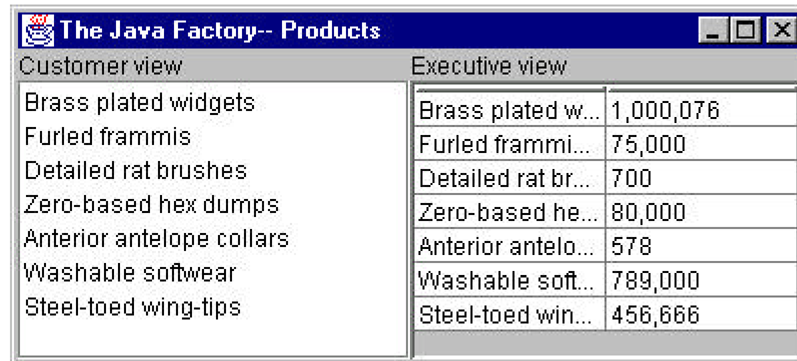
THE BRIDGE PATTERN

The Bridge pattern is used to separate the interface of class from its implementation, so that either can be varied separately. At first sight, the bridge pattern looks much like the Adapter pattern, in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation, so that you can vary or replace the implementation without changing the client code.

Suppose that we have a program that displays a list of products in a window. The simplest interface for that display is a simple JList box. But, once a significant number of products have been sold, we may want to display the products in a table along with their sales figures.

Since we have just discussed the adapter pattern, you might think immediately of the class-based adapter, where we adapt the fairly elaborate interface of the JList to our simpler needs in this display. In simple programs, this will work fine, but as we'll see below there are limits to that approach.

Let's further suppose that we need to produce two kinds of displays from our product data, a customer view that is just the list of products we've mentioned, and an executive view which also shows the number of units shipped. Well display the product list in an ordinary JList box and the executive view in a JTable table display. To simplify peripheral programming issues, we'll just show both displays as two lists in a single window, as we see below:



At the top programming level, we just create instances of a table and a list from classes derived from JList and Jtable but designed to parse apart the names and the quantities of data.

```
pleft.setLayout(new BorderLayout());
pright.setLayout(new BorderLayout());

//add in customer view as list box
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center", new productList(prod));

//add in execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center", new productTable(prod));
```

We derive the *productList* class directly from the JawtList class we just wrote, so that the Vector containing the list of products is the only input to the class.

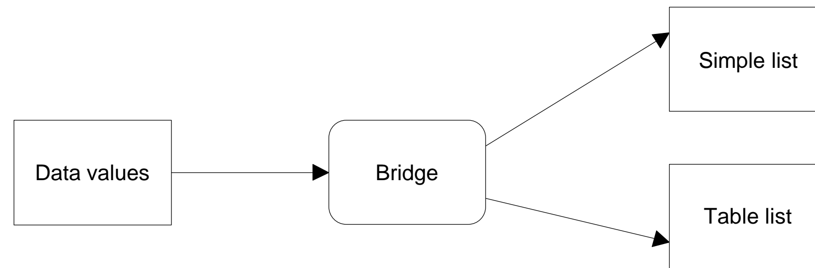
```
public class productList extends JawtList
{
   public productList(Vector products)
   {
      super(products.size());    //for compatibility
      for (int i = 0; i < products.size(); i++)
      {
         //take each string apart and keep only
         //the product names, discarding the quantities
         String s = (String)products.elementAt(i);

          //separate qty from name
          int index = s.indexOf("--");
          if(index > 0)
            add(s.substring(0, index));
         else
            add(s);
      }
   }
}
```

Building a Bridge

Now suppose that we need to make some changes in the way these lists display the data. For example, you might want to have the products displayed in alphabetical order. In order to continue with this approach, you'd need to either modify or subclass *both* of these list classes. This can quickly get to be a maintenance nightmare, especially if more than two such displays eventually are needed. So rather than deriving new classes whenever we need

to change these displays further, let's build a single *bridge* that does this work for us:



We want the bridge class to return an appropriate visual component so we'll make it a kind of scroll pane class:

```
public class listBridge extends JscrollPane
```

When we design a bridge class, we have to decide how the bridge will determine which of the several classes it is to instantiate. It could decide based on the values or quantity of data to be displayed, or it could decide based on some simple constants. Here we define the two constants inside the listBridge class:

```
static public final int TABLE = 1, LIST = 2;
```

We'll keep the main program constructor much the same, replacing specialized classes with two calls to the constructor of our new listBridge class:

```
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center",
             new listBridge(prod, listBridge.LIST));

 //add in execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center",
             new listBridge(prod, listBridge.TABLE));
```

Our constructor for the listBridge class is then simply

```
public listBridge(Vector v, int table_type)
   {
      Vector sort = sortVector(v);      //sort the vector
```

```
      if (table_type == LIST)
         getViewport().add(makeList(sort));  //make table

      if (table_type == TABLE)
         getViewport().add(makeTable(sort)); //make list

   }
```

The important difference in our bridge class is that we can use the JTable and JList class directly without modification and thus can put any adapting interface computations in the data models that construct the data for the list and table.

```
private JList makeList(Vector v)    {
     return new JList(new BridgeListData(v));
  }
//---------------------------------
private JTable makeTable(Vector v)    {
      return new JTable(new prodModel(v));
  }
```

The resulting sorted display is shown below:

The Java Factory-- Products

Customer view	Executive view	
Anterior antelope collars	Anterior antelo...	578
Brass plated widgets	Brass plated w...	1,000,076
Detailed rat brushes	Detailed rat br...	700
Furled frammis	Furled frammi...	75,000
Steel-toed wing-tips	Steel-toed win...	456,666
Washable softwear	Washable soft...	789,000
Zero-based hex dumps	Zero-based he...	80,000

Consequences of the Bridge Pattern

1. The Bridge pattern is intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. This can prevent you from recompiling a complicated set of user interface modules, and only require that you recompile the bridge itself and the actual end display class.

2. You can extend the implementation class and the bridge class separately, and usually without much interaction with each other.

3. You can hide implementation details from the client program much more easily.