Chapter 4 A Dynamic Vector Data Structure and Class Inheritance

The `Vector` container from Chapter 3 has the annoying limitation that an instance may become full. In this chapter we will code a new container that does not have this limitation. The new container will still store its elements in an array, but the array will be resized when necessary to accommodate the insertion of new elements. An array that may be resized at run time is called a *dynamic array*, and any data structure that exhibits such dynamic behavior is said to be a *dynamic data structure*. In later chapters we will examine data structures that are dynamic but do not store their elements in dynamic arrays. In other words, the term *dynamic data structure* does not imply the use of a dynamic array but instead refers to run-time resizing that can be implemented in two ways.

4.1 Our Current Model

The first evolution of our model appears in Figure 4.1. In this chapter we will rename Chapter 3's `Vector` class to `FixedVector`, and we will code a dynamic vector container, `DynamicVector`. We will then generalize `FixedVector` and `DynamicVector` by way of a new `Vector` class.

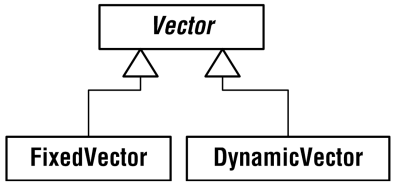

Fig. 4.1. Our current class diagram, which shows our first generalization

4.2 Class `DynamicVector`

The code for class `DynamicVector` appears below. Because `DynamicVector` differs from Chapter 3's class `Vector` only with respect to the `append` and `insertAt` methods, only these methods, an overloaded constructor, and `ensureCapacity` are shown.

```
public class DynamicVector
{
    private static final int DEFAULT_CAPACITY = 100;
    private Object[] data;
    private int numItems;

    public DynamicVector(int initCapacity)
    {
        if (initCapacity <= 0)
            data = new Object[DEFAULT_CAPACITY];
        else
            data = new Object[initCapacity];
    }

    /*
     precondition: N/A
    postcondition: If the array was full, its capacity has been
                   doubled. The new element has been placed at the
                   end of the collection and true has been
                   returned.
    */

    public boolean append(Object element)
    {
        /*
        If the array is full, allocate a new array with twice the
        capacity. Then copy the elements from the old array to the
        new array and overwrite the old array's reference with the
        new array's reference. Setting data = newData sets the old
        array's reference count to zero, which means the garbage
        collector will soon reclaim the old array.
        */

        if (isFull())
        {
            Object[] newData = new Object[data.length * 2];
            for (int j = 0; j < numItems; j++)
                newData[j] = data[j];
            data = newData;
        }
        data[numItems++] = element;
        return true;
    }
```

```
    /*
    This method should be used to avoid repeated resizings when
    many insertions are anticipated.

     precondition: If the array's capacity is already greater than
                   or equal to the desired capacity, then resizing
                   is unnecessary, in which case the method simply
                   returns.
    postcondition: The array's capacity is now equal to the
                   desired capacity.
    */

    public void ensureCapacity(int minCapacity)
    {
        if (minCapacity <= data.length)
            return;
        Object[] newData = new Object[minCapacity];
        for (int j = 0; j < numItems; j++)
            newData[j] = data[j];
        data = newData;
    }

    // If necessary, this method now resizes the array in the same
    // fashion as method append().

    public boolean insertAt(int index, Object element)
    {
        if (index < 0 || index > numItems - 1)
            return false;
        int i, j;
        if (isFull())
        {
            Object[] newData = new Object[data.length * 2];
            for (i = j = 0; j < numItems; i++, j++)
            {
                if (j == index)
                {
                    i++;
                    continue;
                }
                newData[i] = data[j];
            }
            data = newData;
            data[index] = element;
            numItems++;
            return true;
        }
        for (j = numItems - 1; j >= index; j--)
            data[j + 1] = data[j];
        data[index] = element;
        numItems++;
        return true;
    }
}
```

We will henceforth refer to the container from Chapter 3 as `FixedVector`, owing to its use of a *static array*, i.e., an array with a fixed capacity.

4.3 Class Inheritance

The duplication of code in our two vector data structures presents a bit of a maintenance problem. Suppose, for example, that method `removeAt` contains one or more logic errors. Both implementations must be purged of the same error(s). If we repair one implementation but forget to repair the other, then we have one working implementation and one broken one. If we remember to repair both implementations of `removeAt` but introduce a new logic error(s) during our debugging attempt, then we are again left with two broken methods. Our debugging would be less complicated if there were only one copy of `removeAt` common to both `FixedVector` and `DynamicVector`. A feature of object-oriented languages called *class inheritance* makes this possible.

class inheritance: the creation of a new class by the extension (*specialization*) of an existing class or by the generalization of two or more existing classes; in the context of a programming language, inheritance is analogous to the UML generalization relationship presented in Chapter 1; *generalization* is a UML term, while *inheritance* is an object-oriented programming (OOP) term; inheritance is used to implement generalization

Java also supports *interface inheritance*. This is not true of all object-oriented programming languages, for not all object-oriented languages support interfaces in the manner that Java does. At any rate, the UML term *generalization* covers both kinds of inheritance.

4.3.1 Using Inheritance: Best Practices

The following principles should generally be followed when using inheritance. They are presented as principles rather than as strict rules because you may at times feel compelled to violate one or both of them because in your opinion the situation calls for doing so. We will never violate either principle in this book.

1. **First Principle of Inheritance**: A subclass should always extend its superclass. A subclass should not use fewer fields than its superclass.
2. **Second Principle of Inheritance**: The generalization relationship implemented by an inheritance should make sense when stated in everyday language.

We now illustrate the Second Principle with a code example. Class `GoodCircle` follows the Second Principle while class `BadCircle` violates it.

```
/*
An instance of class Point2D represents a location in a
two-dimensional coordinate system.

                +y
                |
                |
       -x_______|_______+x
                |
                |
                |
                -y
*/

public class Point2D
{
    protected double x,  // The instance fields consist of
                     y;  // variables x and y.

    // The following overloaded constructor allows one to
    // initialize a Point2D instance to some location besides the
    // origin.

    public Point2D(double initX, double initY)
    {
        x = initX;
        y = initY;
    }

    // The following accessor methods allow one to query a Point2D
    // object for the location it represents.

    public final double getX()
    {
        return x;
    }

    public final double getY()
    {
        return y;
    }

    // The following mutator methods allow one to move a Point2D
    // instance to a new location in the x-y plane.

    public final void setX(double newX)
    {
        x = newX;
    }

    public final void setY(double newY)
    {
        y = newY;
    }
}
```

```
// A circle has-a point marking its center.

public class GoodCircle
{
    private Point2D center;  // Association is the appropriate
                             // relationship between Point2D and
                             // Circle.
    private double radius;

    public GoodCircle(double cx, double cy, double initRad)
    {
        center = new Point2D(cx, cy);
        if (initRad > 0.0)
            radius = initRad;
        else
            radius = 1.0;
    }

    public final double getX()
    {
        return center.getX();
    }

    public final double getY()
    {
        return center.getY();
    }

    public final void setX(double newX)
    {
        center.setX(newX);
    }

    public final void setY(double newY)
    {
        center.setY(newY);
    }

    public double getRadius()
    {
        return radius;
    }

    public boolean setRadius(double newRad)
    {
        if (newRad > 0.0)
        {
            radius = newRad;
            return true;
        }
        return false;
    }

    public double getCircumference()
    {
        return 2.0 * Math.PI * radius;
    }

    public double getArea()
    {
        return Math.PI * radius * radius;
    }
}
```

```
// BadCircle does not use inheritance appropriately. A circle is
// not a two-dimensional point!

public class BadCircle extends Point2D
{
    private double radius;

    public BadCircle(double cx, double cy, double initRad)
    {
        super(cx, cy);         // Call the superclass constructor.
        if (initRad > 0.0)
            radius = initRad;
        else
            radius = 1.0;
    }

    public double getRadius()
    {
        return radius;
    }

    public boolean setRadius(double newRad)
    {
        if (newRad > 0.0)
        {
            radius = newRad;
            return true;
        }
        return false;
    }

    public double getCircumference()
    {
        return 2.0 * Math.PI * radius;
    }

    public double getArea()
    {
        return Math.PI * radius * radius;
    }
}
```

4.3.2 Inheritance Applied to Our Vector Containers

We now revise our vector containers using inheritance. The fields and methods common to `FixedVector` and `DynamicVector` are extracted and placed in a superclass called `Vector`. Classes `FixedVector` and `DynamicVector` extend `Vector` just as `HandWeapon` and `ProjectileWeapon` extended `Weapon`. Class `Vector` is, like `Weapon`, an *abstract class*.

abstract class: a class whose sole purpose is to offer certain fields and/or methods to any class that subclasses it; an abstract class cannot be instantiated

The code for classes `Vector`, `FixedVector`, and `DynamicVector` appears below.

```
public abstract class Vector
{
    // The fields are now protected so that they will be visible
    // to any subclass. Private fields and methods are inherited
    // by a subclass but are invisible to the subclass.

    protected static final int DEFAULT_CAPACITY = 100;
    protected Object[] data;
    protected int numItems;

    public Vector()
    {
        data = new Object[DEFAULT_CAPACITY];
    }

    /*
    Because this method's implementation varies between subclasses
    FixedVector and DynamicVector, it is declared abstract. An
    abstract method must be implemented by any concrete, i.e.,
    instantiable, subclass. An abstract method need not be
    implemented by an abstract subclass, however.
    */

    public abstract boolean append(Object element);

    public void clear()
    {
        for (int j = 0; j < numItems; j++)
            data[j] = null;
        numItems = 0;
    }

    public boolean contains(Object element)
    {
        return indexOf(element) != -1;
    }

    public Object elementAt(int index)
    {
        if (index < 0 || index > numItems - 1)
            return null;
        return data[index];
    }

    public int indexOf(Object element)
    {
        for (int j = 0; j < numItems; j++)
            if (element.equals(data[j]))
                return j;
        return -1;
    }

    public abstract boolean insertAt(int index, Object element);

    public boolean isEmpty()
    {
        return numItems == 0;
    }
```

```
    // Method isFull() is now protected. Why?

    protected boolean isFull()
    {
        return numItems == data.length;
    }

    public Object removeAt(int index)
    {
        if (index < 0 || index > numItems - 1)
            return null;
        Object temp = data[index];
        while (index < numItems - 1)
        {
            data[index] = data[index + 1];
            index++;
        }
        data[--numItems] = null;
        return temp;
    }

    public boolean remove(Object element)
    {
        int pos = indexOf(element);
        if (pos == -1)
            return false;
        removeAt(pos);
        return true;
    }

    public boolean replace(int index, Object element)
    {
        if (index < 0 || index > numItems - 1)
            return false;
        data[index] = element;
        return true;
    }

    public int size()
    {
        return numItems;
    }
}

public class FixedVector extends Vector
{
    public FixedVector() {}

    public boolean append(Object element)
    {
        if (isFull())
            return false;
        data[numItems++] = element;
        return true;
    }

    public boolean insertAt(int index, Object element)
    {
        if (index < 0 || index > numItems - 1 || isFull())
            return false;
        for (int j = numItems - 1; j >= index; j--)
            data[j + 1] = data[j];
        data[index] = element;
        numItems++;
        return true;
    }
```

```
    public boolean isFull()
    {
        return super.isFull();
    }
}

public class DynamicVector extends Vector
{
    public DynamicVector() {}

    public DynamicVector(int initCapacity)
    {
        if (initCapacity <= 0)
            data = new Object[DEFAULT_CAPACITY];
        else
            data = new Object[initCapacity];
    }

    public boolean append(Object element)
    {
        if (isFull())
        {
            Object[] newData = new Object[data.length * 2];
            for (int j = 0; j < numItems; j++)
                newData[j] = data[j];
            data = newData;
        }
        data[numItems++] = element;
        return true;
    }

    public void ensureCapacity(int minCapacity)
    {
        if (minCapacity <= data.length)
            return;
        Object[] newData = new Object[minCapacity];
        for (int j = 0; j < numItems; j++)
            newData[j] = data[j];
        data = newData;
    }
```

```
    public boolean insertAt(int index, Object element)
    {
        if (index < 0 || index > numItems - 1)
            return false;
        int i, j;
        if (isFull())
        {
            Object[] newData = new Object[data.length * 2];
            for (i = j = 0; j < numItems; i++, j++)
            {
                if (j == index)
                {
                    i++;
                    continue;
                }
                newData[i] = data[j];
            }
            data = newData;
            data[index] = element;
            numItems++;
            return true;
        }
        for (j = numItems - 1; j >= index; j--)
            data[j + 1] = data[j];
        data[index] = element;
        numItems++;
        return true;
    }
}
```

Now our vector containers are not only easier to maintain, but classes `FixedVector` and `DynamicVector` are more or less interchangeable, as we will see in the next chapter. A certain degree of type interchangeability is another benefit of inheritance, as we saw in regard to `FixedVector`'s type independence in Chapter 3. In the next chapter, the inheritance used here will allow the methods presented there to be type independent in a narrower sense.

Exercises

1. Add attributes and operations to the class diagram given at the beginning of the chapter.
2. Create a UML class diagram showing classes `Point2D`, `GoodCircle`, and `BadCircle`. Show all relationships, attributes, and operations.
3. Using class `Point2D` as a guide, design and code classes `Point1D`, `Point3D`, and `Point4D`. Also design and code a class called `Sphere`, an instance of which can be "positioned" at any point in three-dimensional space. Create a UML class diagram before coding.
4. Design and code a class called `EuclidianSpace` that offers class methods for adding or subtracting two real-valued `DynamicVector`s, multiplying a real-valued `DynamicVector` by a scalar, finding the norm of a real-valued `DynamicVector`, finding the unit vector in the direction of a real-valued `DynamicVector`, and computing the dot product of two real-valued `DynamicVectors`. See the MathWorld website or consult a mathematics text for more information on these vector operations.
5. Design and code a class called `Set` that implements the set ADT. Use class `DynamicVector` in your implementation. See the MathWorld website or consult a mathematics text for more information about sets.

```
set: an unordered collection of distinct elements

operations:

            cardinality() - How many elements are in the
                            collection?
                  clear() - Make the collection empty.
             complement(B) - Find the set this - B.
         contains(element) - Does the collection contain the given
                            element?
                equals(B) – Is this set equal to B?
           insert(element) - Add a new element to the collection.
           intersection(B) - Find the set this ∩ B.
                 isEmpty() - Is the collection empty?
           remove(element) - Remove the given element from the
                            collection.
               subsetOf(B) - Is this set a subset of B?
   symmetricDifference(B) - Find the set (this - B) ∪ (B - this).
                union(B) - Find the set this ∪ B.
```