The Factory Down the Road

James W. Cooper

A few months ago, we talked about the basic Factory pattern. That pattern is widely used all through the programming community, in any number of object oriented languages. Briefly, this pattern is simply a class which returns one of a number of related classes based on some input parameter.

For example, in Figure 1, we see a base class X that has two derived classes XY and XZ.

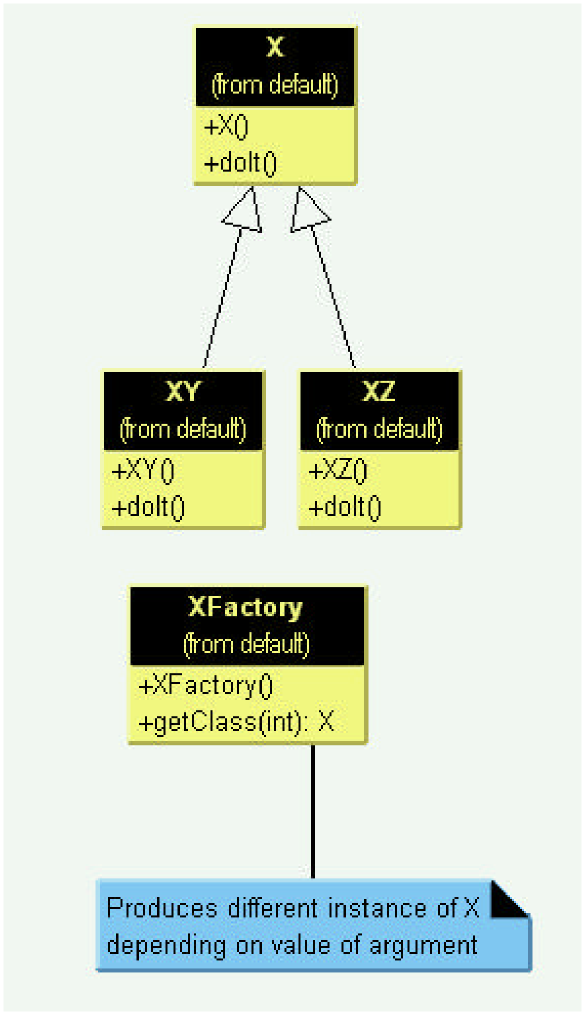


Figure 1- A simple class X, its two derived classes and a simple Factory.

Then we write another class called XFactory which has a method

```
public X getX(int selector) {
   switch (selector) {
      case 0:
            return new XY();
      case 1:
            return new XZ();
      }
}
```

This **getX** method returns an instance of XY or XZ depending on the value of some input parameters, such as the simple **selector** argument we show here. Obviously the computation that decides which of several of child classes to return can be as complex as needed. The important point is that the Factory returns one of the classes but the calling program never needs to know which one: they all have the same public methods but different internal implementations. These simple factories can be useful in developing programs that effectively conceal their implementation details inside classes and gain more flexibility in the process.

The Factory Method Pattern

A more sophisticated and powerful kind of factory is described in the well-known Gang of Four's Factory Method Pattern. This pattern does not actually have a decision point where one derived class is directly selected over another class. Instead, programs written to this pattern define an abstract class that creates objects, but lets each subclass decide which object to create. This sounds like a really powerful approach, but like many new concepts it's a little easier to understand with a concrete example.

After a little thought, it occurred to me that a pretty simple example can be drawn from the way swimmers are seeded into lanes in a swim meet. When swimmers compete in multiple heats in a given event, they are sorted to compete from slowest in the early heats to fastest in the last heat, and arranged within a heat with the fastest swimmers in the center lanes. This is referred to as *straight seeding*.

Now, when swimmers swim in championships, they frequently swim the event twice. During preliminaries everyone competes and the top 12 or 16 swimmers return to compete against each other at finals. In order to make the preliminaries more equitable, the top heats are *circle* seeded, so that the fastest three swimmers are in the center lane in the fastest three heats, the second fastest three swimmers in the next to center lane in the top three heats. Typically, swimmers swim the shorter events twice, but frequently they swim the longer events only once at championships. One might think that this is to prevent exhaustion but those who have ever watched 10 heats of the 1650 yd freestyle realize that it has the gripping excitement of watching paint dry, and the reason may well be spectator fatigue.

OK, enough sports, how do we build some objects to implement this seeding scheme and illustrate the Factory Method. First, let's design an abstract Event class:

```
public abstract class Event  {
   protected int numLanes;
   protected Vector swimmers;

   public abstract Seeding getSeeding();
   public abstract boolean isPrelim();
   public abstract boolean isFinal();
   public abstract boolean isTimedFinal();
 }
```

This defines the methods simply without any necessity of filling in the methods. Then we can derive concrete classes from the Event class, called PrelimEvent and

TimedFinalEvent. The only difference between these classes is that one returns one kind of seeding and the other returns a different kind of seeding.

We also define an abstract Seeding class having the following methods:

```
public abstract class Seeding {
   public abstract Enumeration getSwimmers();
   public abstract int getCount();
   public abstract int getHeats();
}
```

We derive two concrete seeding classes: StraightSeeding and CircleSeeding. The PrelimEvent class will return an instance of CircleSeeding and the TimedFinalEvent class will return an instance of StraightSeeding. Thus we see that we have two hierarchies: one of Events and one of Seedings. We see these two hierarchies illustrated in Figure 2.

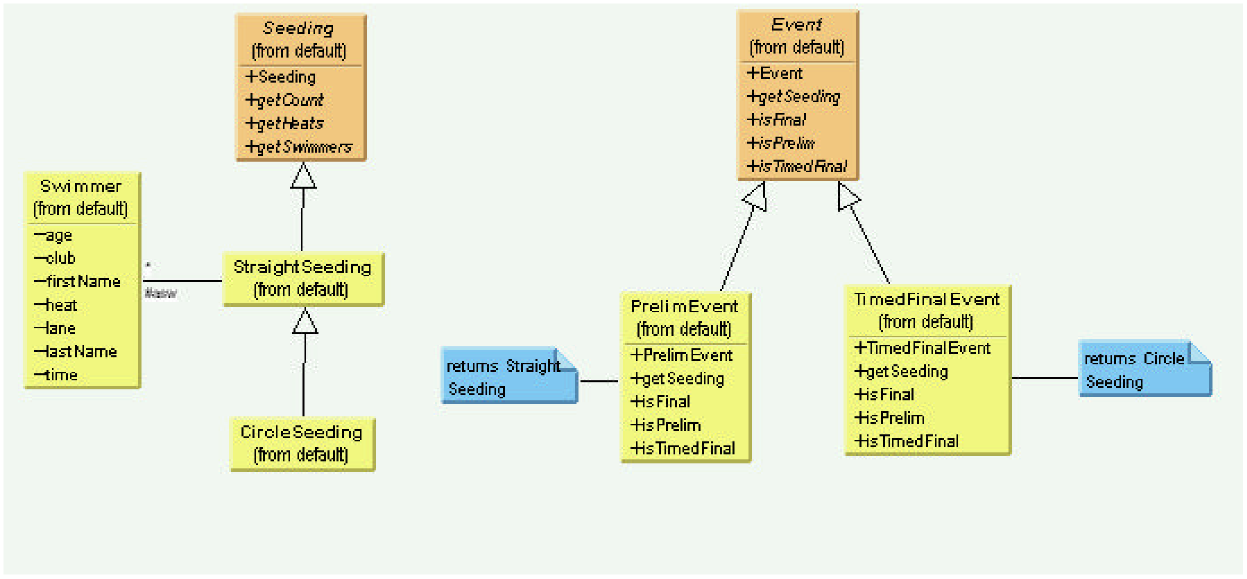


Figure 2 –The Seeding hierarchy and the Event hierarchy.

We can learn a lot from the simple UML diagram in Figure 2. First, we see that Event is an abstract class and has two concrete classes derived from it: PrelimEvent and TimedFinalEvent. Then we see that Seeding is also an abstract class and that it has StraightSeeding derived from it. Since circle seeding reuses the methods of straight seeding for heats after the first three, we derive CircleSeeding from StraightSeeding. They are still both instances of the base Seeding class.

In the Event hierarchy, you will see that both derived Event classes contain a **getSeeding** method. One of them returns an instance of StraightSeeding and the other an instance of CircleSeeding. So you see, there is no real factory decision point as we had in our simple example. Instead, the decision as to which Event class to instantiate is the one that determines which Seeding class will be instantiated.

While it looks like there is a one to one correspondence between the two class hierarchies, there needn't be. There could be many kinds of Events and only a few kinds of Seeding that they use.

The Swimmer class

We haven't said much about the Swimmer class, except that it contains a name, club age, seed time and place to put the heat and lane after seeding. The Event class reads in the Swimmers from some database (a file in our example) and then passes that Vector to the Seeding class when you call the getSeeding method for that event.

The Event Classes

We have seen the abstract base Event class above. In actual use, we use it to read in the swimmer data (here from a file) and pass it on to instances of the Swimmer class to parse

```
public abstract class Event
 {
   protected int numLanes;
   protected Vector swimmers;
   public Event(String filename, int lanes) {
      numLanes = lanes;
      swimmers = new Vector();

      InputFile f = new InputFile(filename);
      String s = f.readLine();
      while(s != null) {
         Swimmer sw = new Swimmer(s);
         swimmers.addElement(sw);
         s = f.readLine();
     }
      f.close();
   }
   public abstract Seeding getSeeding();
   public abstract boolean isPrelim();
   public abstract boolean isFinal();
   public abstract boolean isTimedFinal();
 }
```

Our PrelimEvent class just returns an instance of CircleSeeding:

```
public class PrelimEvent extends Event {
 //creates a preliminary event which is circle seeded
   public PrelimEvent(String filename, int lanes) {
      super(filename, lanes);
   }
   public  Seeding getSeeding() {
      return new CircleSeeding(swimmers, numLanes);
   }
}
```

while the TimedFinalEvent returns an instance of StraightSeeding:

```
public class TimedFinalEvent extends Event {
//creates an event that will be straight seeded
   public TimedFinalEvent(String filename, int lanes) {
      super(filename, lanes);
   }
   public  Seeding getSeeding() {
      return new StraightSeeding(swimmers, numLanes);
   }
 }
```

Straight Seeding

In actually writing this program, we'll discover that most of the work is done in straight seeding. The changes for circle seeding are pretty minimal. So we instantiate our StraightSeeding class and copy in the Vector of swimmers and the number of lanes

```
public StraightSeeding(Vector sw, int lanes) {
      Swimmers = sw;
      numLanes = lanes;
      count = sw.size();
      calcLaneOrder();
      seed();
   }
```

Then, as part of the constructor, we do the basic seeding.

```
   //-------------------------------
   protected void seed() {
      //loads the asw array and sorts it
      sortUpwards();
      //number in last heat
      int lastHeat = count % numLanes;
    if (lastHeat < 3)
       lastHeat = 3;   //last heat must have 3 or more
    int lastLanes = count - lastHeat;
    numHeats = count / numLanes;

    if (lastLanes > 0)
       numHeats++;         //compute total number of heats
    int heats = numHeats;

    //place heat and lane in each swimmer's object
    int j = 0;
    //load from fastest to slowest
    //so we start with last heat # and work downwards
    for(int i = 0; i < lastLanes; i++) {
       Swimmer sw = asw[i];     //get each swimmer
       sw.setLane(lanes[j++]);  //copy in lane
       sw.setHeat(heats);       //and heat
       if(j >= numLanes) {
          heats--;              //next heat
          j=0;
       }
    }
    //Add in last partial heat
    if(j < numLanes)
       heats--;
    j = 0;
    for(int i = lastLanes-1; i<count; i++) {
       Swimmer sw = asw[i];
       sw.setLane(lanes[j++]);
       sw.setHeat(heats);
    }
   //copy from array back into Vector
    Swimmers = new Vector();
    for(int i=0; i< count; i++)
       Swimmers.addElement(asw[i]);
    }
```

This makes the entire array of seeded Swimmers available when you call the getSwimmers method.

Circle Seeding

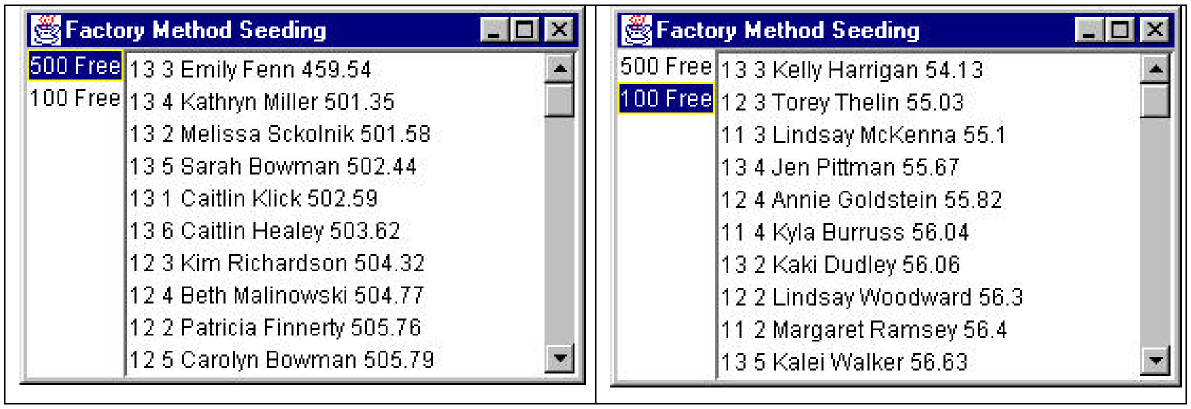
The CircleSeeding class is derived from StraightSeeding, so it copies in the same data.

```
public class CircleSeeding extends StraightSeeding {

  public CircleSeeding(Vector sw, int lanes) {
  super(sw, lanes);     //also straight seeds
  super.seed();
  seed();
}
//-----------------------------
protected void seed() {
  int circle;

 //circle seed top heats if there are 2 or more
   if (numHeats >=2 ) {
      if(numHeats>=3)
         circle = 3;
      else
         circle = 2;
      //this just replaces the top heat seeding
      //and leaves the rest untouched
      int i= 0;
      for (int j=0; j < numLanes; j++) {
         for(int k=0; k < circle; k++) {
            asw[i].setLane(lanes[j]);
            asw[i++].setHeat(numHeats - k);
         }
      }
   }
 }
}
```

Since the constructor calls the parent class constructor, it copies the swimmer vector and lanes values. The, our call to **super.seed()** *does the straight seeding*. This simplifies things, because we will always need to seed the remaining heats by straight seeding. Then we seed the last 2 or 3 heats as shown above and we are done with that type of seeding as well.

Our Seeding Program

No study of classes and data handling is complete unless we show a working example of the resulting final program at work. In this example, we took a list of swimmers in the 500 yd freestyle and the100 yd freestyle and used them to build our TimeFinalEvent and PrelimEvent classes. You can see the results of these two seedings in Figure 3.



```
500 Free   13 3 Emily Fenn 459.54
100 Free   13 4 Kathryn Miller 501.35
           13 2 Melissa Sckolnik 501.58
           13 5 Sarah Bowman 502.44
           13 1 Caitlin Klick 502.59
           13 6 Caitlin Healey 503.62
           12 3 Kim Richardson 504.32
           12 4 Beth Malinowski 504.77
           12 2 Patricia Finnerty 505.76
           12 5 Carolyn Bowman 505.79
```

```
500 Free   13 3 Kelly Harrigan 54.13
100 Free   12 3 Torey Thelin 55.03
           11 3 Lindsay McKenna 55.1
           13 4 Jen Pittman 55.67
           12 4 Annie Goldstein 55.82
           11 4 Kyla Burruss 56.04
           13 2 Kaki Dudley 56.06
           12 2 Lindsay Woodward 56.3
           11 2 Margaret Ramsey 56.4
           13 5 Kalei Walker 56.63
```

Figure 3- Seedings of a straight and a circle seeded event.

Other Factories

Now one issue that we have skipped over is how the program that reads in the swimmer data decides which kind of event to generate. We finesse this here by simply calling the two constructors directly:

```
events.addElement(new TimedFinalEvent("500free.txt", 6));
events.addElement(new PrelimEvent("100free.txt", 6));
```

Clearly, this is an instance where an EventFactory may be needed to decide which kind of event to generate. This revisits the simple factory we began the article with and have discussed previously.

References

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