# 3

# Class Diagrams

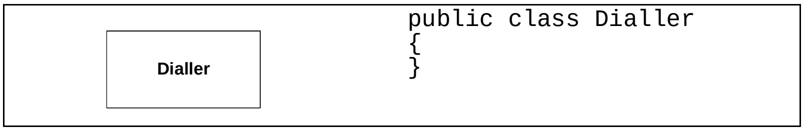
UML class diagrams allow us to denote the static contents of, and relationships between classes. In a class diagram we can show the member variables, and member functions of a class. We can also show whether one class inherits from another, or whether it holds a reference to another. In short, we can depict all the *source code dependencies* between classes.

This can be valuable. It can be much eaiser to evaluate the dependency structure of a system from a diagram than from source code. Diagrams make certain dependency structures visible. We can *see* dependency cycles, and determine how best to break them. We can see when abstract classes depend upon concrete classes, and determine a strategy for rerouting such dependencies. .

## The Basics

### Classes

Figure 3-1 shows the simplest form of class diagram. The class named `Dialler` is represented as a simple rectangle. This diagram represents nothing more than the code shown to its right.

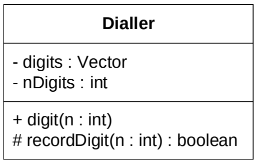


```
public class Dialler
{
}
```

**Figure 3-1**
Class Icon

This is the most common way you will represent a class. The classes on most diagramd don’t need any more than their name to make clear what is going on.

A class icon can be subdivided into compartments. The top compartment is for the name of the class, the second is for the variables of the class, and the third is for the methods of the class. Figure 3-2 shows these compartments and how they translate into code.



```
public class Dialler
{
  private Vector digits;
  int nDigits;
  public void digit(int n);
  protected boolean recordDigit(int n);
}
```

**Figure 3-2**

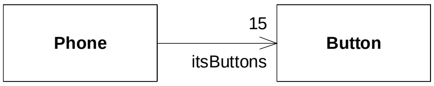
Notice the character in front of the variables and functions in the class icon. A dash `(-)` denotes `private`, hash `(#)` denotes `protected`, and plus `(+)` denotes `public`.

The type of a variable, or a function argument is shown after the colon following the variable or argument name. Similarly, the return value of a function is shows after the colon following the function.

This kind of detail is sometimes useful; but should not be used very often. UML diagrams are not the place to declare variables and function. Such declarations are better done in source code. Use these adornments only when they are essential to the purpose of the diagram.

## Association

Associations between classes most often represent instance variables that hold references to other objects. For example, in Figure 3-3 we see an association between `Phone` and `Button`. The direction of the arrow tells us that `Phone` holds a reference to `Button`. The name near the arrowhead is the name of the instance variable. The number near the arrowhead tells us how many references are held.

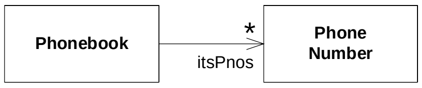


```
public class Phone
{
  private Button itsButtons[15];
}
```

**Figure 3-3**

## Multiplicity

The number near the arrowhead represents the number of objects that connected to the other associate. In Figure 3-3 above we saw that fifteen `Button` objects were connected to the `Phone` object. Below, in Figure 3-4, we see what happens when there is no limit. A `Phonebook` is connected to *many* `PhoneNumber` objects. The star means *many*. In Java this is most commonly implemented with a `Vector`, a `List`, or some other container type.



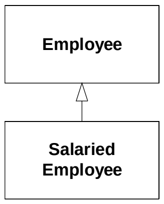
```
public class Phonebook
{
  private Vector itsPnos;
}
```

**Figure 3-4**

**Why didn't I use HASA?** You may have noticed that I avoided using the word "has". I could have said: "A `Phonebook` has many `PhoneNumbers`." This was intentional. The common OO verbs HASA and ISA have lead to a number of unfortunate misunderstandings. We'll explore some of them later in Chapter 6. For now, don't expect me to use the common terms. Rather, I'll use terms that are descriptive of what actually happens in software.

## Inheritance

You have to be very careful with your arrowheads in UML. Figure 3-5 shows why. The little arrowhead pointing at `Employee` denotes *inheritance*[1]. If you draw your arrowheads carelessly, it may be hard to tell whether you mean inheritance or association. To make it clearer, I often make inheritance relationships vertical and associations horizontal.



```
public class Employee
{
  ...
}

public class SalariedEmployee extends Employee
{
  ...
}
```

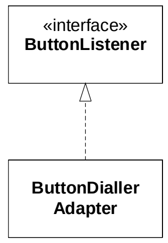
**Figure 3-5**

In UML all arrowheads point in the direction of *source code dependency*. Since it is the `SalariedEmployee` class that mentions the name of `Employee`, the arrowhead points at `Employee`. So, in UML, inheritance arrows point at the base class.

UML has a special notation for the kind of inheritance used between a Java class and a Java interface. It is shown, in Figure 3-6, as a dashed inheritance arrow[2]. In the diagrams to come, you'll probably catch me forgetting to dash the arrows that point to interfaces. I

1. Actually, it denotes *Generalization*, but as far as a Java programmer is concerned, the difference is moot.
2. This is called a Realizes relationship. There's more to it than just inheritance of interface, but the difference is beyond the scope of this book.

suggest you forget to dash the arrows that you draw on whiteboards too. Life's too short to be dashing arrows.



```
interface ButtonListener
{
  ...
}

public class ButtonDiallerAdapter
  implements ButtonListener
{
  ...
}
```

**Figure 3-6**

Figure 3-7 shows another way to convey the same information. Interfaces can be drawn as little lollipops on the classes that implement them. We often see this kind of notation in COM designs.

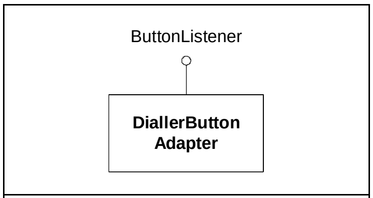


**Figure 3-7**

## An Example Class Diagram

Figure 3-8 shows a simple class diagram of part of an ATM system. This diagram is interesting both for what it shows, and for what it does not show. Note that I have taken pains to mark all the interfaces. I consider it crucial to make sure my readers know what classes I intend to be interfaces and which I intend to be implemented. For example, the diagram immediately tells you that `WithdrawTransaction` talks to a `CashDispenser` interface. Clearly some class in the system will have to implement the `CashDispenser`, but in this diagram we don't care which class it is.

Note that I have not been particularly thorough in documenting the methods of the various UI interfaces. Certainly `WithdrawlUI` will need more than just the two methods shown there. What about `promptForAccount` or `informCashDispenserEmpty`? Putting those methods in the diagram would just clutter it. By providing a representative batch of methods, I've given the reader the idea. That's all that's really necessary.

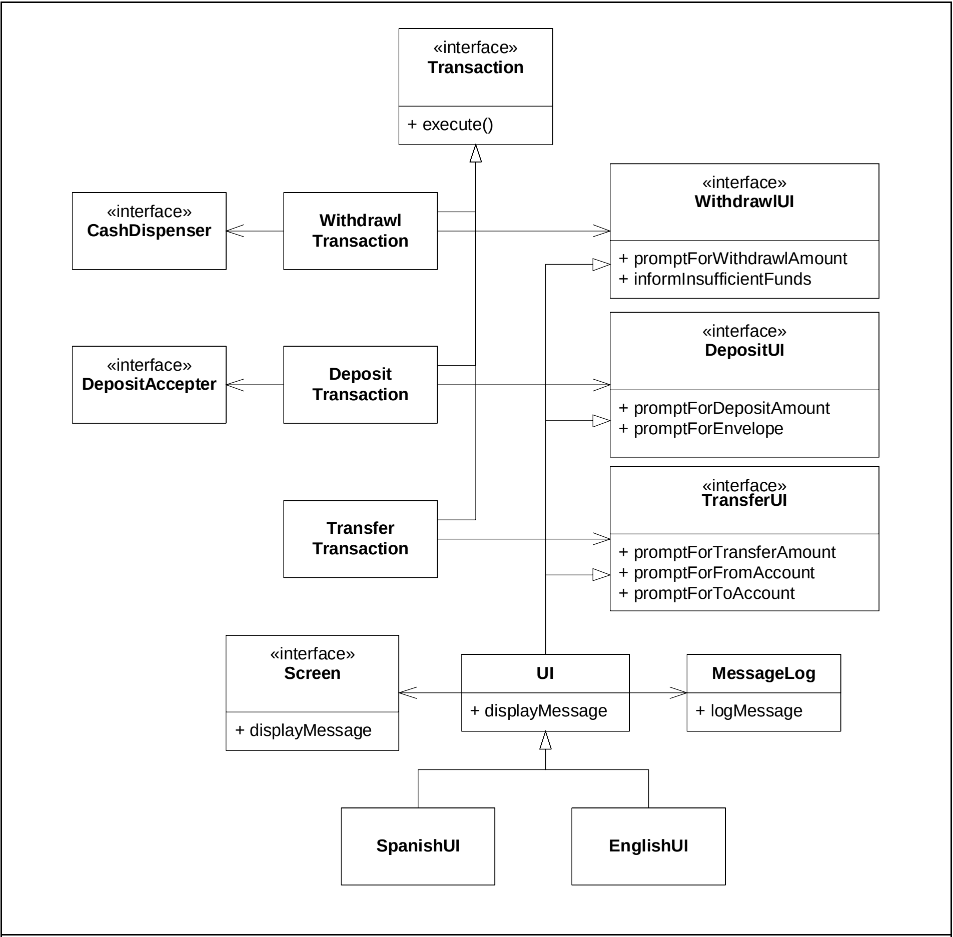


**Figure 3-8**
ATM Class Diagram

Again note the convention of horizontal association and vertical inheritance. This really helps to differentiate these vastly different kinds of relationships. Without a convention like this it can be hard to tease the meaning out of the tangle.

Notice how I've separated the diagram into three distinct zones. The transactions and their actions are on the left, the various UI interfaces are all on the right, and the UI implementation is on the bottom. Note also that the connections between the groupings are minimal and regular. In one case it is three associations, all pointing the same way. In the other case it is three inheritance relationships all merged into a single line. The grouping, and the way they are connected help the reader to see the diagram in coherent pieces.

You should be able to *see* the code as you look at the diagram. Is Listing 3-1 close to what you expected for the implementation of UI?

**Listing 3-1**
UI.java

```
public class UI implements
  WithdrawlUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public void displayMessage(String message)
  {
    itsMessageLog.logMessage(message);
    itsScreen.displayMessage(message);
  }
}
```

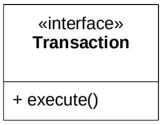
## The Details

There are a vast number of details and adornments that can be added to UML class diagrams. Most of the time these details and adornments should not be added. But there are times when they can be helpful.

### Class Stereotypes

Class stereotypes appear between guilmette[3] characters, usually above the name of the class. We have seen them before. The **«interface»** denotation in Figure 3-8 is a class stereotype. «interface» is one of two standard stereotypes that can be used by java programmers. The other is «utility».

**«interface».** All the methods of classes marked with this stereotype are abstract. None of the methods can be implemented. Morevover, «interface» classes can have no instance variables. The only variables they can have are static variables. This corresponds exactly to java interfaces. See Figure 3-9.

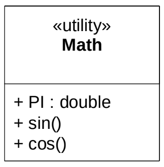


```
interface Transaction
{
  public void execute();
}
```

**Figure 3-9**

**«utility».** All the methods and variables of a «utility» class are static. Booch[4] used to call these *class utilities*. See Figure 3-10.

3. The quotation marks that look like double angle brackets « ». These are *not* two less-than, and two greater-than signs. If you use doubled inequality operators instead of the appropriate and proper guilmette characters, the UML police *will* find you.
4. [Booch94], p. 186



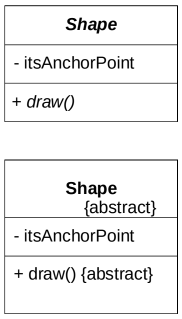
```
public class Math
{
  public static final double PI =
                              3.14159265358979323;

  public static double sin(double theta){...}
  public static double cos(double theta){...}
}
```

**Figure 3-10**

You can make your own stereotypes if you like. I often use stereotypes like «persistent», «C-API», «struct», or «function». You just have to make sure that the people who are reading your diagrams know what your stereotype means.

## Abstract classes

In UML there are two ways to denote that a class, or a method, is abstract. You can write the name in italics, or you can use the {abstract} property. Both options are shown in Figure 3-11.



```
public abstract class Shape
{
  private Point itsAnchorPoint;
  public abstract void draw();
}
```

**Figure 3-11**

It's a little difficult to write italics at a whiteboard, and the {abstract} property is wordy. So at the whiteboard, if I need to denote a class or method as abstract, I use the convention shown in Figure 3-12. This isn't legal UML, but at the whiteboard it is a lot more convenient[5].

## Properties

Properties, like {abstract} can be added to any class. They represent extra information that's not usually part of a class. You can create your own properties at any time.

5. Some of you may remember the Booch notation. One of the nice things about that notation was it's convenience. It was truly a whiteboard notation.

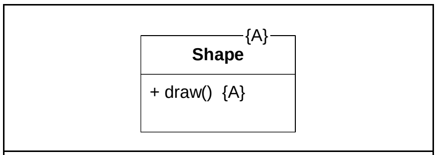


**Figure 3-12**

Properties are written in a comma separated list of name value pairs like this:

```
{author=Martin, date=20020429, file=shape.java, private}
```

The properties in the above example are not part of UML. The {abstract} property is the only defined property of UML that java programmres would find useful.

If a property does not have a value, it is assumed to take the boolean value true. Thus {abstract} and {abstract = true} are synonyms.

Properties are written below and to the right of the name of the class as shown in Figure 3-13.

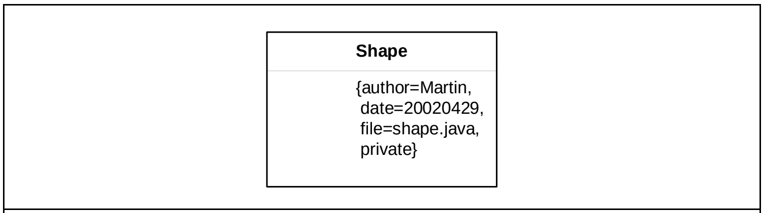
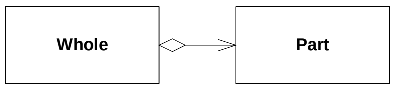


**Figure 3-13**

Other than the {abstract} property, I don't know when you'd this useful. Personally, in the many years that I've been writing UML diagrams, I've never had occasion to use class properties for anything.

## Aggregation

Aggregation is a special form of association that connotes a "whole/part" relationship. Figure 3-14 shows how it is drawn and implemented. Notice that the implementation shown in Figure 3-14 is indistinguishable from association. That's a hint.



```
public class Whole
{
  private Part itsPart;
}
```

**Figure 3-14**

Unfortunately, UML does not provide a strong definition for this relationship. This leads to confusion because various programmers and analysts adopt their own pet definitions for the relationship. For that reason, I don't use the relationship at all; and I recommend that you avoid it as well.

The one hard rule that UML gives us regarding aggregations is simply this. A whole cannot be its own part. Therefore *instances* cannot form cycles of aggregations. A single object cannot be an aggregate of itself; two objects cannot be aggregates of each other; three objects cannot form a ring of aggregation, etc. See Figure 3-15

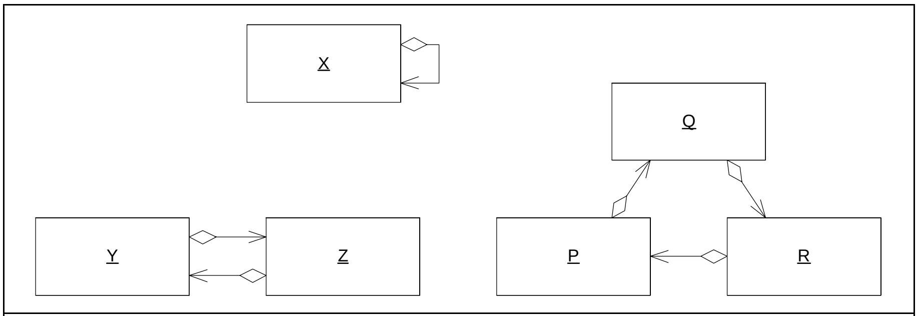
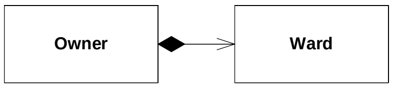


**Figure 3-15**
Cycles of aggregation between instances.

I don't find this to be a particularly useful definition. How often am I concerned about making sure that instances form a directed acyclic graph? Not very often. Therefore I find this relationship useless in the kinds of diagrams I draw.

## Composition

Composition is a special form of aggregation shown in Figure 3-16. Again, notice that the implementation is indistinguishable from association. However, this time the reason is not due to a lack of definition; this time it's because the relationship does not have a lot of use in a Java program. C++ programmers, on the other hand, find a *lot* of use for it.



```
public class Owner
{
  private Ward itsWard;
}
```

**Figure 3-16**

The same rule applies to Composition that applied to aggregation. There can be no cycles of instances. An owner cannot be its own ward. However UML provides quite a bit more definition.

- An instance of a ward cannot be simultaneously owned by two owners. The object diagram in Figure 3-17 is illegal. Note, however, that the corresponding class diagram is not illegal. An owner can transfer ownership of a ward to another owner.

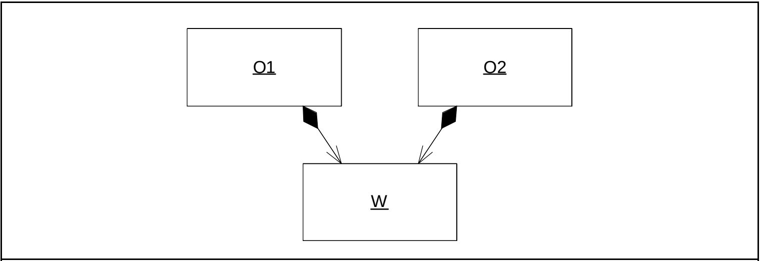


**Figure 3-17**
Illegal composition.

- The owner is reponsible for the lifetime of the ward. If the owner is destroyed, the ward must be destroyed with it. If the owner is copied, the ward must be copied with it.

In Java destruction happens behind the scenes by the garbage collector, so there is seldom a need to manage the lifetime of an object. Deep copies are not unheard of, but the need to show deep copy semantics on a diagram is rare. So, though I have used composition relationships to describe some Java programs, such use is infrequent.

Figure 3-18 shows how composition is used to denote deep copy. We have a class named `Address` which holds many `Strings`. Each string holds one line of the address. Clearly, when you make a copy of the `Address`, you want the copy to change independently of the original. Thus, we need to make a deep copy. The composition relationship between the `Address` and the `Strings` indicates that copies need to be deep[6].

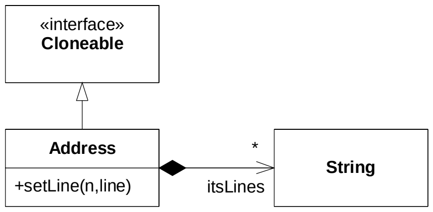
## Multiplicity

Objects can hold arrays or vectors of other objects; or they can hold many of the same kind of object in seperate instance variables. In UML this can be shown by placing a *multiplicity* expression on the far end of the association. Multiplicity expressions can be simple numbers, ranges, or a combination of both. For example Figure 3-19 shows a `BinaryTreeNode`, using a multiplicity of 2.

Here are the allowable forms:

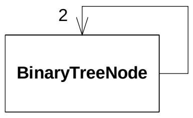
| | |
|---|---|
| • Digit. | The exact number of elements |
| • * or 0..* | zero to many. |
| • 0..1 | Zero or one. In Java this is often implemented with a reference that can be `null`. |
| • 1..* | One to many. |
| • 3..5 | Three to five. |
| • 0, 2..5, 9..* | Silly, but legal. |

6. Exercize: Why was it enough to clone the itsLines vector? Why didn't I have to clone the actual String instance?



```
import java.util.Vector;

public class Address implements Cloneable{
  private Vector itsLines = new Vector();

  public void setLine(int n, String line) {
    if (n >= itsLines.size())
      itsLines.setSize(n+1);
    itsLines.setElementAt(line, n);
  }

  public Object clone() throws CloneNotSupportedException
  {
    Address clone = (Address)super.clone();
    clone.itsLines = (Vector)itsLines.clone();
    return clone;
  }
}
```

**Figure 3-18**
DeepCopy is implied by Composition



```
public class BinaryTreeNode
{
  private BinaryTreeNode leftNode;
  private BinaryTreeNode rightNode;
}
```

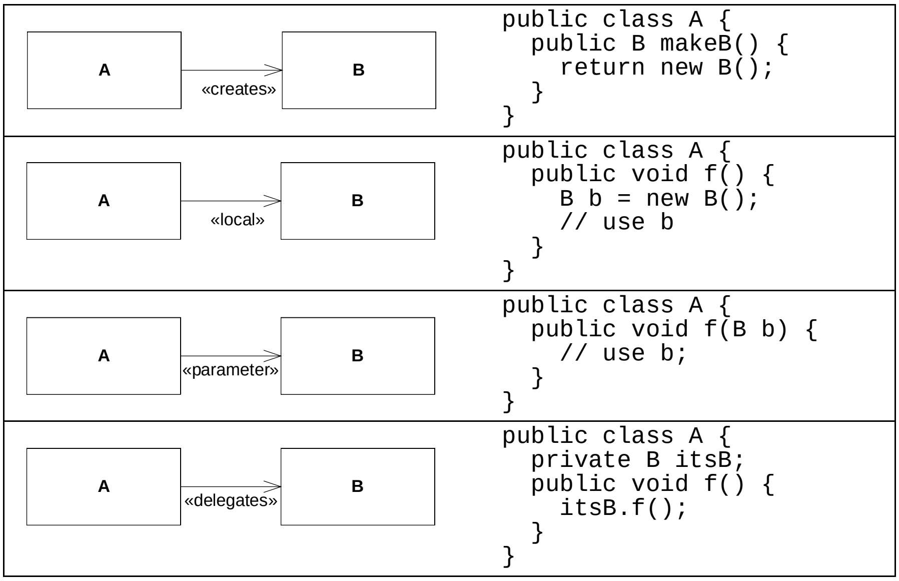
**Figure 3-19**
Simple multiplicity

## Association Stereotypes

Associations can be labeled with stereotypes that change their meaning. Figure 3-20 shows the ones that I use most often. All but the last are standard UML.

The **«creates»** sterotype indicates that the target of the association is created by the source. The implication is that the source creates the target and then passes it around to other parts of the system. In the example I've shown a typical factory.

The **«local»** stereotype is used when the source class creates an instance of the target and holds it in a local variable. The implication is that the created instance does not sur-



```
public class A {
  public B makeB() {
    return new B();
  }
}
```

```
public class A {
  public void f() {
    B b = new B();
    // use b
  }
}
```

```
public class A {
  public void f(B b) {
    // use b;
  }
}
```

```
public class A {
  private B itsB;
  public void f() {
    itsB.f();
  }
}
```

**Figure 3-20**

vive the member function that creates it. Thus, it is not held by any instance variable nor passed around the system in any way.

The **«parameter»** stereotype shows that the source class gains access to the target instance though the parameter of one of its member functions. Again, the implication is that the source forgets all about this object once the member function returns. The target is not saved in an instance variable.

The **«delegates»** stereotype is not a standard part of UML, it is one of my own. However, I find I use it so often that I thought I'd include it here. I use it when the source class forwards a member function invocation to the target. There are a number of design patterns where this technique is applied, such as PROXY, DECORATOR, and COMPOSITE[7]. Since I use these patterns a lot, I find the notation helpful.

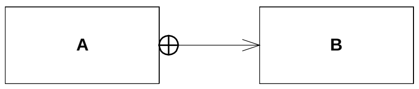
## Inner Classes

Inner (nested) classes are represented in UML with an association adorned with a crossed circle.

## Anonymous Inner Classes

One of Java's most interesting features is anonymous inner classes. While UML does not have an official stance on these, I find the notation in Figure 3-22 works well for me. It is

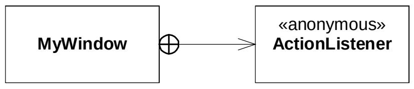
7. [GOF94] p207, 175, 163



```
public class A {
  private class B {
    ...
  }
}
```

Figure 3-21

concise and descriptive. The anonymous inner class is shown as a nested class that is given the «anonymous» stereotype, and is also given the name of the interface it implements.

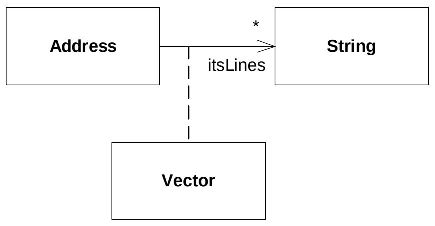


```
public class Window {
  public void f() {
    ActionListener l =
      new ActionListener() {
        // implementation
      };
  }
}
```

Figure 3-22

## Association classes

Associations with multiplicity tell us that the source is connected to many instances of the target; but the diagram doesn't tell us what kind of container class is used. This can be depicted by using an association class as shown in Figure 3-23.



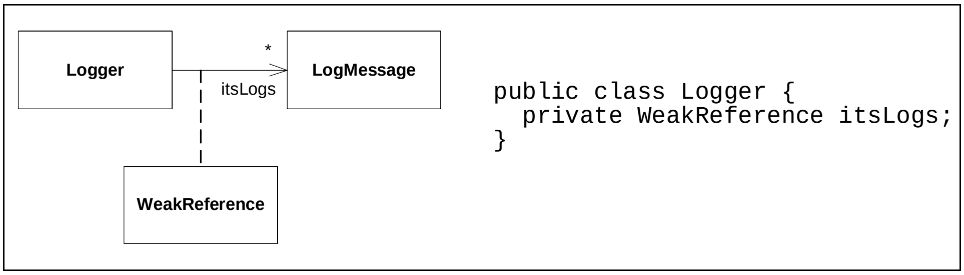
```
public class Address {
  private Vector itsLines;
};
```

**Figure 3-23**
Association class.

Association classes show how a particular association is implemented. On the diagram they appear as a normal class connected to the association with a dashed line. As Java programmers we interpret this to mean that the source class really contains a reference to the association class, which in turn contains references to the target.

Association classes can also be used to indicate special forms of references, such as weak, soft, or phantom references. See Figure 3-24.

On the other hand, this notation is a bit cumbersome and is probably better done with stereotypes as in Figure 3-25.



```
public class Logger {
  private WeakReference itsLogs;
}
```

Figure 3-24

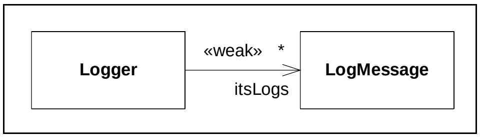
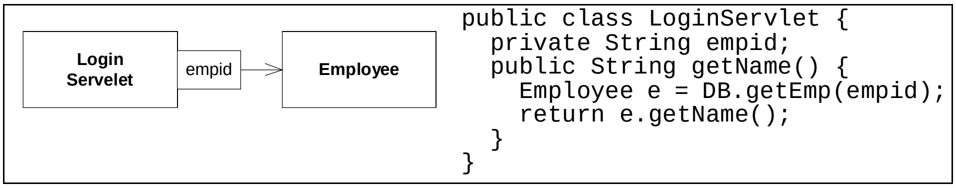


Figure 3-25

## Association Qualifiers

Association qualifiers are used when the association is implemented through some kind of key or token, instead of with a normal Java reference. The example in Figure 3-26 shows a `LoginServlet` associated with an `Employee`. The association is mediated by a member variable named `empid` which contains the database key for the `Employee`.



```
public class LoginServlet {
  private String empid;
  public String getName() {
    Employee e = DB.getEmp(empid);
    return e.getName();
  }
}
```

Figure 3-26

I find this notation useful in rare situations. Sometimes it's convenient to show that an object is associated to another through a database or dictionary key. It is important, however, that all the parties reading the diagram know how the qualifier is used to access the actual object. This is not something that's immediately evident from the notation.

## Conclusion

There are lots of widgets, adornments, and whatchamajiggers in UML. There are so many that you can spend a long time becoming an UML language lawyer enabling you to do what all lawyers can -- write documents nobody can understand.

In this chapter I have avoided most of the arcanities and byzantine features of UML. Rather I have showed you the parts of UML that *I* use. I hope that along with that knowledge I have instilled within you the values of minimalism. Using too little of UML is almost always better than using too much.

## Bibliography

**[Booch94]:** *Object Oriented Analysis and Design with Applications*, Grady Booch, Benjamin Cummings, 1994

**[GOF94]:** *Design Patterns*, Gamma, Helm, Vlissides, Johnson, Addison Wesley, 1994.