Chapter 5

More Object-Oriented Concepts

Peter Müller
Globewide Network Academy (GNA)
pmueller@uu-gna.mit.edu

Whereas the previous lecture introduces the fundamental concepts of object-oriented programming, this lecture presents more details about the object-oriented idea. This section is mainly adopted from [2][1].

5.1 Relationships

In exercise 3.6.5 you already investigate relationships between abstract data types and instances and describe them in your own words. Let's go in more detail here.

A-Kind-Of relationship

Consider you have to write a drawing program. This program would allow drawing of various *objects* such as points, circles, rectangles, triangles and many more. For each object you provide a *class* definition. For example, the point class just defines a point by its coordinates:

```
class Point {
attributes:
  int x, y

methods:
```

[1] This book is only available in German. However, since this is one of the best books about object-oriented programming I know of, I decided to cite it here.

```
    setX(int newX)
    getX()
    setY(int newY)
    getY()
  }
```

You continue defining classes of your drawing program with a class to describe circles. A circle defines a center point and a radius:

```
  class Circle {
  attributes:
    int x, y,
        radius

  methods:
    setX(int newX)
    getX()
    setY(int newY)
    getY()
    setRadius(newRadius)
    getRadius()
  }
```

Comparing both class definitions we can observe the following:

- Both classes have two data elements x and y. In the class *Point* these elements describe the position of the point, in the case of class *Circle* they describe the circle's center. Thus, x and y have the same meaning in both classes: They describe the position of their associated object by defining a point.
- Both classes offer the same set of methods to get and set the value of the two data elements x and y.
- Class *Circle* "adds" a new data element *radius* and corresponding access methods.

Knowing the properties of class *Point* we can describe a circle as a point plus a radius and methods to access it. Thus, a circle is "a-kind-of" point. However, a circle is somewhat more "specialized". We illustrate this graphically as shown in Figure 5.1.

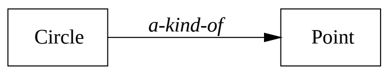

Figure 5.1: Illustration of "a-kind-of" relationship.

In this and the following figures, classes are drawn using rectangles. Their name always starts with an uppercase letter. The arrowed line indicates the direction of the relation, hence, it is to be read as "Circle is a-kind-of Point."

Is-A relationship

The previous relationship is used at the class level to describe relationships between two similar classes. If we create objects of two such classes we refer to their relationship as an "is-a" relationship.

Since the class *Circle* is a kind of class *Point*, an instance of *Circle*, say *acircle*, is a *point*[2]. Consequently, each circle behaves like a point. For example, you can move points in x direction by altering the value of x. Similarly, you move circles in this direction by altering their x value.

Figure 5.2 illustrates this relationship. In this and the following figures, objects are drawn using rectangles with round corners. Their name only consists of lowercase letters.

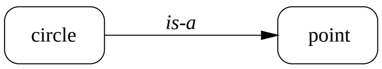

Figure 5.2: Illustration of "is-a" relationship.

Part-Of relationship

You sometimes need to be able to build objects by combining them out of others. You already know this from procedural programming, where you have the structure or record construct to put data of various types together.

Let's come back to our drawing program. You already have created several classes for the available figures. Now you decide that you want to have a special figure which represents your own logo which consists of a circle and a triangle. (Let's assume, that you already have defined a class *Triangle*.) Thus, your logo consists of two *parts* or the circle and triangle are *part-of* your logo:

```
class Logo {
attributes:
  Circle circle
  Triangle triangle

methods:
  set(Point where)
}
```

We illustrate this in Figure 5.3.

Has-A relationship

This relationship is just the inverse version of the part-of relationship. Therefore we can easily add this relationship to the part-of illustration by adding arrows in the other direction (Figure 5.4).

[2]We use lowercase letters when we talk at the object level.

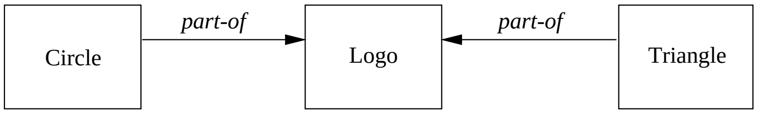

Figure 5.3: Illustration of "part-of" relationship.

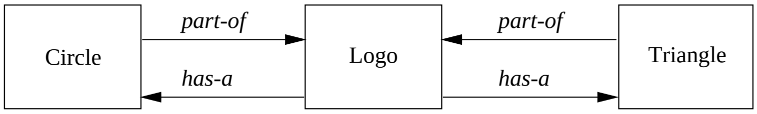

Figure 5.4: Illustration of "has-a" relationship.

5.2 Inheritance

With inheritance we are able to make use of the a-kind-of and is-a relationship. As described there, classes which are a-kind-of another class share properties of the latter. In our point and circle example, we can define a circle which *inherits from* point:

```
class Circle inherits from Point {
attributes:
  int radius

methods:
  setRadius(int newRadius)
  getRadius()
}
```

Class *Circle* inherits all data elements and methods from point. There is no need to define them twice: We just use already existing and well-known data and method definitions.

On the object level we are now able to use a circle just as we would use a point, because a circle is-a point. For example, we can define a circle object and set its center point coordinates:

```
Circle acircle
acircle.setX(1)        /* Inherited from Point */
acircle.setY(2)
acircle.setRadius(3)   /* Added by Circle */
```

"Is-a" also implies, that we can use a circle everywhere where a point is expected. For example, you can write a function or method, say *move()*, which should move a point in x direction:

```
move(Point apoint, int deltax) {
  apoint.setX(apoint.getX() + deltax)
}
```

As a circle inherits from a point, you can use this function with a circle argument to move its center point and, hence, the whole circle:

```
Circle acircle
  ...
move(acircle, 10)   /* Move circle by moving */
                    /* its center point */
```

Let's try to formalize the term "inheritance":

Definition 5.2.1 (Inheritance) **Inheritance** *is the mechanism which allows a class A to inherit properties of a class B. We say "A inherits from B". Objects of class A thus have access to attributes and methods of class B without the need to redefine them.*

The following definition defines two terms with which we are able to refer to participating classes when they use inheritance.

Definition 5.2.2 (Superclass/Subclass) *If class A inherits from class B, then B is called* **superclass** *of A. A is called* **subclass** *of B.*

Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass.

In the literature you may also find other terms for "superclass" and "subclass". Superclasses are also called *parent classes*. Subclasses may also be called *child classes* or just *derived classes*.

Of course, you can again inherit from a subclass, making this class the superclass of the new subclass. This leads to a hierarchy of superclass/subclass relationships. If you draw this hierarchy you get an *inheritance graph*.

A common drawing scheme is to use arrowed lines to indicate the inheritance relationship between two classes or objects. In our examples we have used "inherits-from". Consequently, the arrowed line starts from the subclass towards the superclass as illustrated in Figure 5.5.

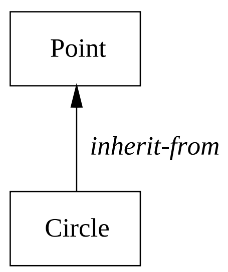

Figure 5.5: A simple inheritance graph.

In the literature you also find illustrations where the arrowed lines are used just the other way around. The direction in which the arrowed line is used, depends on how the corresponding author has decided to understand it.

Anyway, within this tutorial, the arrowed line is always directed towards the superclass.

In the following sections an unmarked arrowed line indicates “inherit-from”.

5.3 Multiple Inheritance

One important object-oriented mechanism is multiple inheritance. Multiple inheritance does **not** mean that multiple subclasses share the same superclass. It also does **not** mean that a subclass can inherit from a class which itself is a subclass of another class.

Multiple inheritance means that one subclass can have *more than one* superclass. This enables the subclass to inherit properties of more than one superclass and to “merge” their properties.

As an example consider again our drawing program. Suppose we already have a class *String* which allows convenient handling of text. For example, it might have a method to *append* other text. In our program we would like to use this class to add text to the possible drawing objects. It would be nice to also use already existing routines such as *move()* to move the text around. Consequently, it makes sense to let a drawable text have a point which defines its location within the drawing area. Therefore we derive a new class *DrawableString* which inherits properties from *Point* and *String* as illustrated in Figure 5.6.

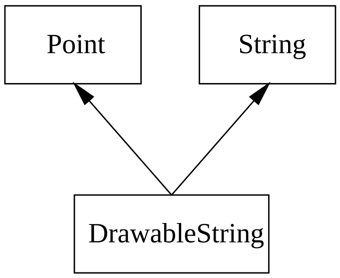

Figure 5.6: Derive a drawable string which inherits properties of *Point* and *String*.

In our pseudo language we write this by simply separating the multiple superclasses by comma:

```
class DrawableString inherits from Point, String {
attributes:
            /* All inherited from superclasses */

methods:
```

```
            /* All inherited from superclasses */
}
```

We can use objects of class *DrawableString* like both points and strings. Because a *drawablestring* is-a *point* we can move them around

```
DrawableString dstring
...
move(dstring, 10)
...
```

Since it is a *string*, we can append other text to them:

```
dstring.append("The red brown fox ...")
```

Now it's time for the definition of multiple inheritance:

Definition 5.3.1 (Multiple Inheritance) *If class A inherits from more than one class, ie. A inherits from B_1, B_2, ..., B_n, we speak of* **multiple inheritance**. *This may introduce* **naming conflicts** *in A if at least two of its superclasses define properties with the same name.*

The above definition introduce *naming conflicts* which occur if more than one superclass of a subclass use the same name for either attributes or methods. For an example, let's assume, that class *String* defines a method *setX()* which sets the string to a sequence of "X" characters[3]. The question arises, what should be inherited by *DrawableString*? The *Point*, *String* version or none of them?

These conflicts can be solved in at least two ways:

- The order in which the superclasses are provided define which property will be accessible by the conflict causing name. Others will be "hidden".
- The subclass must resolve the conflict by providing a property with the name and by defining how to use the ones from its superclasses.

The first solution is not very convenient as it introduces implicit consequences depending on the order in which classes inherit from each other. For the second case, subclasses must explicitly redefine properties which are involved in a naming conflict.

A special type of naming conflict is introduced if a class *D* multiply inherits from superclasses *B* and *C* which themselves are derived from one superclass *A*. This leads to an inheritance graph as shown in Figure 5.7.

The question arises what properties class *D* actually inherits from its superclasses *B* and *C*. Some existing programming languages solve this special inheritance graph by deriving *D* with

[3] Don't argue whether such a method makes really sense or not. It is just introduced for illustrating purposes.

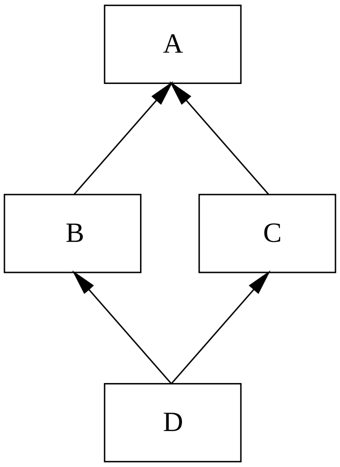

Figure 5.7: A name conflict introduced by a shared superclass of superclasses used with multiple inheritance.

- the properties of A plus
- the properties of B and C **without** the properties they have inherited from A.

Consequently, D cannot introduce naming conflicts with names of class A. However, if B and C add properties with the same name, D runs into a naming conflict.

Another possible solution is, that D inherits from both inheritance paths. In this solution, D owns **two** copies of the properties of A: one is inherited by B and one by C.

Although multiple inheritance is a powerful object-oriented mechanism the problems introduced with naming conflicts have lead several authors to “doom” it. As the result of multiple inheritance can always be achieved by using (simple) inheritance some object-oriented languages even don't allow its use. However, carefully used, under some conditions multiple inheritance provides an efficient and elegant way of formulating things.

5.4 Abstract Classes

With inheritance we are able to force a subclass to offer the same properties like their superclasses. Consequently, objects of a subclass behave like objects of their superclasses.

Sometimes it make sense to only describe the properties of a set of objects without knowing the actual behaviour beforehand. In our drawing program example, each object should provide a method to draw itself on the drawing area. However, the necessary steps to draw an objects depends on its represented shape. For example, the drawing routine of a circle is different from the drawing routine of a rectangle.

Let's call the drawing method *print()*. To force every drawable object to include such method, we define a class *DrawableObject* from which every other class in our example inherits general properties of drawable objects:

```
abstract class DrawableObject {
attributes:

methods:
  print()
}
```

We introduce the new keyword **abstract** here. It is used to express the fact that derived classes must "redefine" the properties to fulfill the desired functionality. Thus from the abstract class' point of view, the properties are only *specified* but not fully *defined*. The full definition including the semantics of the properties must be provided by derived classes.

Now, every class in our drawing program example inherits properties from the general drawable object class. Therefore, class *Point* changes to:

```
class Point inherits from DrawableObject {
attributes:
  int x, y

methods:
  setX(int newX)
  getX()
  setY(int newY)
  getY()
  print()    /* Redefine for Point */
}
```

We are now able to force every drawable object to have a method called *print* which should provide functionality to draw the object within the drawing area. The superclass of all drawable objects, class *DrawableObject*, does not provide any functionality for drawing itself. It is not intended to create objects from it. This class rather specifies properties which must be defined by every derived class. We refer to this special type of classes as *abstract classes*:

Definition 5.4.1 (Abstract Class) *A class A is called* **abstract class** *if it is only used as a superclass for other classes. Class A only specifies properties. It is not used to create objects. Derived classes must define the properties of A.*

Abstract classes allow us to structure our inheritance graph. However, we actually don't want to create objects from them: we only want to express common characteristics of a set of classes.

5.5 Exercises

1. Inheritance. Consider the drawing program example again.

   (a) Define class *Rectangle* by inheriting from class *Point*. The point should indicate the upper left corner of the rectangle. What are your class attributes? What additional methods do you introduce?

   (b) All current examples are based on a two-dimensional view. You now want to introduce 3D objects such as spheres, cubes or cuboids. Design a class *Sphere* by using a class *3D-Point*. Specify the role of the point in a sphere. What relationship do you use between class *Point* and *3D-Point*?

   (c) What functionality does *move()* provide for 3D objects? Be as precise as you can.

   (d) Draw the inheritance graph including the following classes *DrawableObject*, *Point*, *Circle*, *Rectangle*, *3D-Point* and *Sphere*.

   (e) Have a look at the inheritance graph of Figure 5.8.

   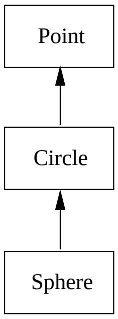

   Figure 5.8: Alternative inheritance graph for class *Sphere*.

   A corresponding definition might look like this:

   ```
   class Sphere inherits from Circle {
   attributes:
     int z        /* Add third dimension */

   methods:
     setZ(int newZ)
     getZ()
   }
   ```

   Give reasons for advantages/disadvantages of this alternative.

2. Multiple inheritance. Compare the inheritance graph shown in Figure 5.9 with that of Figure 5.7. Here, we illustrate that *B* and *C* have each their own copy of *A*.

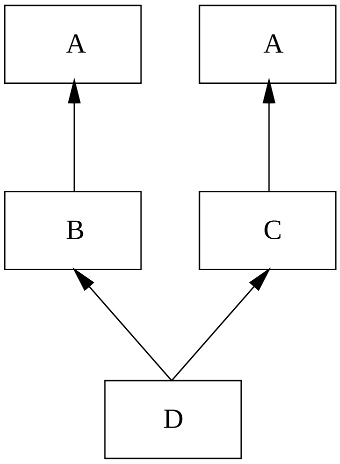

Figure 5.9: Illustration of the second multiple inheritance semantics.

What naming conflicts can occur? Try to define cases by playing with simple example classes.