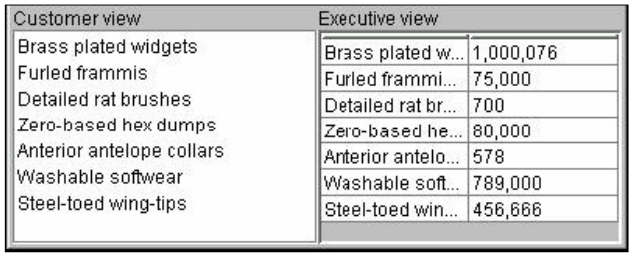
An Example Application

- We want a program that displays a list of products in a window.
  - The simplest interface for that display is a simple JList box.
- Once a significant number of products have been sold, we want to display the products in a table along with their sales figures.
- 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.
- We'll display the product list in an ordinary JList box and the executive view in a JTable table display.

The UI

- For simplicity, lets just show both displays in a single window:



The Adapter Pattern?

- Can someone implement it?

Top Programming Level

- 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));
```

productList

- We derive the productList class directly from the JawtList class we wrote in the Adapter pattern slides, 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);
      }
   }
}
```


productTable

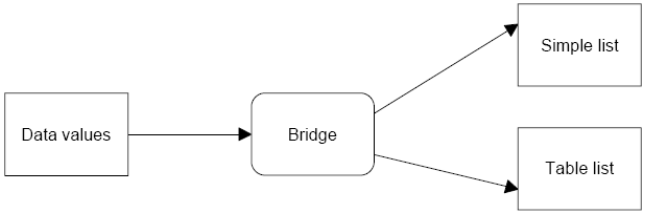
- Something similar?

Maintenance Task

- 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.

The Bridge Pattern

- 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



What kind of Bridge?

- 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;

At the Top Programming Level

- 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));
```


listBridge()

- 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

}
```


- We can use the JTable and JList class directly in our bridge class 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));
   }
```

CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Bridge Pattern

Mar. 29, 2007

What is it?

- 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.
- 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.