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 classgeneralizationor 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 missingabstractbits 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 waitas late as runtime if you chooseto decide which classes it will work with.

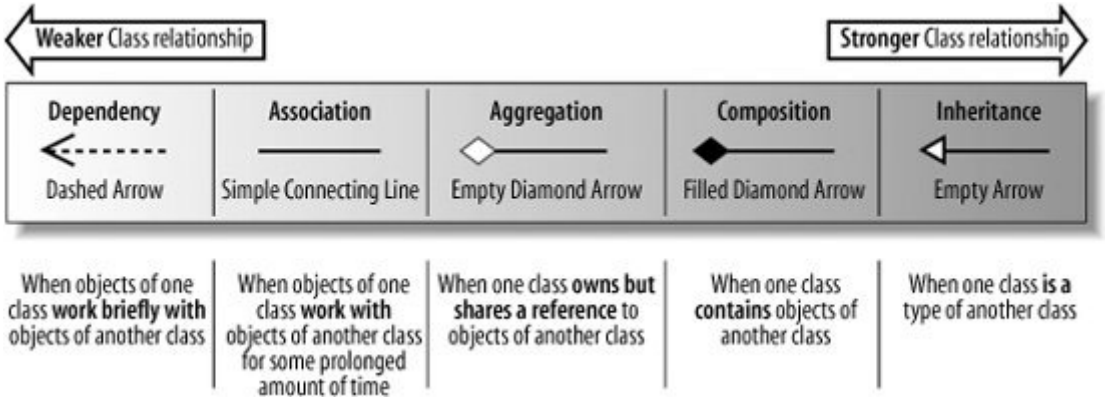
Together, these techniques complete your class diagram toolbox. They represent some of the most powerful concepts in object-oriented design and, when applied correctly, can make the difference between an OK design and a great piece of reusable design.

5.1. Class Relationships

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

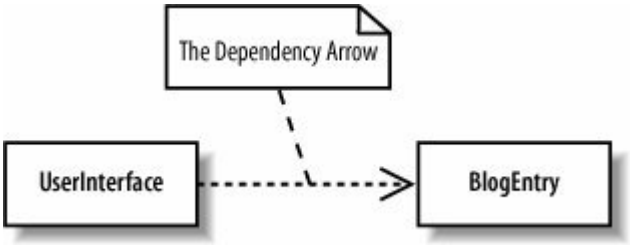
Figure 5-1. UML offers five different types of class relationship



5.1.1. 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.

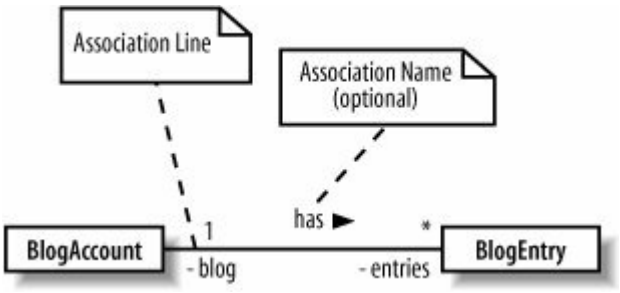
A dependency implies only that objects of a class can work together; therefore, it is considered to be the weakest direct relationship that can exist between two classes.

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.

5.1.2. 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.

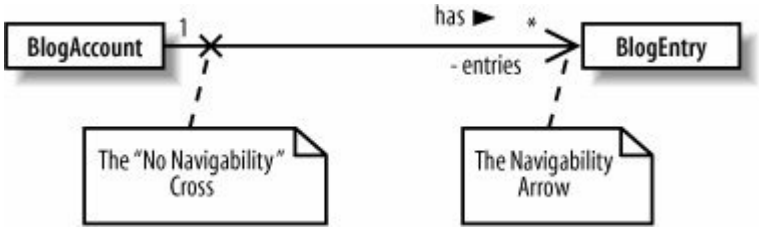
Example 5-1. The BlogAccount and BlogEntry classes without navigability applied to their association relationship

```
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

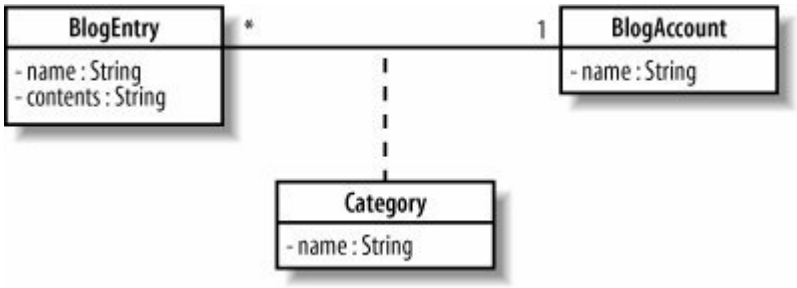
```
public class BlogAccount {

   // 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 ...
}
```

5.1.2.1. 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 an Author 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

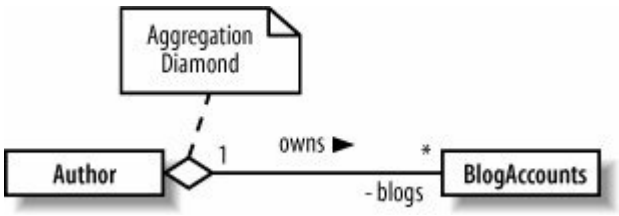
```
public class BlogAccount {
   private String name;
   private Category[] categories;
   private BlogEntry[] entries;
}

public class Category {
   private String name;
}

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

5.1.3. 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!

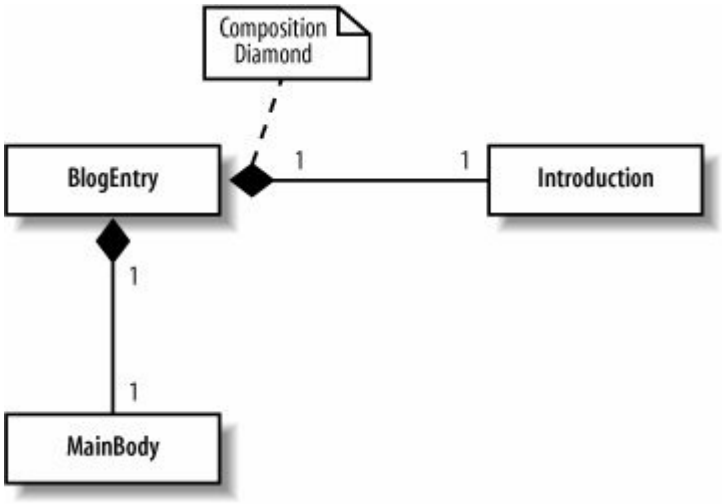
Where's the code? Actually, the Java code implementation for an aggregation relationship is exactly the same as the implementation for an association relationship; it results in the introduction of an attribute.

5.1.4. 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.

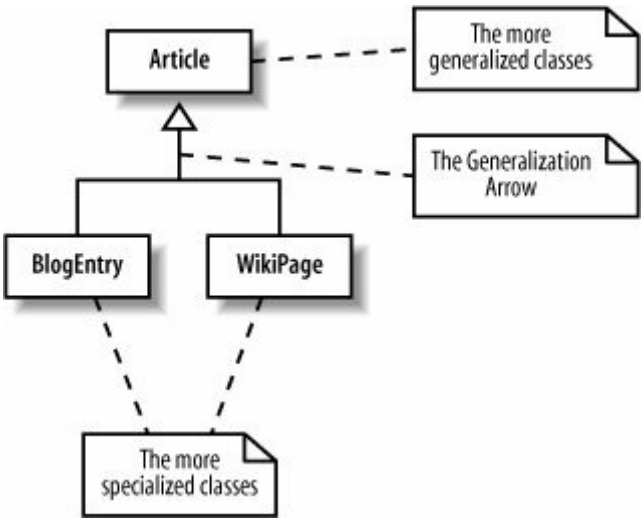
Similar to aggregation, the Java code implementation for a composition relationship results only in the introduction of an attribute.

5.1.5. 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 fromat the arrow end of the generalization relationship, `Article` in this caseis often referred to as the parent, base, or superclass. The more specialized classes that do the inheriting`BlogEntry` and `WikiPage` in this caseare 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 guitarists are musicians.

5.1.5.1. 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.

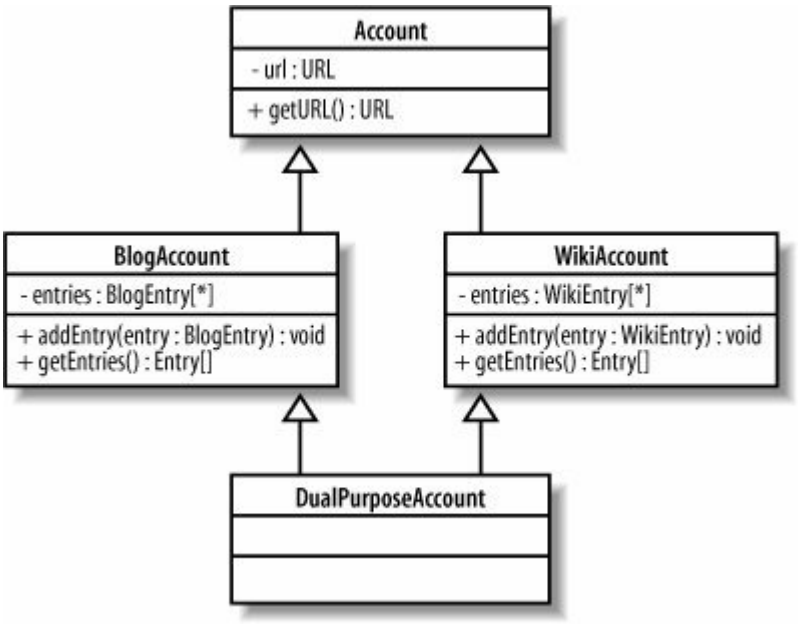
If you still want to reuse a class's behavior in another class, think about using delegation . For more information on how delegation works and why it is

preferred over inheritance, check out the excellent book, Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley).

5.1.5.2. Multiple inheritance

Multiple inheritanceor multiple generalization in the official UML terminologyoccurs 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 developmentto 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 implementedin languages such as C++, for exampleso UML still needs to support it.

5.2. 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 classor 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

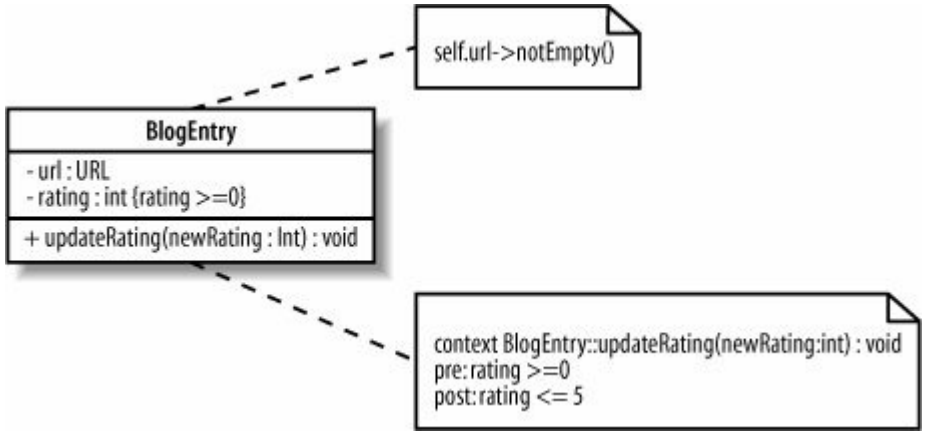
> A *precondition* is a constraint that is defined on a method and is checked before the method executes. Preconditions are frequently used to validate input parameters to a method.

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.

5.3. 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

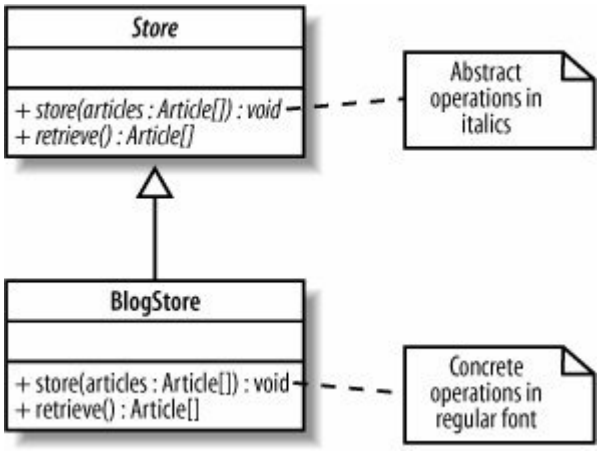
Now that the `store(..)` and `retrieve(..)` operations on the `Store` class are declared as abstract, they do not have to have any methods implemented, as shown in Example 5-4.

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 extends Store {

   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 extends Store {
```

```
   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).

5.4. 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 implementationsvery 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.

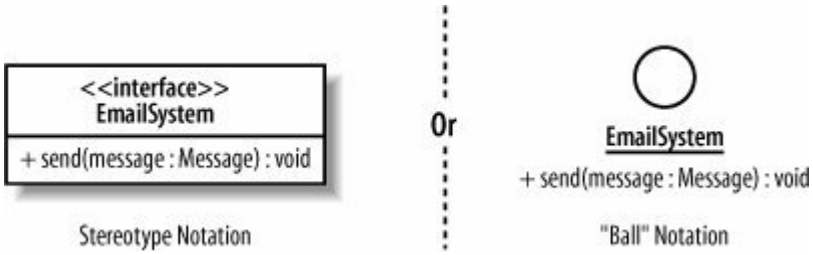
Interfaces tend to be much safer to use than abstract classes because they

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



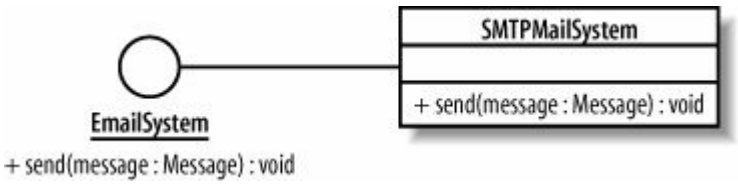
If you were implementing the `EmailSystem` interface from Figure 5-15 in Java, then your code would look like Example 5-7.

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);
}
```

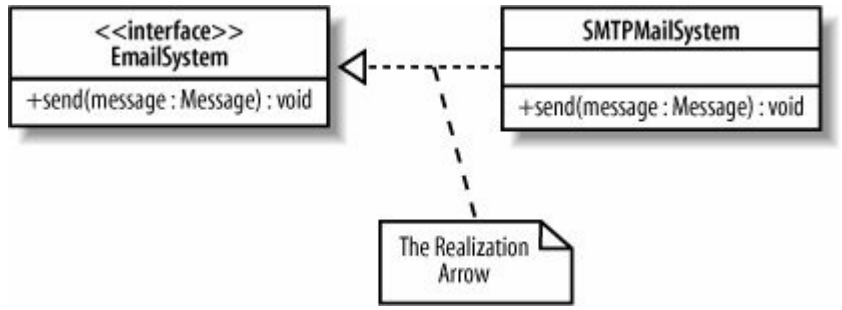
You can't instantiate an interface itself, much like you can't instantiate an abstract class. This is because all of the implementations for an interface's operations are missing until it is realized by a class. If you are using the "ball" interface notation, then you realize an interface by associating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



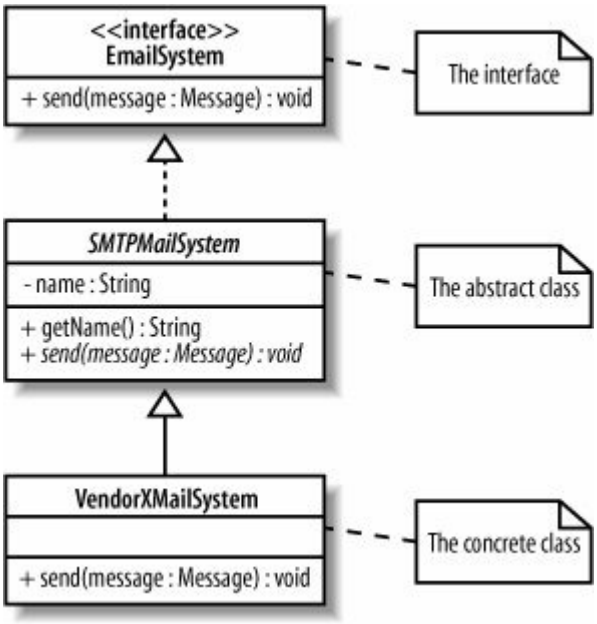
Both Figures 5-16 and 5-17 and would have resulted in the same Java code being generated, as shown in Example 5-8.

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).

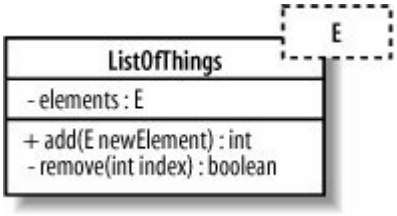
Using Interfaces

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.

5.5. Templates

Templates are an advanced but useful feature of object orientation. A templateor parameterized class, as they are sometimes referred tois 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.

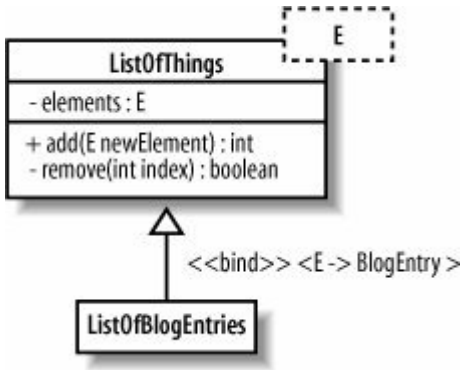
Lists

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 bindingbinding 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.

5.6. 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.