Chapter 1 Modeling and the UML

The source code of an object-oriented software application is an implementation, in some programming language, of an object-oriented *model*.

model: an approximate mental or visual image of a software system that serves as an aid in designing the system and in implementing the system in a programming language; a picture is worth a thousand words (or many lines of code)

The *Unified Modeling Language* (*UML*) was created as a standard means of representing models for object-oriented software. The UML consists of several kinds of diagram, but we will restrict our use to the *class diagram*, or *static-structure diagram*. The class diagram allows the modeler to describe *classifiers* (kinds of software entities) and relationships among classifiers. The remainder of this chapter will be devoted to creating a UML class diagram for a small object-oriented software architecture. Our final diagram will show the four primary kinds of relationship between classifiers. Your objective for this chapter should be to understand those four relationships and their expression in the UML.

1.1 First-Person Shooter

You have undoubtedly played, or at least seen played, a first-person shooter game. A key enjoyment in first-person shooter games is accumulating exotic weapons with which to kill one's opponents. We will model the weapons in such a game. In an actual game, our weapons model would be embedded in a much larger model that would include players, buildings, weather, etc.; for the sake of simplicity, we will not consider these other components.

1.1.1 Classifiers

In the author's favorite first-person shooter there are only projectile weapons, like shotguns or rail guns, and hand weapons, like fists or chainsaws. It seems reasonable to distinguish between weapons of these two types in our model, given their striking differences, the most obvious of which is that a projectile weapon has ammo while a hand weapon does not. A UML representation of the distinction between projectile and hand weapons appears below in Figure 1.1.

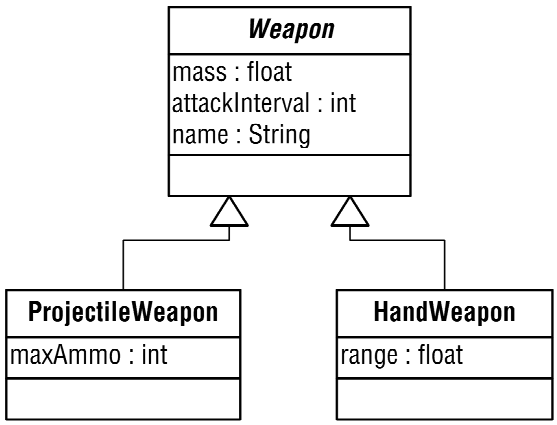

Fig. 1.1. UML class diagram representing two kinds of weapon in a computer game

Notice that each classifier is represented by a rectangle having three compartments. The classifier's name appears in the top compartment and the classifier's *attributes* and *operations* appear in the middle and bottom compartments, respectively.

attribute: a characteristic, or trait, of a classifier; the values of an object's attributes are collectively referred to as the object's *state*

operation: a behavior, or action, of a classifier

Like any kind of real-world entity, a classifier's identity is determined by its characteristics and by the behaviors that follow from those characteristics. Balls can roll because they are round. The attribute *round* gives rise to the behavior *roll*.

1.1.2 Generalization

The classifiers `ProjectileWeapon` and `HandWeapon` are connected to `Weapon` by solid arrows with unfilled triangular heads. This type of arrow simultaneously denotes *generalization* and *specialization*.

generalization: a relationship between classifiers in which one classifier is said to be the kind of the other; e.g., a chair is a kind of furniture is a kind of manmade object, and so we might say that manmade object generalizes furniture, which generalizes chair; specialization is generalization viewed from the other direction–chair specializes furniture, which specializes manmade object; the acronym *ako*, which signifies "a kind of," may be used in place of the term *specializes*

The generalization arrows in Figure 1.1 tell us that `Weapon` generalizes, and therefore is specialized by, both `ProjectileWeapon` and `HandWeapon`.

When classifier P generalizes classifier C, P is said to be the *parent* of C and C is said to be the *child* of P. A child classifier *inherits* all protected and public attributes and operations of its parent. A child classifier extends its parent in some way.

For example, every `Weapon` has a real-number attribute called its `mass`. Because `Weapon` is the parent of `ProjectileWeapon`, a `ProjectileWeapon` also has the `mass` attribute. (A `Weapon`'s `mass` might be used by the game to calculate the `Weapon`'s weight; the weight would be used to adjust the acceleration and top speed of the player carrying the `Weapon`. A rocket launcher will slow its carrier down far more than will a silenced pistol.) In addition, `ProjectileWeapon` has an integer attribute called `maxAmmo`, which serves to distinguish `ProjectileWeapon`s from `HandWeapon`s.

Class `Weapon`'s second attribute, `attackInterval`, is of type `int`. This attribute corresponds to the time required to reload or recharge a projectile weapon, or in the case of a hand weapon, the time required to, say, throw another punch or draw back a whip in preparation for another crack.

1.1.3 Association

A `Weapon` also possesses a `String` called `name`. Because `String` is a class type rather than a primitive type, and we are creating a class diagram, we include the classifier `String` and show its relationship to `Weapon`. The UML term for the relationship is *association*, and it is denoted by a solid line. The term *has-a* is often used in place of *association*: a `Weapon` *has-a* `String`. Figure 1.2 shows the updated diagram.

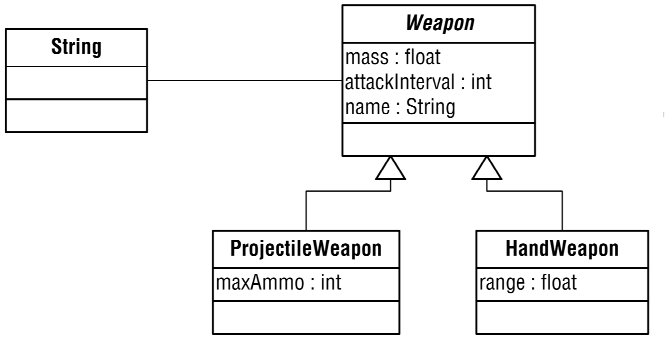

Fig. 1.2. UML class diagram depicting the *has-a* relationship between classes `Weapon` and `String`

Each `Weapon` object will of course need to be displayed, or *rendered*, many times as the game is played, and the `Weapon`'s appearance must vary depending on the perspective from which it is being viewed. This can be accomplished by giving each `Weapon` a collection of images, with each image showing the `Weapon` as it should appear from a particular perspective.

In our model, class `Weapon` will have an array, `images`, of type Bitmap. Figure 1.3 shows this *has-a* relationship between classes `Weapon` and `Bitmap`.

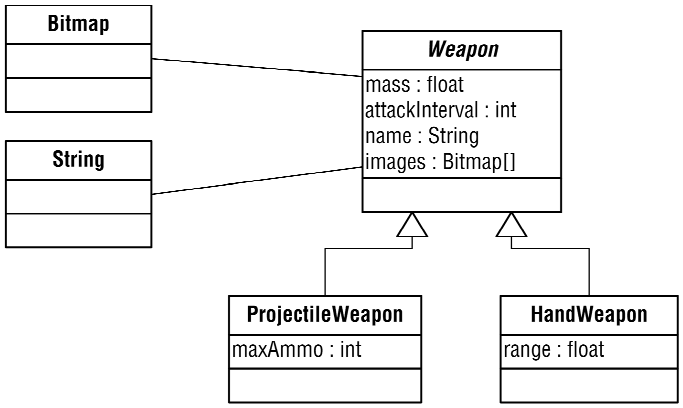

Fig. 1.3. UML class diagram depicting `Weapon`'s association with `Bitmap`

A projectile weapon, like a shotgun, makes a sound when it is fired or reloaded. And hand weapons also make sounds. A chainsaw, for example, idles when it is not being used to cut down an opponent, and makes a loud and rather annoying whine when its throttle is depressed. We will incorporate this characteristic of weapons into our model by giving class `Weapon` an array of type `Sound`. Figure 1.4 shows the new diagram.

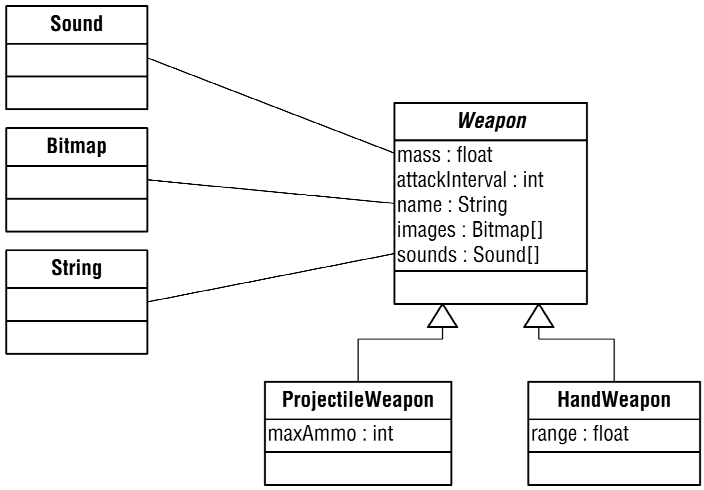

Fig. 1.4. UML class diagram depicting `Weapon`'s association with `Sound`

1.1.4 Dependency

When a `Weapon` object is called upon to render itself, the object must be given certain information about how and where to do so. A `Weapon` might need to know where it is to appear on the display, by how much it should scale its size so as to create the illusion of distance, or how bright it should appear given the light level in its virtual environment. This information is collectively known as the object's current *display context*.

The display context must not be an attribute of the `Weapon` object, for the `Weapon`'s display context at any given moment has nothing to do with the `Weapon` itself, but is instead determined entirely by the `Weapon`'s virtual environment. In other words, a given `Weapon`'s display context is provided when the game requires the object to appear on the display. If the `Weapon` must be displayed again a fraction of a second later, its display context will probably have changed, and so the game must provide the `Weapon` with the new context. The `Weapon` will then use the new context to carry out its rendering operation.

Class `Weapon`'s `render` operation, and that operation's dependence on a fleeting display context, are shown in Figure 1.5. The relationship between classes `Weapon` and `DisplayContext` is called *dependency*, or *uses-a*. A `Weapon` *uses-a* `DisplayContext` to render itself. The dependency relationship is expressed in the UML by a dashed arrow with an open head.

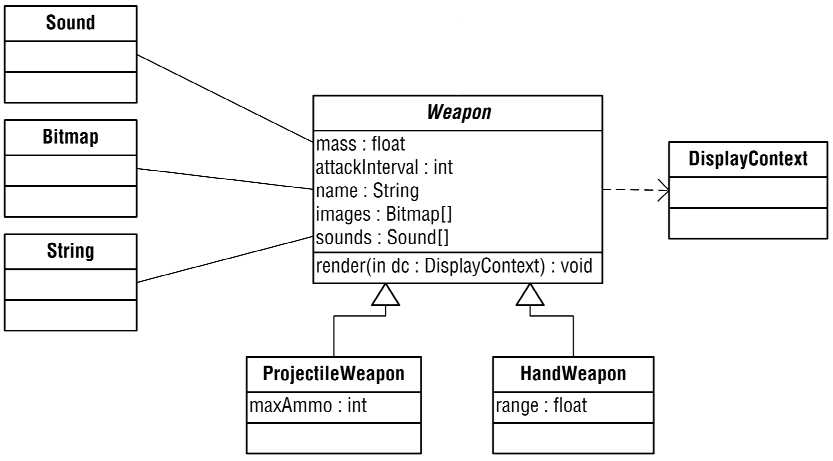

Fig. 1.5. UML class diagram depicting the *uses-a* relationship between classes `Weapon` and `DisplayContext`

1.1.5 Realization

It is typical of first-person shooter games that at least several of each type of weapon are scattered throughout the game world, waiting to be found by players. It is desirable, then, that our model include some mechanism for copying a `Weapon`. It just so happens that the Java platform offers an interface called `Cloneable` for this very purpose. Any Java class that *realizes* interface `Cloneable` includes a method called `clone`. The `clone` method returns a copy of the object that called it.

Figure 1.6 shows the realization relationship between class `Weapon` and interface `Cloneable`. In the UML, realization is indicated by a dashed arrow with an unfilled triangular head. `Cloneable` is *stereotyped* to make clearer the fact that it is an interface rather than a class. The « and » are called *guillemets*.

> We will soon have a closer look at interfaces. For now it suffices to say that realization is a kind of relationship between a class and an interface.

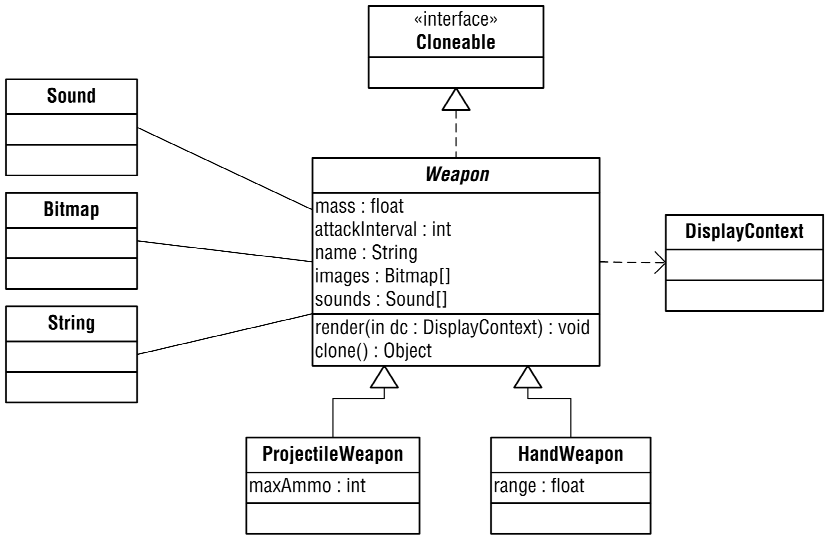

Fig. 1.6. UML class diagram depicting `Weapon`'s realization of `Cloneable`

1.2 From UML to Code

The diagram of Figure 1.6 is not merely a pictorial representation of our model's structure. The diagram is also equivalent to Java code that implements the structure. That code can be generated from the UML diagram by most any UML software package. The programmer may then complete the implementation by coding bodies for the stubbed methods and by adding any fields or methods not shown in the model. The Java code corresponding to the diagram of Figure 1.6 is shown below. Each class definition would of course be in its own file.

```
public class Bitmap {}

public class DisplayContext {}

public class Sound {}

public abstract class Weapon implements Cloneable
{
    protected float mass;
    protected int attackInterval;
    protected String name;
    protected Bitmap[] images;
    protected Sound[] sounds;

    public void render(DisplayContext dc) {}

    public Object clone() {}
}

public class HandWeapon extends Weapon
{
    protected float range;
}

public class ProjectileWeapon extends Weapon
{
    protected int maxAmmo;
}
```

Exercises

1. Define *association*.
2. Define *dependency*.
3. In our weapons model, the dependency between classes `Weapon` and `DisplayContext` occurs by way of an operation's parameter. What other forms can a dependency take?
4. What relationship is missing from Figure 1.6? Hint: Consider your answer to Exercise 3.
5. Extend the model of Figure 1.6 so that it includes ammo.
6. Does the Java platform offer a class that we could use in place of the fictitious class `Bitmap`?
7. Does the Java platform offer a class that we could use in place of the fictitious class `Sound`?
8. Model a class called `PlayingCard`, a deck(s) of which could be used in a Blackjack simulation or interactive game. Class `PlayingCard` should realize `java.io.Serializable`. A `Serializable` object can easily be transmitted across a network, which would be useful to someone developing an online casino application, for example.