Javatecture I

Are You Writing Code or Building Objects ?

James W. Cooper

Most of us who write programs daily are familiar with a number of languages. Each time we learn a new one we bring all of our accumulated experience in algorithm development and coding practices to learning the newest language. You've probably gone through several programming sea-changes already, such as Fortran or Basic to C, Cobol to C, C to Visual Basic, C to C++ and something to Java. Even if you already know a number of languages, I think you'll find that your approach to Java is different because of its object-oriented (OO) structure. You'll probably discover that Java makes you think differently about the process of programming than you have in the past.

Since programmers discovered that you could write perfectly good C and compile it using a C++ compiler, many C++ programs are really C program with double-slash comments. Similarly, while you can't really write Java in C-style, you can import some of the habits that served you well in C, but that might not be the best way to write Java.

Java makes you write everything using classes or *objects*. Therefore, you have to start thinking in terms of objects and designing your program with objects in mind. This is actually pretty easy, since almost all programming problems devolve into some logical entities that you might well represent as objects. They could represent physical objects (bills, requisitions, payments) or they might represent program constructs (trees, names, queues) , but the division is pretty easy most of the time.

In fact, there might be several kinds of object structures you *could* build. And it is very likely that more than one of them will work fine. It is also rather likely that you'll consider and start implementing more than one before you're satisfied.

Objects for Soccer Teams

For example, suppose you are writing a program to keep statistics for your daughter's soccer team. It's pretty clear that there will be an object representing each player on the team. Then, depending on the data you have available, there might be objects representing matches, points scored, assists, and probably a lot more if you really know the sport well.

So now we have this pile of objects laying on our desk: kids, scores, matches. Oh, and what about their addresses and phone numbers? Where do they go? Now we begin to see the real discipline of OO programming in Java. It's not designing the objects, it's designing the *interactions between objects* that is the heart of the matter. This is where we will be spending our time in this column: worrying about these interactions and learning whether someone has already worked out solutions for some of these problems.

Inheritance and Containment

Let's think about this soccer team a little more. Soccer teams have players, and they all play positions (more or less well depending on their experience). One of these positions is Goalie, and the Goalie has special properties other players do not have. Not only can she touch the ball with her hands (which we don't have to abstract into any class), she is the person who defends the goal and therefore the Goalie object will have some additional properties, such as a count of goals saved and goals scored against her. But the Goalie is still a kind of Player, she just has additional properties, and thus it is appropriate that we define a goalie class by *deriving* it from the *player* class. In simple diagram terms, we draw the inheritance diagram shown in Figure 1, showing that the Goalie class inherits from the Player class. Note that it is customary to have the arrow pointing to the base class from the derived class.

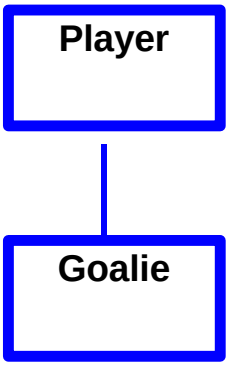

Figure 1

Now one of the most important relations in object oriented programming is *inheritance* as we just showed above. We create a new derived class which has some special properties but still has all the properties of the original. We call this an "is-a" relationship. A Goalie *is a* type of player.

But this is not the only important part of OO programming in Java.. We have also proposed objects representing goals and assists, matches played and so forth. It doesn't look like they inherit from anything we've talked about. How do we relate them to our players?

The answer is that each player has goals, assists, matches and so forth as properties. This means that the player *contains* instances of each of those classes. We might draw this as shown in Figure 2.

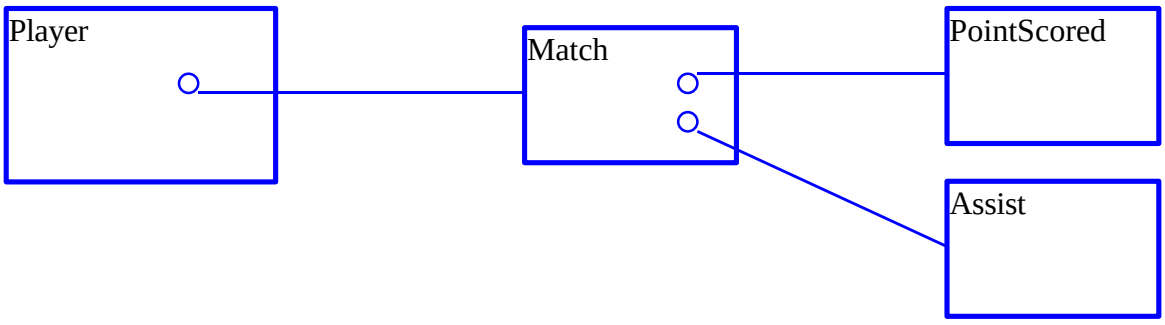

Figure 2

This diagram means that each player contains one (or more) instances of Match and each Match contains one or more instances of PointScored and one or more instances of Assist. We call this a “has-a” relationship. A Player *has a* number of points scored.

Now what about those PointScored and Assist classes? Aren’t they just numbers? Well, they might be. But each scoring action could have additional characteristics associated with it, such as who assisted, the air temperature, the type of kick and position from which the score was made. We’ll create an object for each of these groups of data values.

```
//class to hold data about one point scored
class PointScored {
float airTemperature;   //temperature at time of goal
int kickType;           //type of kick
int scorePosition;      //position from which score was made

public PointScored(float temperature, int kick, int position) {
      //copy all the values into the object
      airTemperature = temperature;
      kickType = kick;
      scorePosition = position;
      }
}
```

Now we experience our first “Aha!” about our object design. We realize that we are going to keep the same information about both scores and assists, and that we really only need a ScoreAction class in which we can ask whether the action is a score or an assist.

```
//class to hold data about score or assist
class ScoreAction {
float airTemperature;   //temperature at time of goal
int kickType;           //type of kick
int scorePosition;      //position from which score was made
boolean scored;         //true if point was scored, false means assist

public ScoreAction(float temperature, int kick,
             int position, boolean score) {
      //copy all the values into the object
      airTemperature = temperature;
      kickType = kick;
      scorePosition = position;
      scored = score;         //true if score
      }
      //---------------------------
      public boolean isScore() {
      return scored;          //return true if a point was scored
      }
}
```

And yes, we are ignoring the case of failed scoring attempts to keep this example simple.

But What Contains What?

We’ve outlined a Player object and then alluded to matches, points and assists. How do we organize all this stuff? The easiest way to implement an open-ended list of data in Java

is by using a Vector. A Vector is just an unbounded array to which we can add data and from which we can retrieve data. Each Player object will contain a Vector containing a list of matches, and each Match object will contain a Vector representing a list of scoring actions.

```
class Match {
  Vector scores;        //list of scoring actions
  String name;          //name of match
  Date date;            //date of match
}
class Player {
  Vector matches;       //list of matches for that player
  String player_name;   //name of player
}
```

Most importantly, each player and match will have public methods to give us access to this information in a convenient fashion.

```
class Match {
  Vector scores;
  String name;

public int getScoreCount() {
  return scanScores(true);    //count of actions which scored
}

public int getAssistCount() {
  return scanScores(false);   //count of assist actions
}
//scan vector and count scores or assists
private int scanScore(boolean score_flag) {
int count  = 0;
ScoreAction sa;
  Enumeration e = score.elements();       //get Iterator
  while(score.hasMoreElements()           //scan thru vector
    {
     sa = (ScoreAction)e.nextElement();   //go through list
      if (sa.isScore() == score_flag)     //compare with boolean
            count++;                      //count it if it matches
    }
}
```

Finally, for each player, we scan through the vector of Matches to sum up all points scored and assists in a fashion just analogous to that we illustrated for the scores Vector above. You'll find the complete code for this simple exercise available on the Java Pro FTP site.

Review: Using Inheritance and Containment

We've seen two of the most basic methods of relating objects here: inheritance and containment. While many object-oriented mavens insist that inheritance is the most important part of a "real" object-oriented language, we've just seen that we used containment several times and inheritance only once in our soccer program. The fact that containment is more important is emphasized by the extremely popular and well-regarded "Gang of Four" book on Design Patterns(1). They clearly state in chapter 2: "Favor

containment over inheritance." You can see why: classes which contain other classes can occur very frequently indeed and may be the more powerful construct.

Next: Enumeration

You may have noticed the following line in the above example:

```
Enumeration e = score.elements();         //get Iterator
```

In Java, both the Vector and Hashtable classes provide this **elements()** method as a way of obtaining an Enumeration class to scan through the array an element at a time. Using this Enumeration interface, you can obtain these items sequentially:

```
while ( x.hasMoreElements() )
  {
      z = x.nextElement();
  }
```

You can include an **elements** method in any class you write, and users can then move through the data that class contains without having to understand the details of that class's construction.

This approach is so popular, that it has been given a name: the *Iterator* Design Pattern. This pattern is one of the simplest of a series of very useful programming patterns which are catalogued in the "Gang of Four" book and a number of others. We'll be talking about more of these useful patterns and how you can use them to write simple, but elegant programs in the columns to come.

1. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley: Reading, MA, 1995.

James W. Cooper is a computer science researcher and the author of twelve books: the two most recent of which are on Java.