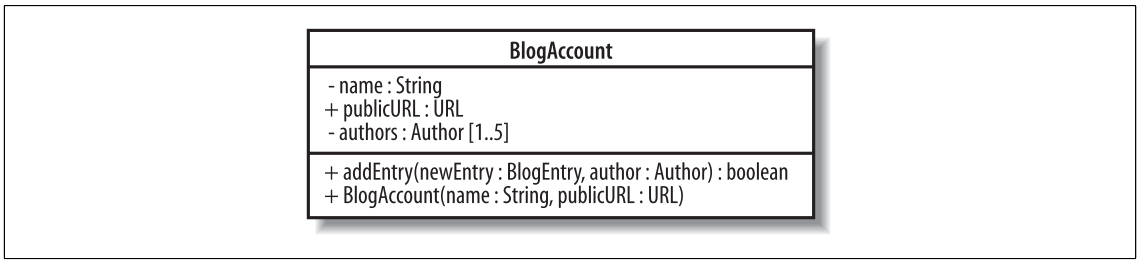


Figure 4-20. The BlogAccount(..) constructor must always return an instance of BlogAccount, so there is no need to explicitly show a return type

Static Parts of Your Classes

To finish off this introduction to the fundamentals of class diagrams, let's take a look at one of the most confusing characteristics of classes: when a class operation or attribute is static.

In UML, operations, attributes, and even classes themselves can be declared static. To help us understand what static means, we need to look at the lifetime of regular non-static class members. First, lets take another look at the `BlogAccount` class from earlier on in this chapter, shown in Figure 4-21.

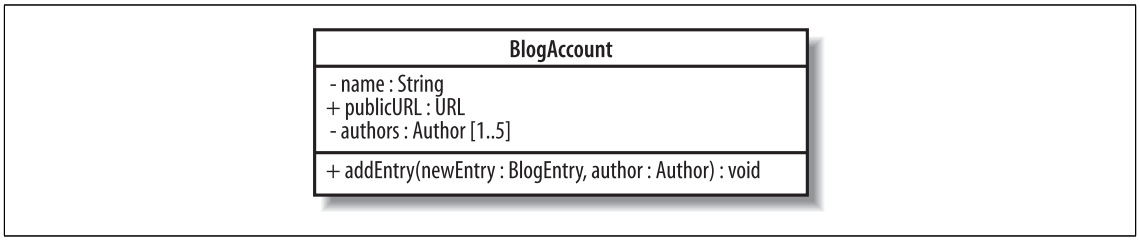


Figure 4-21. The BlogAccount class is made up of three regular attributes and one regular operation

Because each of the attributes and operations on the `BlogAccount` class are non-static, they are associated with instances, or objects, of the class. This means that each object of the `BlogAccount` class will get their own copy of the attributes and operations, as shown in Figure 4-22.

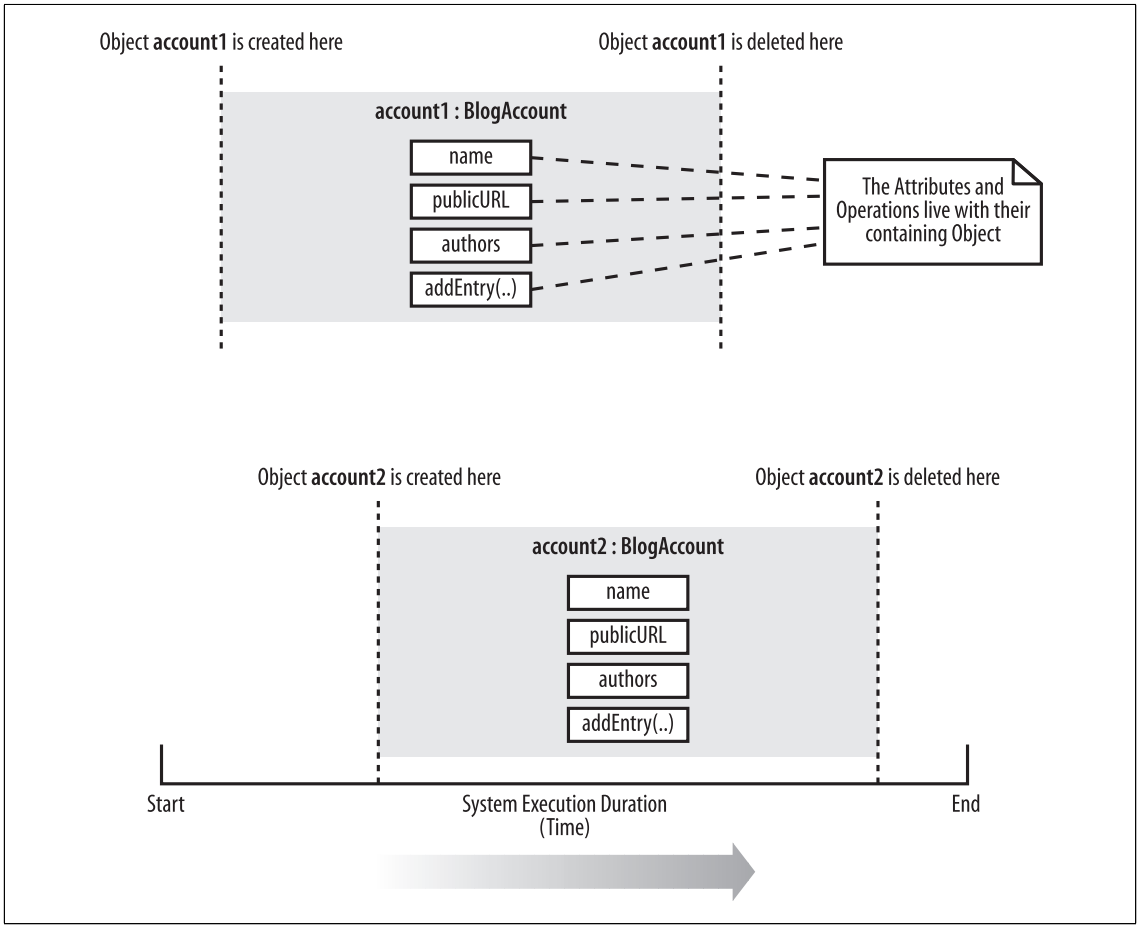


Figure 4-22. Both account1 and account2 contain and exhibit their own copy of all the regular non-static attributes and operations declared on the BlogAccount class

Sometimes you want all of the objects in a particular class to share the same copy of an attribute or operation. When this happens, a class's attributes and operations are associated with the class itself and have a lifetime beyond that of the any objects that are instantiated from the class. This is where static attributes and operations become useful.

For example (and let's ignore the possibility of multiple classloaders for now), if we wanted to keep a count of all the `BlogAccount` objects currently alive in the system, then this counter would be a good candidate for being a static class attribute. Rather than the counter attribute being associated with any one object, it is associated with the `BlogAccount` class and is therefore a static attribute, as shown in Figure 4-23.

The `accountCounter` attribute needs to be incremented every time a new `BlogAccount` is created. The `accountCounter` attribute is declared static because the same copy needs to be shared between all of the instances of the `BlogAccount` class. The instances can increment it when they are created and decrement it when they are destroyed, as shown in Figure 4-24.

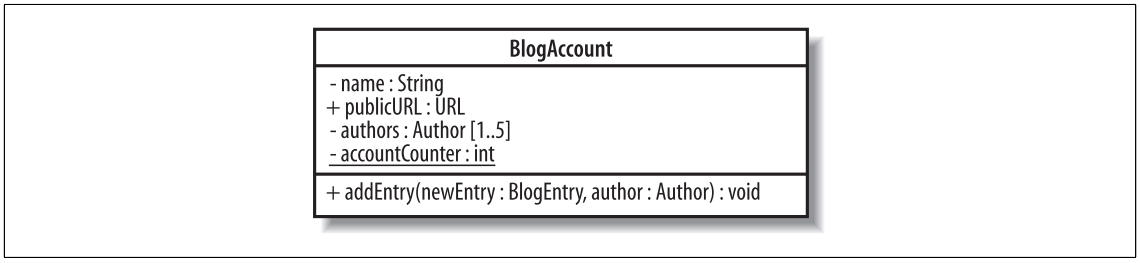


Figure 4-23. An attribute or operation is made static in UML by underlining it; the accountCounter attribute will be used to keep a running count of the number of objects created from the BlogAccount class

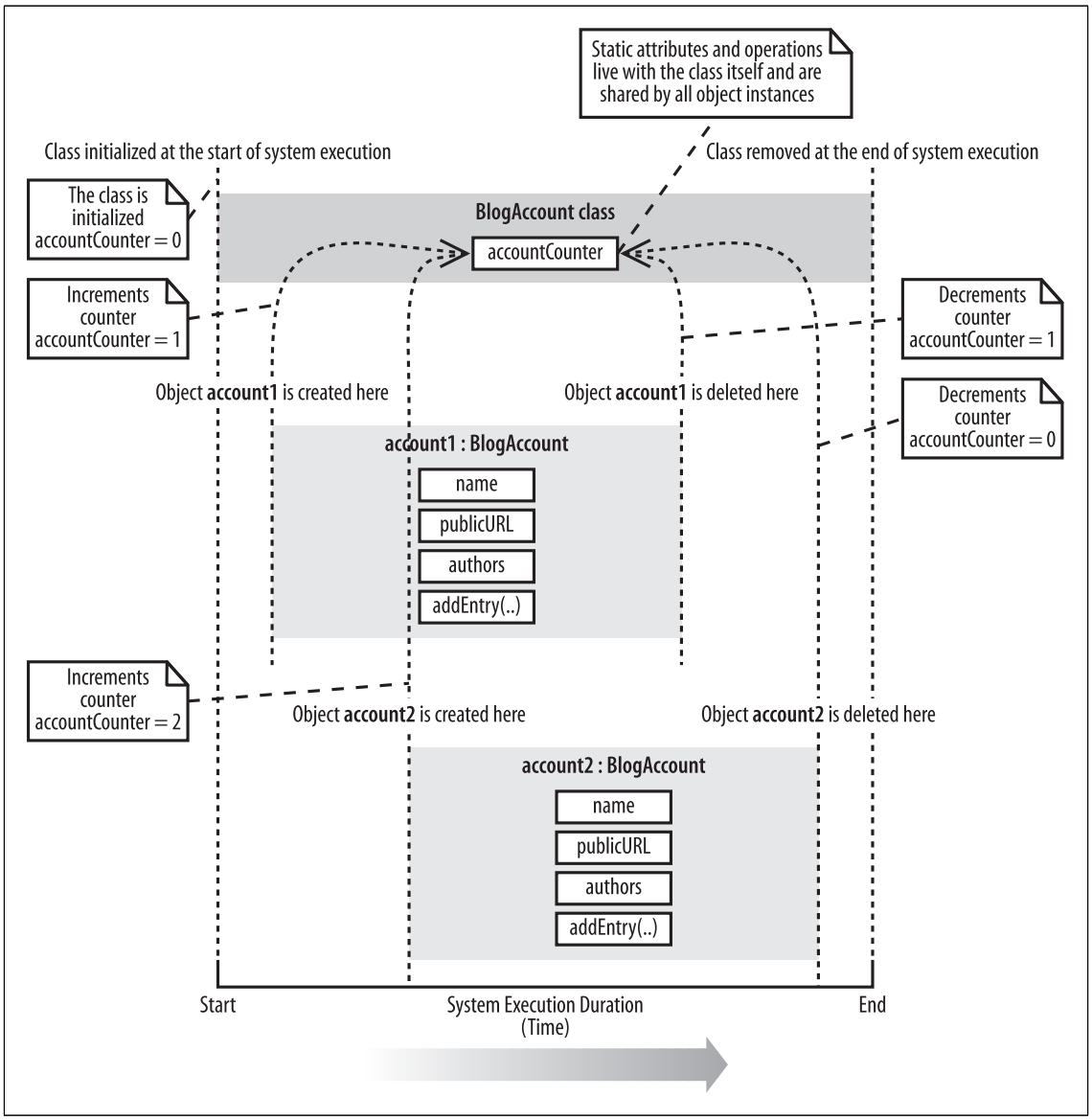


Figure 4-24. The static accountController attribute is shared between the different BlogAccount objects to keep a count of the currently active BlogAccount objects within the system

If the `accountCounter` attribute were not static, then every `BlogAccount` instance would get its own copy of the `accountCounter` attribute. This would not be very useful at all since each `BlogAccount` object would update only its own copy of `accountCounter` rather than contributing to a master object instance counter—in fact, if `accountCounter` were not static, then every object would simply increment its own copy to 1 and then decrement it to 0 when it is destroyed, which is not very useful at all!

The Singleton Design Pattern

Another great example of when static attributes and operations are used is when you want to apply the Singleton design pattern. In a nutshell, the Singleton design pattern ensures that one and only one object of a particular class is ever constructed during the lifetime of your system. To ensure that only one object is ever constructed, typical implementations of the Singleton pattern keep an internal static reference to the single allowed object instance, and access to that instance is controlled using a static operation. To learn more about the Singleton pattern, check out *Head First Design Patterns* (O'Reilly).

What's Next

This chapter has given you only a first glimpse of all that is possible with class diagrams. Classes can be related to one another, and there are even advanced forms of classes, such as templates, that can make your system's design even more effective. Class relationships, abstract classes, and class templates are all covered in Chapter 5.

Class diagrams show the types of objects in your system; a useful next step is to look at object diagrams because they show how classes come alive at runtime as object instances, which is useful if you want to show runtime configurations. Object diagrams are covered in Chapter 6.

Composite structures are a diagram type that loosely shows context-sensitive class diagrams and patterns in your software. Composite structures are described in Chapter 11.

After you've decided the responsibilities of the classes in your system, it's common to then create sequence and communication diagrams to show interactions between the parts. Sequence diagrams can be found in Chapter 7. Communication diagrams are covered in Chapter 8.

It's also common to step back and organize your classes into packages. Package diagrams allow you to view dependencies at a higher level, helping you understand the stability of your software. Package diagrams are described in Chapter 13.

CHAPTER 5

Modeling a System's Logical Structure: Advanced Class Diagrams

If all you could do with class diagrams was declare classes with simple attributes and operations, then UML would be a pretty poor modeling language. Luckily, object orientation and UML allows much more to be done with classes than just simple declarations. For starters, classes can have relationships to one another. A class can be a type of another class—generalization—or it can contain objects of another class in various ways depending on how strong the relationship is between the two classes.

Abstract classes help you to partly declare a class's behavior, allowing other classes to complete the missing—abstract—bits of behavior as they see fit. Interfaces take abstract classes one stage further by specifying only the needed operations of a class but without any operation implementations. You can even apply constraints to your class diagrams that describe how a class's objects can be used with the Object Constraint Language (OCL).

Templates complete the picture by allowing you to declare classes that contain completely generic and reusable behavior. With templates, you can specify what a class will do and then wait—as late as runtime if you choose—to decide which classes it will work with.

Together, these techniques complete your class diagram toolbox. They represent some of the most powerful concepts in object-oriented design and, when applied correctly, can make the difference between an OK design and a *great* piece of reusable design.

Class Relationships

Classes do not live in a vacuum—they work together using different types of relationships. Relationships between classes come in different strengths, as shown in Figure 5-1.

The strength of a class relationship is based on how dependent the classes involved in the relationship are on each other. Two classes that are strongly dependent on one another are said to be *tightly coupled*; changes to one class will most likely affect the

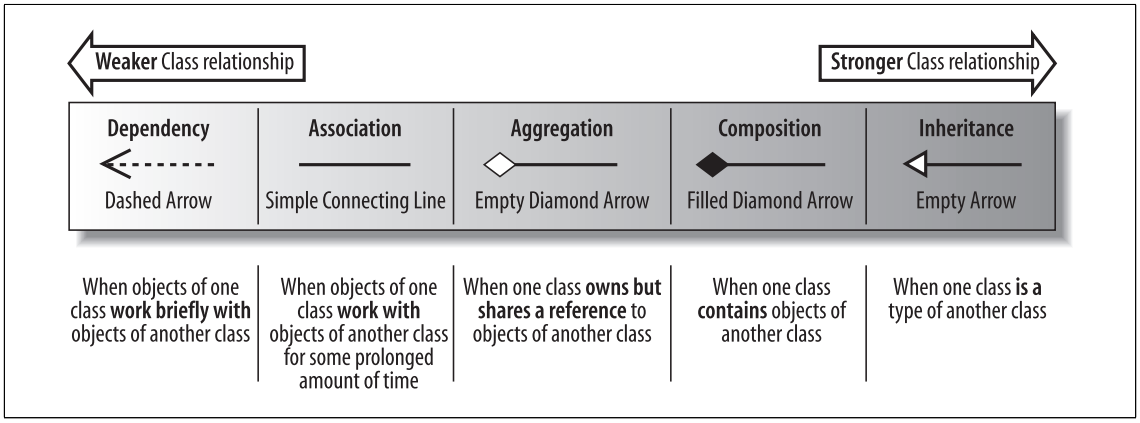


Figure 5-1. UML offers five different types of class relationship

other class. Tight coupling is usually, but not always, a bad thing; therefore, the stronger the relationship, the more careful you need to be.

Dependency

A *dependency* between two classes declares that a class needs to know about another class to use objects of that class. If the `UserInterface` class of the CMS needed to work with a `BlogEntry` class's object, then this dependency would be drawn using the dependency arrow, as shown in Figure 5-2.

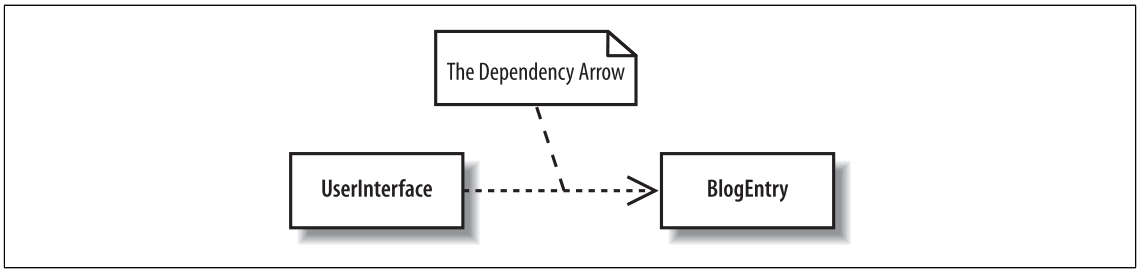


Figure 5-2. The UserInterface is dependent on the BlogEntry class because it will need to read the contents of a blog's entries to display them to the user

The `UserInterface` and `BlogEntry` classes simply work together at the times when the user interface wants to display the contents of a blog entry. In class diagram terms, the two classes of object are dependent on each other to ensure they work together at runtime.

A dependency implies only that objects of a class *can* work together; therefore, it is considered to be the weakest direct relationship that can exist between two classes.

The dependency relationship is often used when you have a class that is providing a set of general-purpose utility functions, such as in Java's regular expression (`java.util.regex`) and mathematics (`java.math`) packages. Classes depend on the `java.util.regex` and `java.math` classes to use the utilities that those classes offer.

Association

Although dependency simply allows one class to use objects of another class, *association* means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute. If you find yourself saying that a class *works with* an object of another class, then the relationship between those classes is a great candidate for association rather than just a dependency. Association is shown using a simple line connecting two classes, as shown in Figure 5-3.

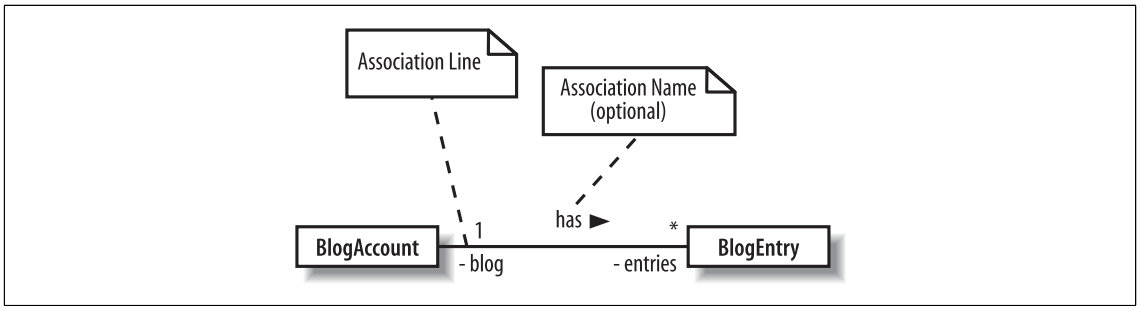


Figure 5-3. The BlogAccount class is optionally associated with zero or more objects of the BlogEntry class; the BlogEntry is also associated with one and only one BlogAccount

Navigability is often applied to an association relationship to describe which class contains the attribute that supports the relationship. If you take Figure 5-3 as it currently stands and implement the association between the two classes in Java, then you would get something like that shown in Example 5-1.

Example 5-1. The BlogAccount and BlogEntry classes without navigability applied to their association relationship

```
public class BlogAccount {

   // Attribute introduced thanks to the association with the BlogEntry class
   private BlogEntry[] entries;

   // ... Other Attributes and Methods declared here ...
}

public class BlogEntry {

   // Attribute introduced thanks to the association with the Blog class
   private BlogAccount blog;

   // ... Other Attributes and Methods declared here ...
}
```

Without more information about the association between the `BlogAccount` and `BlogEntry` classes, it is impossible to decide which class should contain the association introduced attribute; in this case, both classes have an attribute added. If this was intentional, then there might not be a problem; however, it is more common to have only one class referencing the other in an association.

In our system, it makes more sense to be able to ask a blog account what entries it contains, rather than asking the entry what blog account it belongs to. In this case, we use navigability to ensure that the `BlogAccount` class gets the association introduced attribute, as shown in Figure 5-4.

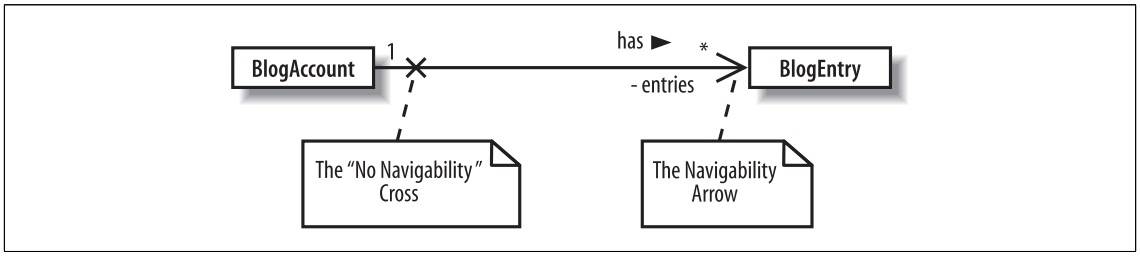


Figure 5-4. If we change Figure 5-3 to incorporate the navigability arrow, then we can declare that you should be able to navigate from the blog to its entries

Updating the association between the `BlogAccount` class and the `BlogEntry` class as shown in Figure 5-4 would result in the code shown in Example 5-2.

Example 5-2. With navigability applied, only the BlogAccount class contains an association introduced attribute

```
public class BlogAccount {

   // Attribute introduced thanks to the association with the BlogEntry class
   private BlogEntry[] entries ;

   // ... Other Attributes and Methods declared here ...
}

public class BlogEntry
{
   // The blog attribute has been removed as it is not necessary for the
   // BlogEntry to know about the BlogAccount that it belongs to.

   // ... Other Attributes and Methods declared here ...
}
```

Association classes

Sometimes an association itself introduces new classes. Association classes are particularly useful in complex cases when you want to show that a class is related to two classes *because* those two classes have a relationship with each other, as shown in Figure 5-5.

In Figure 5-5, the `BlogEntry` class is associated with a `BlogAccount`. However, depending on the categories that the account contains, the blog entry is also associated with any number of categories. In short, the association relationship between a blog account and a blog entry results in an association relationship with a set of categories (whew!).

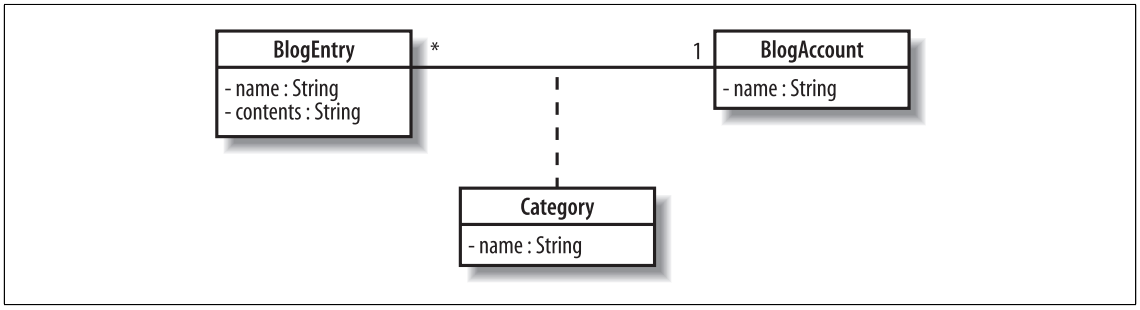


Figure 5-5. A BlogEntry is associated with a Category by virtue of the fact that it is associated with a particular BlogAccount

There are no hard and fast rules for exactly how an association class is implemented in code, but, for example, the relationships shown in Figure 5-5 could be implemented in Java, as shown in Example 5-3.

Example 5-3. One method of implementing the BlogEntry to BlogAccount relationship and the associated Category class in Java

```
public class BlogAccount {
   private String name;
   private Category[] categories;
   private BlogEntry[] entries;
}

public class Category {
   private String name;
}

public class BlogEntry {
   private String name;
   private Category[] categories
}
```

Aggregation

Moving one step on from association, we encounter the aggregation relationship. Aggregation is really just a stronger version of association and is used to indicate that a class actually *owns but may share* objects of another class.

Aggregation is shown by using an empty diamond arrowhead next to the owning class, as shown in Figure 5-6.

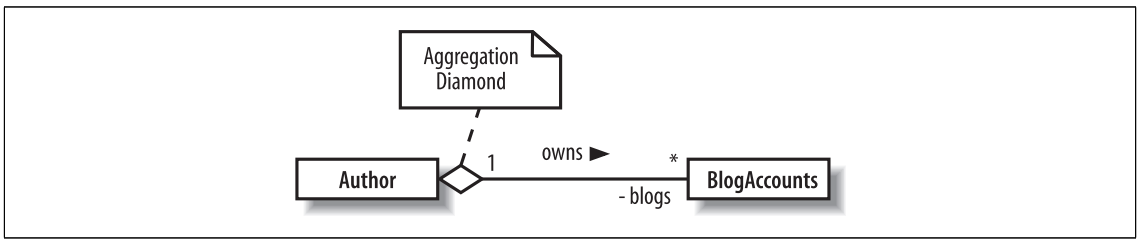


Figure 5-6. An aggregation relationship can show that an Author owns a collection of blogs

The relationship between an author and his blogs, as shown in Figure 5-6, is much stronger than just association. An author owns his blogs, and even though he *might* share them with other authors, in the end, his blogs are his own, and if he decides to remove one of his blogs, then he can!

Where's the code? Actually, the Java code implementation for an aggregation relationship is exactly the same as the implementation for an association relationship; it results in the introduction of an attribute.

Composition

Moving one step further down the class relationship line, composition is an even stronger relationship than aggregation, although they work in very similar ways. Composition is shown using a closed, or filled, diamond arrowhead, as shown in Figure 5-7.

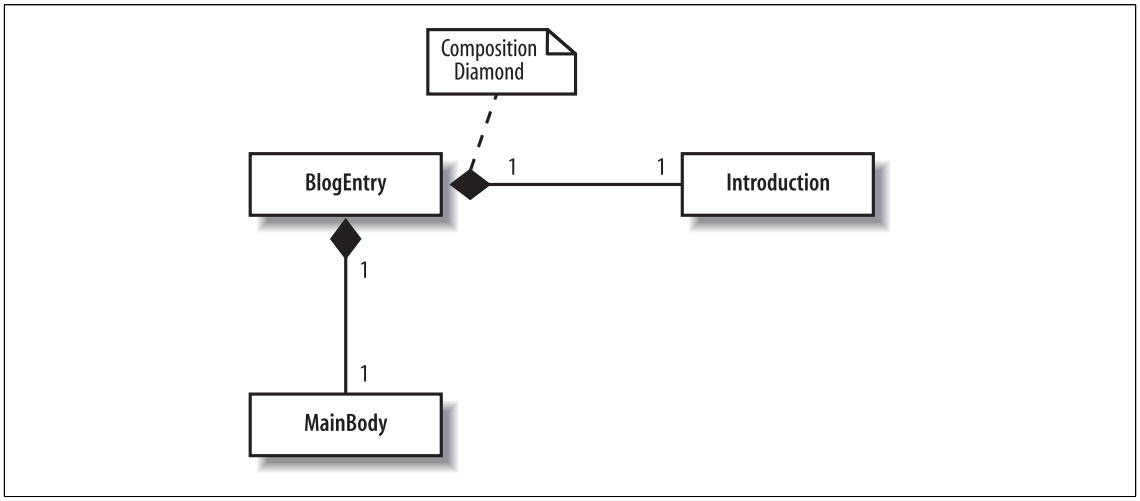


Figure 5-7. A BlogEntry is made up of an Introduction and a MainBody

A blog entry's introduction and main body sections are actually *parts of* the blog entry itself and won't usually be shared with other parts of the system. If the blog entry is deleted, then its corresponding parts are also deleted. This is exactly what composition is all about: you are modeling the internal parts that make up a class.

Similar to aggregation, the Java code implementation for a composition relationship results only in the introduction of an attribute.

Generalization (Otherwise Known as Inheritance)

Generalization and inheritance are used to describe a class that *is a type of* another class. The terms *has a* and *is a type of* have become an accepted way of deciding

whether a relationship between two classes is aggregation or generalization for many years now. If you find yourself stating that a class has a part that is an object of another class, then the relationship is likely to be one of association, aggregation, or composition. If you find yourself saying that the class is a type of another class, then you might want to consider using generalization instead.

In UML, the generalization arrow is used to show that a class is a type of another class, as shown in Figure 5-8.

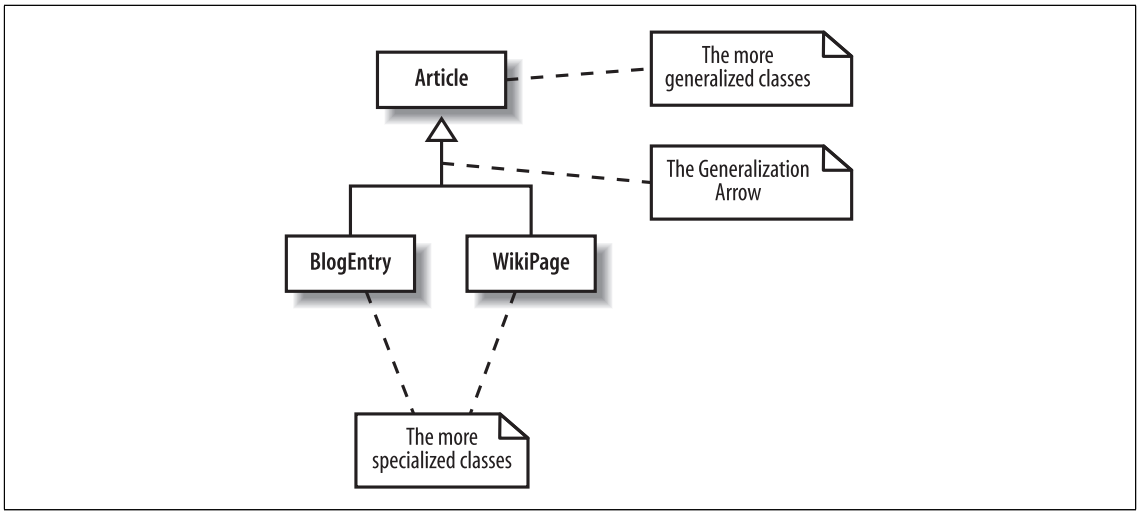
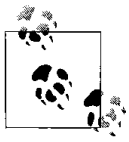


Figure 5-8. Showing that a BlogEntry and WikiPage are both types of Article

The more generalized class that is inherited from—at the arrow end of the generalization relationship, `Article` in this case—is often referred to as the parent, base, or superclass. The more specialized classes that do the inheriting—`BlogEntry` and `WikiPage` in this case—are often referred to as the children or derived classes. The specialized class inherits all of the attributes and methods that are declared in the generalized class and may add operations and attributes that are only applicable in specialized cases.

The key to why inheritance is called generalization in UML is in the difference between what a parent class and a child class each represents. Parent classes describe a more *general* type, which is then made more specialized in child classes.



If you need to check that you've got a generalization relationship correct, this rule of thumb can help: generalization relationships make sense only in one direction. Although it's true to say that a guitarist is a musician, it is not true to say that all musicians are guitarists.

Generalization and implementation reuse

A child class inherits and reuses all of the attributes and methods that the parent contains and that have public, protected, or default visibility. So, generalization offers a great way of expressing that one class is a type of another class, and it offers a

way of reusing attributes and behavior between the two classes. That makes generalization look like the answer to your reuse prayers, doesn't it?

Just hold on a second! If you are thinking of using generalization just so you can reuse some behavior in a particular class, then you probably need to think again. Since a child class can see most of the internals of its parent, it becomes tightly coupled to its parent's implementation.

One of the principles of good object-oriented design is to avoid tightly coupling classes so that when one class changes, you don't end up having to change a bunch of other classes as well. Generalization is the strongest form of class relationship because it creates a tight coupling between classes. Therefore, it's a good rule of thumb to use generalization only when a class really is a more specialized type of another class and not just as a convenience to support reuse.

If you still want to reuse a class's behavior in another class, think about using delegation. For more information on how delegation works and why it is preferred over inheritance, check out the excellent book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley).

Multiple inheritance

Multiple inheritance—or multiple generalization in the official UML terminology—occurs when a class inherits from two or more parent classes, as shown in Figure 5-9.

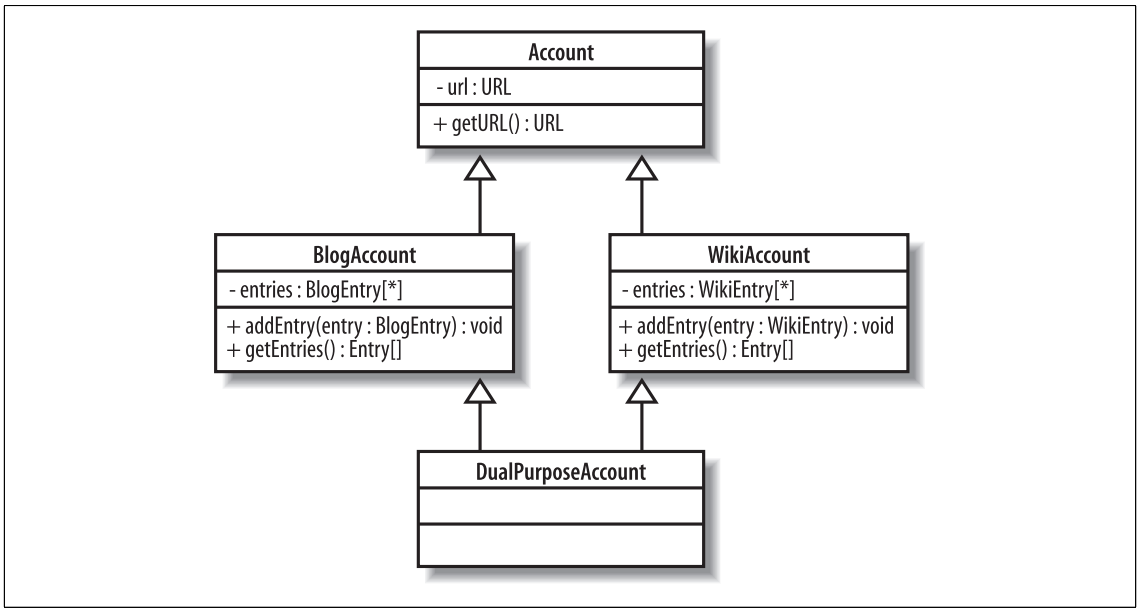


Figure 5-9. The DualPurposeAccount is a BlogAccount and a WikiAccount all combined into one

Although multiple inheritance is supported in UML, it is still not considered to be the best practice in most cases. This is mainly due to the fact that multiple inheritance

presents a complicated problem when the two parent classes have overlapping attributes or behavior.

So, why the complication? In Figure 5-9, the `DualPurposeAccount` class inherits all of the behavior and attributes from the `BlogAccount` and `WikiAccount` classes, but there is quite a bit of duplication between the two parent classes. For example, both `BlogAccount` and `WikiAccount` contain a copy of the `name` attribute that they in turn inherited from the `Account` class. Which copy of this attribute does the `DualPurposeAccount` class get, or does it get two copies of the same attribute? The situation becomes even more complicated when the two parent classes contain the same operation. The `BlogAccount` class has an operation called `getEntries()` and so does the `WikiAccount`.

Although the `BlogAccount` and `WikiAccount` classes are kept separate, the fact that they both have a `getEntries()` operation is not a problem. However, when both of these classes become the parent to another class through inheritance, a conflict is created. When `DualPurposeAccount` inherits from both of these classes, which version of the `getEntries()` method does it get? If the `DualPurposeAccount`'s `getEntries()` operation is invoked, which method should be executed to get the Wiki entries or the blog entries?

The answers to these question are unfortunately often hidden in implementation details. For example, if you were using the C++ programming language, which supports multiple inheritance, you would use the C++ language's own set of rules about how to resolve these conflicts. Another implementation language may use a different set of rules completely. Because of these complications, multiple inheritance has become something of a taboo subject in object-oriented software development—to the point where the current popular development languages, such as Java and C#, do not even support it. However, the fact remains that there are situations where multiple inheritance can make sense and be implemented—in languages such as C++, for example—so UML still needs to support it.

Constraints

Sometimes you will want to restrict the ways in which a class can operate. For example, you might want to specify a *class invariant*—a rule that specifies that a particular condition should never happen within a class—or that one attribute's value is based on another, or that an operation should never leave the class in an irregular state. These types of constraints go beyond what can be done with simple UML notation and calls for a language in its own right: the OCL.

There are three types of constraint that can be applied to class members using OCL:

Invariants
: An *invariant* is a constraint that must always be true; otherwise the system is in an invalid state. Invariants are defined on class attributes.

Preconditions
: A *precondition* is a constraint that is defined on a method and is checked before the method executes. Preconditions are frequently used to validate input parameters to a method.

Postconditions
: A *postcondition* is also defined on a method and is checked after the method executes. Postconditions are frequently used to describe how values were changed by a method.

Constraints are specified using either the OCL statement in curly brackets next to the class member or in a separate note, as shown in Figure 5-10.

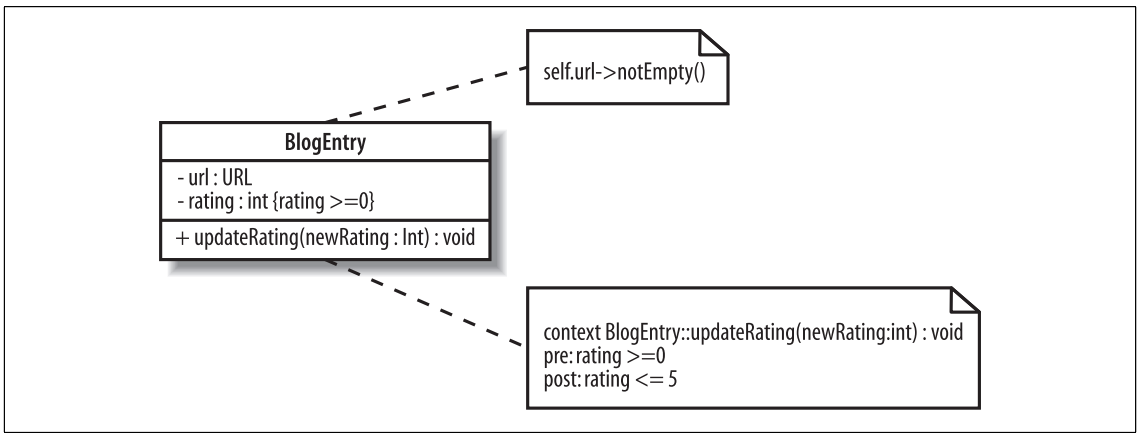


Figure 5-10. Three constraints are set on the BlogEntry class: self.url>notEmpty() and rating>=0 are both invariants, and there is a postcondition constraint on the updateRating(..) operation

In Figure 5-10, the `url` attribute is constrained to never being `null` and the `rating` attribute is constrained so that it must never be less than 0. To ensure that the `updateRating(..)` operation checks that the `rating` attribute is not less than 0, a `precondition` constraint is set. Finally, the `rating` attribute should never be more than 5 after it has been updated, so this is specified as a `postcondition` constraint on the `updateRating(..)` operation.

OCL allows you to specify all sorts of constraints that limit how your classes can operate. For more information on OCL, see Appendix A.

Abstract Classes

Sometimes when you are using generalization to declare a nice, reusable, generic class, you will not be able to implement all of the behavior that is needed by the general class. If you are implementing a `Store` class to store and retrieve the CMS's articles, as shown in Figure 5-11, you might want to indicate that exactly how a `Store`

stores and retrieves the articles is not known at this point and should be left to subclasses to decide.

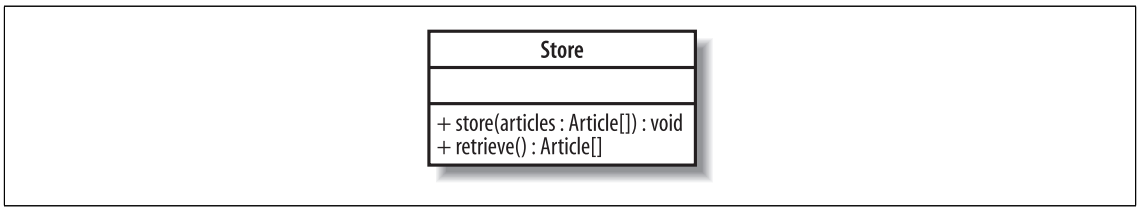


Figure 5-11. Using regular operations, the Store class needs to know how to store and retrieve a collection of articles

To indicate that the implementation of the `store(..)` and `retrieve(..)` operations is to be left to subclasses by declaring those operations as abstract, write their signatures in italics, as shown in Figure 5-12.

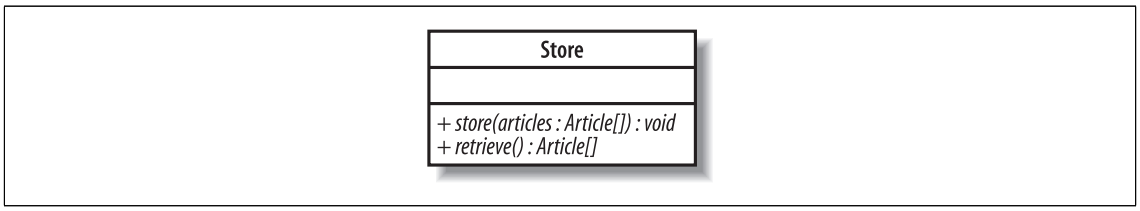


Figure 5-12. The store(..) and retrieve(..) operations do not now need to be implemented by the Store class

An abstract operation does not contain a method implementation and is really a placeholder that states, "I am leaving the implementation of this behavior to my subclasses." If any part of a class is declared abstract, then the class itself also needs to be declared as abstract by writing its name in italics, as shown in Figure 5-13.

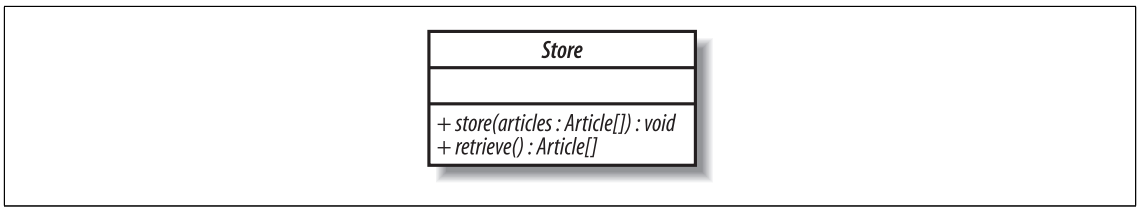


Figure 5-13. The complete abstract Store class

Now that the `store(..)` and `retrieve(..)` operations on the `Store` class are declared as abstract, they do not have to have any methods implemented, as shown in Example 5-4.

Example 5-4. The problem of what code to put in the implementation of the play()operation is solved by declaring the operation and the surrounding class as abstract

```
public abstract class Store {
   public abstract void store(Article[] articles);
   public abstract Article[] retrieve( );
}
```

An abstract class cannot be instantiated into an object because it has pieces missing. The `Store` class might implement the `store(..)` and `retrieve(..)` operations but because it is abstract, children who inherit from the `Store` class will have to implement or declare abstract the `Store` class's abstract operations, as shown in Figure 5-14.

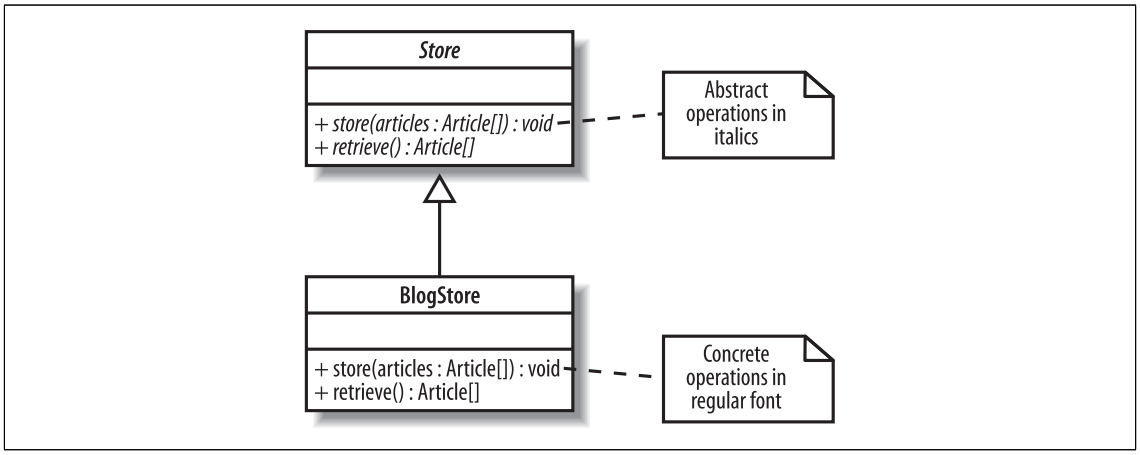


Figure 5-14. The BlogStore class inherits from the abstract Store class and implements the store(..) and retrieve(..) operations; classes that completely implement all of the abstract operations inherited from their parents are sometimes referred to as "concrete"

By becoming abstract, the `Store` class has delayed the implementation of the `store(..)` and `retrieve(..)` operations until a subclass has enough information to implement them. The `BlogStore` class can implement the `Store` class's abstract operations because it knows how to store away a blog, as shown in Example 5-5.

Example 5-5. The BlogStore class completes the abstract parts of the Store class

```
public abstract class Store {

   public abstract void store(Article[] articles);
   public abstract Article[] retrieve( );
}

public class BlogStore {

   public void store(Article[] articles) {
      // Store away the blog entries here ...
   }

   public Article[] retrieve( ) {
      // Retrieve and return the stored blog entries here...
   }
}
```

An abstract class cannot be instantiated as an object because there are parts of the class definition missing: the abstract parts. Child classes of the abstract class can be

instantiated as objects if they complete all of the abstract parts missing from the parent, thus becoming a concrete class, as shown in Example 5-6.

Example 5-6. You can create objects of non-abstract classes, and any class not declared as abstract needs to implement any abstract behavior it may have inherited

```
public abstract class Store {

   public abstract void store(Article[] articles);
   public abstract Article[] retrieve( );
}

public class BlogStore {

   public void store(Article[] articles) {
      // Store away the blog entries here ...
   }

   public Article[] retrieve( ) {
      // Retrieve and return the stored blog entries here...
   }
}
public class MainApplication {

   public static void main(String[] args) {

      // Creating an object instance of the BlogStore class.
      // This is totally fine since the BlogStore class is not abstract.
      BlogStore store = new BlogStore( );
      blogStore.store(new Article[]{new BlogEntry( )});
      Article[] articlesInBlog = blogStore.retrieve( );

      // Problem! It doesn't make sense to create an object of
      // an abstract class because the implementations of the
      // abstract pieces are missing!
      Store store = new Store( ); // Compilation error here!
   }
}
```

Abstract classes are a very powerful mechanism that enable you to define common behavior and attributes, but they leave some aspects of how a class will work to more concrete subclasses. A great example of where abstract classes and interfaces are used is when defining the generic roles and behavior that make up design patterns. However, to implement an abstract class, you have to use inheritance; therefore, you need to be aware of all the baggage that comes with the strong and tightly coupling generalization relationship.

See the "Generalization (Otherwise Known as Inheritance)" section earlier in this chapter for more information on the trials and tribulations of using generalization. For more on design patterns and how they make good use of abstract classes, check out the definitive book on the subject *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley).

Interfaces

If you want to declare the methods that concrete classes should implement, but not use abstraction since you have only one inheritance relationship (if you're coding in Java), then interfaces could be the answer.

An *interface* is a collection of operations that have no corresponding method implementations—very similar to an abstract class that contains only abstract methods. In some software implementation languages, such as C++, interfaces are implemented as abstract classes that contain no operation implementations. In newer languages, such as Java and C#, an interface has its own special construct.

Interfaces tend to be much safer to use than abstract classes because they avoid many of the problems associated with multiple inheritance (see the "Multiple inheritance" section earlier in this chapter). This is why programming languages such as Java allow a class to implement any number of interfaces, but a class can inherit from only one regular or abstract class.

Think of an interface as a very simple contract that declares, "These are the operations that must be implemented by classes that intend to meet this contract." Sometimes an interface will contain attributes as well, but in those cases, the attributes are usually static and are often constants. See Chapter 4 for more on the use of static attributes.

In UML, an interface can be shown as a stereotyped class notation or by using its own ball notation, as shown in Figure 5-15.

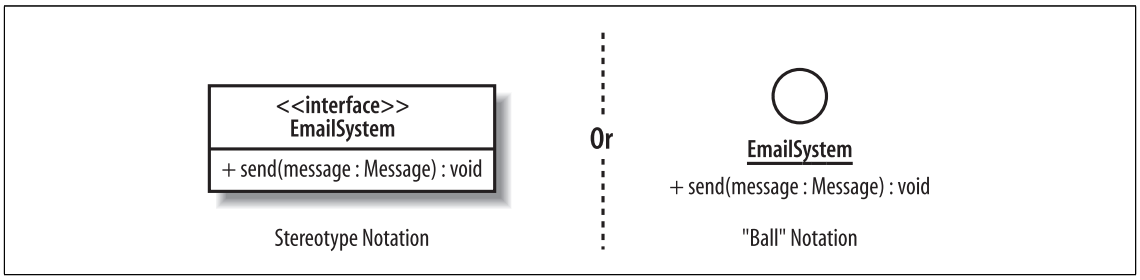


Figure 5-15. Capturing an interface to an EmailSystem using the stereotype and "ball " UML notation; unlike abstract classes, an interface does not have to show that its operations are not implemented, so it doesn't have to use italics

If you were implementing the `EmailSystem` interface from Figure 5-15 in Java, then your code would look like Example 5-7.

Example 5-7. The EmailSystem interface is implemented in Java by using the interface keyword and contains the single send(..) operation signature with no operation implementation

```
public interface EmailSystem {
   public void send(Message message);
}
```

You can't instantiate an interface itself, much like you can't instantiate an abstract class. This is because all of the implementations for an interface's operations are missing until it is realized by a class. If you are using the "ball" interface notation, then you realize an interface by associating it with a class, as shown in Figure 5-16.

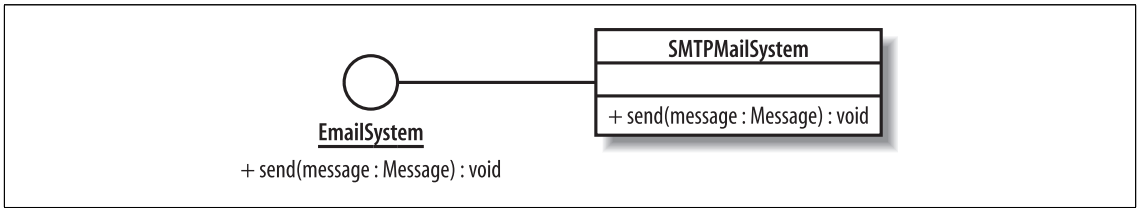


Figure 5-16. The SMTPMailSystem class implements, or realizes, all of the operations specified on the EmailSystem interface

If you have used the stereotype notation for your interface, then a new arrow is needed to show that this is a realization relationship, as shown in Figure 5-17.

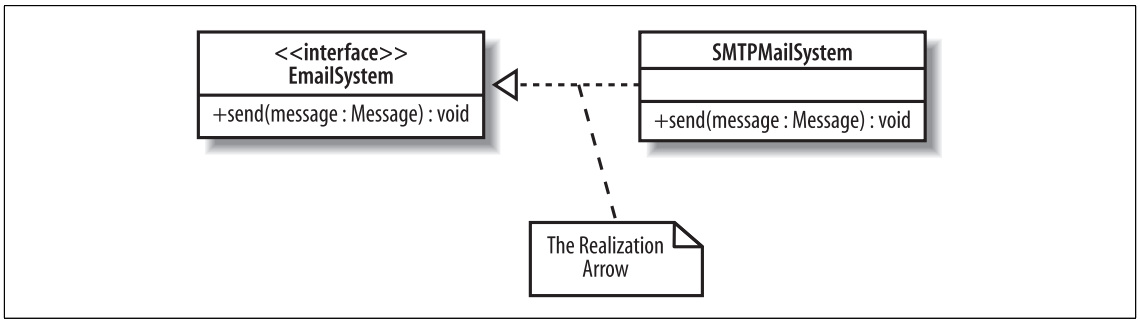


Figure 5-17. The realization arrow specifies that the SMTPMailSystem realizes the EmailSystem interface

Both Figures 5-16 and 5-17 and would have resulted in the same Java code being generated, as shown in Example 5-8.

Example 5-8. Java classes realize interfaces using the implements keyword

```
public interface EmailSystem
{
   public void send(Message message));
}

public class SMTPMailSystem implements EmailSystem
{
   public void send(Message message)
   {
      // Implement the interactions with an SMTP server to send the message
   }

   // ... Implementations of the other operations on the Guitarist class ...
}
```

If a class realizes an interface but does not implement all of the operations that the interface specifies, then that class needs to be declared abstract, as shown in Figure 5-18.

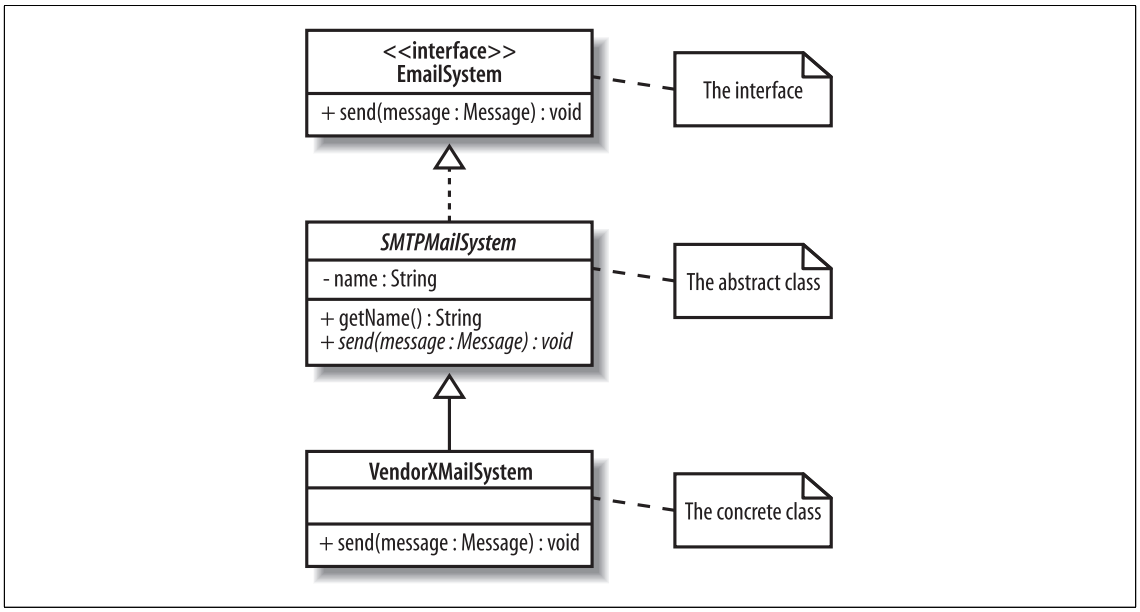


Figure 5-18. Because the SMTPMailSystem class does not implement the send(..) operation as specified by the EmailSystem interface, it needs to be declared abstract; the VendorXMailSystem class completes the picture by implementing all of its operations

Interfaces are great at completely separating the behavior that is required of a class from exactly how it is implemented. When a class implements an interface, objects of that class can be referred to using the interface's name rather than the class name itself. This means that other classes can be dependent on interfaces rather than classes. This is generally a good thing since it ensures that your classes are as loosely coupled as possible. If your classes are loosely coupled, then when a class implementation changes other classes should not break (because they are dependent on the interface, not on the class itself).

Using Interfaces

It is good practice to de-couple dependencies between your classes using interfaces; some programming environments, such as the Spring Framework, enforce this interface-class relationship. The use of interfaces, as opposed to abstract classes, is also useful when you are implementing design patterns. In languages such as Java, you don't really want to use up the single inheritance relationship just to use a design pattern. A Java class can implement any number of interfaces, so they offer a way of enforcing a design pattern without imposing the burden of having to expend that one inheritance relationship to do it.

Templates

Templates are an advanced but useful feature of object orientation. A *template*—or parameterized class, as they are sometimes referred to—is helpful when you want to postpone the decision as to which classes a class will work with. When you declare a template, as shown in Figure 5-19, it is similar to declaring, "I know this class will have to work with other classes, but I don't know or necessarily care what those classes actually end up being."

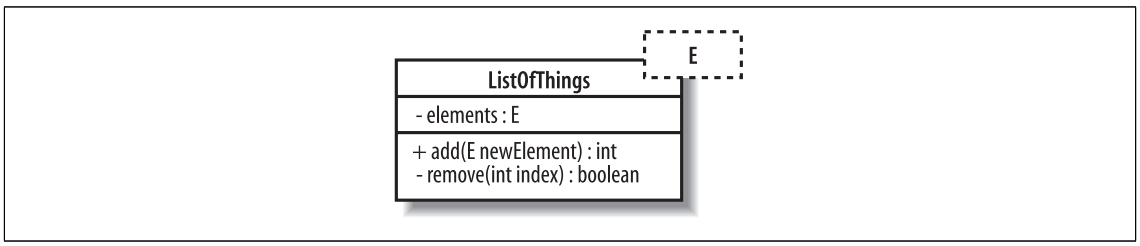


Figure 5-19. A template in UML is shown by providing an extra box with a dashed border to the top right of the regular class box

The `ListOfThings` class in Figure 5-19 is parameterized with the type referred to as `E`. There is no class in our model called `E`; `E` is nothing more than a placeholder that can be used at a later point to tell the `ListOfThings` class the type of object that it will need to store.

Lists

Lists tend to be the most common examples of how to use templates, and with very good reason. Lists and their cousins, such as maps and sets, all store objects in different ways, but they don't actually care what classes those objects are constructed from. For this reason, one of the best real-world uses of templates is in the Java collection classes. Prior to Java 5, the Java programming language did not have a means of specifying templates. With the release of Java 5 and its generics feature, you can now not only create your own templates, but the original collection classes are all available to use as templates as well. To find out more about Java 5 generics, check out the latest edition of *Java in a Nutshell* (O'Reilly).

To use a class that is a template, you first need to bind its parameters. The `ListOfSomething` class template doesn't yet know what it's supposed to be storing; you need to tell the template what actual classes it will be working with; you need to bind the parameter referred to so far as just `E` to an actual class.

You can bind a template's parameters to a specific set of classes in one of two ways. First, you can subclass the template, binding the parameters as you go, as shown in Figure 5-20.

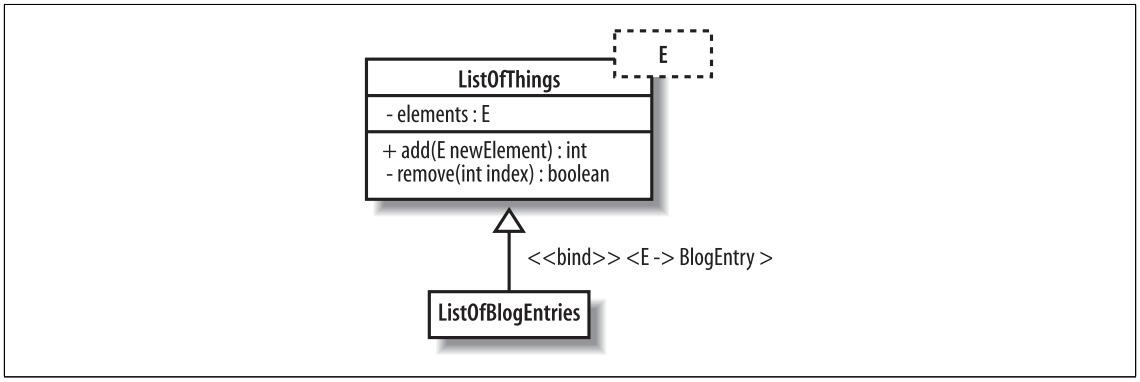


Figure 5-20. The ListOfThings class is subclassed into a ListOfBlogEntries, binding the single parameter E to the concrete BlogEntry class

Binding by subclass in Figure 5-20 allows you to reuse all of the generic behavior in the `ListOfThings` class and restrict that behavior in the `ListOfBlogEntries` class to only adding and removing `BlogEntry` objects.

The real power of templates is much more obvious when you use the second approach to template parameter binding—binding at runtime. You bind at runtime when a template is told the type of parameters it will have *as it is constructed into an object*.

Runtime template binding is about objects rather than classes; therefore, a new type of diagram is needed: the object diagram. Object diagrams use classes to show some of the important ways they are used as your system runs. As luck would have it, object diagrams are the subject of the very next chapter.

What's Next

Class diagrams show the types of objects in your system. A useful next step is to look at object diagrams since they show how classes come alive at runtime as object instances, which is useful if you want to show runtime configurations. Object diagrams are covered in Chapter 6.

Composite structures are a diagram type that loosely shows context sensitive class diagrams and patterns in your software. Composite structures are described in Chapter 11.

After you've decided the responsibilities of the classes in your system, it's common to then create sequence and communication diagrams to show interactions between the parts. Sequence diagrams can be found in Chapter 7; communication diagrams are covered in Chapter 8.

It's also common to step back and organize your classes into packages. Package diagrams allow you to view dependencies at a higher level, helping you understand the stability of your software. Package diagrams are described in Chapter 13.