By John A. Grosberg: John, a principal staff engineer for Motorola, is currently on the team developing satellite software for the Iridium communication system. He can be reached at grosberg@email.mot.com. (Java-ified by Binkley)

When developing object-oriented designs, one of your goals is to define a set of classes that are related hierarchically through inheritance. One school of thought, call it the modeling school, holds that each class in an object-oriented design should represent a concept from the problem domain and that a class inheritance hierarchy should model the “is-a” relationships among the concepts. (Two concepts participate in an “is-a” relationship if every instance of one “is an” instance of the other. For example, every instance of Dog is an instance of Animal.) The second school of thought, which I’ll call the “implementation-oriented” school, holds that inheritance is a convenience for the reuse and extension of class behavior.

Both viewpoints are valid, and each is useful in its appropriate phase of the software-development cycle. The modeling point of view tends to be most helpful in the earlier phases of development (analysis and top-level design, for instance). The implementation approach tends to prevail in the later phases (detail design and code).

In this article, I’ll focus on the modeling viewpoint, examining issues that are often overlooked when this viewpoint is applied. In doing so, I’ll work with two kinds of diagrams: Venn diagrams and Object Models. Venn diagrams are traditionally used to show logical relationships among concepts. The Object Model was developed by James Rumbaugh et al. in Object-Oriented Modeling and Design to show relationships among classes and instances. Venn diagrams help explain the logic behind the design principles; Object Models show their application to OOD.

The Problem: Figure 1 is a Venn diagram representing a single class (conceptual) hierarchy. The rectangular box represents the “universe,” the class of all classes of interest to us. You could think of it as a context diagram – it defines the scope of our interest. In this case, the universe is rectangles. The circular region serves to isolate a particular part, or subclass, of that universe for attention. The Venn diagram is a conceptual, or logical, map of the problem domain.

Any object or instance we discuss belongs to some compartment in the diagram. Any rectangle that is a square belongs inside the circle, any that is not a square belongs outside. Think about how you would implement this diagram in an object-oriented language. Typically, the first attempt would be to define two classes, Rectangle and Square, with

- Each Square is carrying around a length and width, when it really only needs a single value, since length and width are always equal.
- It is possible to create a Square instance, and then, using the inherited set_length and/or set_width member functions, make the length and width unequal, violating the definition of a square.
- A simpler calculation is possible for a Square’s perimeter (4×side) but can only be implemented (inelegantly) by selecting either length or width as a stand-in for side.

These problems are not incidental to this particular example. They are the result of an incorrect mapping of the Venn diagram into an Object Model. This mapping arises from an inadequate understanding of the Venn diagram and the “is-a” relationship.
Abstract. Alternative names for the abstract class might be OtherRectangle
(on the right). A black dot represents an instance of the class because these two classes completely cover the Rectangle because all you know about any instance that is not a Square is that it is a Rectangle. Alternative names for the complement class might be Rectangles
that cannot be guaranteed to be Squares, that is, to have equal widths and lengths.

What about a Rectangle instance that has both sides equal? Is it a Square? In a geometrical or mathematical context, a rectangle becomes a square, incurring all the properties of a square, whenever its length becomes equal to its width. This means that in Figure 3, a dot can cross the boundary between Square and Rectangle if its attributes change. By contrast, in a statically typed language, such as Java, an instance’s type does not change merely because the values of its attributes (member data) change. After all, there is more to a class than its data attributes: It also has operations or functions that cannot change when the data changes. Thus, even if a Rectangle’s sides happen to be equal, its type doesn’t change to Square – once a Rectangle, always a Rectangle. This means that in Figure 3, for a statically typed language, the dots cannot move across a boundary. Nevertheless, the principles advocated here are not affected by whether or not the dots can cross the boundary, because at any one time, a dot must be in one region or the other – it can’t be in both. Thus, if you had a dynamically typed language, where an instance’s type could change depending on its attribute values, the Venn diagram of Figure 3 would still map to the Object Model of Figure 4. You can see in Listing2.java, which is an implementation of the improved Object Model of Figure 4, that the three problems mentioned earlier have disappeared. Square is not carrying around any extra data or function baggage, and both concrete classes can be extended or specialized as much as desired without affecting each other or their base class. A Square can no longer be given unequal sides. You have collected what is truly common to both classes and factored it into the abstract base class.

In this code you now have a true universe class, ARectangle, which, while it can’t have separate instances of its own, can be used to create pointers and/or references to instances of its derived classes. Thus, all instances of ARectangle must be either instances of Rectangle or of Square as in the logic and the Venn diagrams. The abstract class ARectangle is the point in our class hierarchy at which polymorphism is centered so that a call to the member function area() via an ARectangle pointer that points to an instance of Square (pAR.area()) will get the special calculation appropriate to a Square. Likewise, a call to a similar pointer that indicates a Rectangle will invoke the calculation appropriate to that class.

Abstract and Concrete Classes: Thus far, I’ve discussed the construction of a class hierarchy consisting of only two types of classes, which are “duals,” or polar opposites:
Classes that have subclasses but no instances, which are called “abstract classes.”

Classes that have instances but no subclasses, which are called “concrete classes.” To properly implement an “is-a” hierarchy, those are the only types of classes allowed.

An abstract class defines a family of classes. Thus an abstract provides a single place to hold the common features of its “descendants.” Common interfaces and behaviors are united in the function members of the abstract class. Common attributes are collected in the data members of the abstract class. These interfaces, behaviors, and attributes are shared by the abstract and concrete descendants of the abstract class through the inheritance mechanism. In Java, an abstract class has at least one abstract function.

A concrete class defines a family of object instances. It adds the behaviors and attributes that are unique to a group of instances to the behaviors and attributes inherited from its parent abstract classes. In Java, a concrete class has no abstract functions.

In an “is-a” hierarchy, using a “mixed class” (one that has both instances and subclasses) is forbidden because it leads to the difficulties encountered earlier. Java allows you to subclass a concrete class, but doing so is inconsistent with implementing a true “is-a” hierarchy.

The Mapping Rules: You can codify the mapping from Venn diagram to Object Model to code into two rules that apply to “is-a” hierarchies of any complexity:

- Rule 1. In a Venn diagram, every region that includes (or overlaps) another region will become two classes on the Object Model. One of those classes will be an abstract supertype of the other. The subtype will be concrete.
- Rule 2. Every region that does not include or overlap another region will map to a single concrete class that is a subclass of the abstract class(es) mapped from the immediately including region.

A Venn diagram with two subclasses permits three possible relationships:

- Case 1. The two subclasses are mutually exclusive.
- Case 2. One subclass includes the other.
- Case 3. The two subclasses overlap.

Single Inheritance: The first two cases are examples of single inheritance, which is said to hold when a subclass is subordinate to only one immediate parent class. Figure 5 shows both the Venn diagram and the Object Model for Case 1. (In this and subsequent figures, the prefix “A” emphasizes that a class is an abstract class.) From Rule 1, since Animal contains other classes (Dog and Cat), it must become two classes on an Object Model – one an abstract superclass (AAAnimal) and one a concrete subclass (Animal). The other subclasses,

Fig. 5. Venn diagram and Object Model for two disjoint classes.

Dog and Cat, contain no subclasses and therefore become concrete classes by Rule 2. Since the Venn diagram shows them within the Animal region, they are drawn as subclasses of AAAnimal. Concrete classes are always “leaves” of the class hierarchy. Abstract classes are never leaves.

Rule 1 reminds you that the identified subclasses (Dog, Cat, and Animal on the Venn diagram) are not the only subclasses – there is usually a “difference class.” The difference class is the subclass on a Venn diagram that remains after all named subclasses are subtracted from a superclass. The reason for staying aware of the difference class is so that you do not confuse it with its superclass, as in Figure 2.

In Figure 5, the difference class is ((not Dog) and (not Cat)). In the Object Model of Figure 5, I called it “Animal.” Members of that class can be thought of as “Animals that cannot be guaranteed to be either a Dog or a Cat.” Notice that as you add classes, the meaning of the difference class Animal changes. For example, if we add Gerbil as a subclass of AAAnimal, then Animal becomes ((not Dog) and (not Cat) and (not Gerbil)).

In a particular application, if a difference class is null or of no interest, it can be left out. Implementing the difference class is usually necessary when dealing with traditional taxonomies such as geometric shapes (rectangles, squares, and so on), types of numbers (complex, real, rational, and so on), and biological taxonomies as in the current example. If your “is-a” hierarchy does not need it, leave it out. Just make sure you use only abstract and concrete classes, and no mixed classes.

In Figure 6, which illustrates Case 2, there is a three-level class structure. How many classes are there? If you count the difference classes, you get Animal, Mammal, Dog, Mammals that are not Dogs, and Animals that are not Mammals, for a total of five classes.

Following the rules leads you from the Venn diagram in Figure 6 to the Object Model in Figure 7. Note that both Animal and Mammal have become pairs of classes, one abstract and one concrete.
Multiple Inheritance: Turning to Case 3, the Venn diagram in Figure 8 shows two overlapping classes. An instance of the intersection class has characteristics of both Mammal and EggLaying Animals. This is an example of multiple inheritance, which is said to hold when a subclass has more than one immediate parent. For the curious, actual biological examples (concrete subclasses) of the EggLaying-Mammal class are the platypus class and the spiny anteater class (I’m no expert – I had to look it up).

In the Object Model, the triangle is filled in to indicate that the subclasses are not mutually exclusive – there is overlap among at least some of its subclasses. In contrast to the empty triangle, which I used to represent the Exclusive OR relationship, the solid triangle represents the Inclusive OR relationship. The Object Model shows the multiple inheritance of the EggLaying-Mammal class simply, by connecting the subclass to both of its superclasses. The difference classes Animal, Mammal, and EggLaying are shown, and the abstract classes AAnimal, AMammal, and AEggLaying are added in accordance with the rules.

As an exercise, try drawing the Object Model for the case of multiple inheritance in Figure 9(a). The resulting Object Model has 15 classes, comprising 7 abstract and 8 concrete classes; see Figure 9(b). If it will help you to have meaningful classes in the figure, then try letting W be Animals, X be LandAnimals, Y be AirAnimals, and Z WaterAnimals. You can easily name animals for each of the concrete classes. This might be a case where the difference class W is null.

Rationalizing Legacy Hierarchies with Delegation – commented out

Summary: Inheritance is used to implement “is-a” relationships, but inheritance is not the same thing as an “is-a” relationship. Inheritance is simply a language mechanism with characteristics that can be used for a variety of implementation objectives and given various interpretations. My focus here is on just one of those applications of inheritance – the “is-a” relationship. The rules advocated here are essential for the “is-a” interpretation of inheritance, but they do not necessarily apply to any of the other uses or interpretations of inheritance. They also aren’t intended to preclude the many uses of mixed classes that might be appropriate in implementing working code outside the “is-a” realm.

References:

Copyright © 1997, Dr. Dobb’s Journal