Structural Patterns: Decorator [4/7]

- **Name** - Decorator
- **A.K.A** - Wrapper
- **Intent** - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Problem** -
    - You want to add responsibilities to individual objects dynamically and transparently without affecting other objects
    - You want to create responsibilities that can be withdrawn
    - When you want to alter behavior of a set of classes and extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.
- **Solution** -
- **Participants and Collaborators**
    - Component
    - ConcrecteComponent
    - Decorator
    - ConcreteDecorator

- **Consequences**
    - More flexibility than static inheritance
    - Avoids feature-laden classes high up in the hierarchy
    - A decorator and its component aren't identical
    - Lots of little objects - can be difficult to debug
- **Implementation**
    - Interface conformance - the decorators need a common interface
    - Omitting the abstract Decorator class - if adding only a few Decorators
    - Keeping Component classes lightweight - the subclasses should contain the guts, not the superclass
    - Changing the skin of an object versus changing its guts - this is a Decorator vs. Strategy issue.
- GoF Reference (175)

The `java.io` package provides, among other things, a set of classes for reading input streams. Known as readers, each of those classes has a name that follows the pattern: `xxxReader`.

Readers provide specialized functionality; for example, one reader reads from a file, another tracks line numbers, and yet another pushes characters back on the input stream. In all, eight different readers exist to read input streams.

It's often necessary to combine the capabilities offered by `java.io` readers; for example, you might want to read from a file, keep track of line numbers, and push

certain characters back on the input stream, all at the same time. The `java.io` package's designers could have used inheritance to provide a wide array of such reader combinations; for example, a `FileReader` class could have a `LineNumberFileReader` subclass, which could in turn have a `PushBackLineNumberFileReader` subclass. But using inheritance to compose the most widely used combinations of reader functionality would result in a veritable explosion of classes. So how did the `java.io` package's designers create a design that allows you to combine reader functionality in any way you desire with only 10 reader classes? As you might guess, they used the Decorator pattern.

Instead of using inheritance to add functionality to *classes* at *compile time*, the Decorator pattern lets you add functionality to individual *objects* at *runtime*. That is accomplished by enclosing an object in another object. The enclosing object forwards method calls to the enclosed object and typically adds some functionality of its own before or after forwarding. The enclosing object -- known as a decorator -- conforms to the interface of the object it encloses, allowing the decorator to be used as though it were an instance of the object it encloses. That may sound complicated, but using decorators is actually quite simple. For example, the code listed in Example 3 uses a decorator to read and print the contents of a file. The code also prints line numbers and transforms "Tab" characters to three spaces.

Example 3. Use the Decorator pattern

```
import java.io.FileReader;
import java.io.LineNumberReader;

public class Test {
   public static void main(String args[]) {
      if(args.length < 1) {
         System.err.println("Usage: " + "java Test filename");
         System.exit(1);
      }
      new Test(args[0]);
   }
   public Test(String filename) {
      try {
         FileReader       frdr = new FileReader(filename);
         LineNumberReader lrdr = new LineNumberReader(frdr);

         for(String line; (line = lrdr.readLine()) != null;) {
            System.out.print(lrdr.getLineNumber() + ":\t");
            printLine(line);
         }
      }
      catch(java.io.FileNotFoundException fnfx) {
         fnfx.printStackTrace();
      }
      catch(java.io.IOException iox) {
         iox.printStackTrace();
```

```
      }
   }
   private void printLine(String s) {
      for(int c, i=0; i < s.length(); ++i) {
         c = s.charAt(i);

         if(c == '\t') System.out.print("   ");
         else          System.out.print((char)c);
      }
      System.out.println();
   }
}
```

If you use the code listed in Example 3 to read itself, you will get output that looks like Example 4.

Example 4. Output from using the application in Example 3 to print itself

```
1:    import java.io.FileReader;
2:    import java.io.LineNumberReader;
3:
4:    public class Test {
5:       public static void main(String args[]) {
6:          if(args.length < 1) {
7:             System.err.println("Usage: " + "java Test
filename");
8:             System.exit(1);
9:          }
10:         new Test(args[0]);
11:      }
12:      public Test(String filename) {
13:         try {
14:            FileReader       frdr = new FileReader(filename);
15:            LineNumberReader lrdr = new LineNumberReader(frdr);
16:
17:            for(String line; (line = lrdr.readLine()) != null;)
{
18:               System.out.print(lrdr.getLineNumber() + ":\t");
19:               printLine(line);
20:            }
21:         }
22:         catch(java.io.FileNotFoundException fnfx) {
23:            fnfx.printStackTrace();
24:         }
25:         catch(java.io.IOException iox) {
26:            iox.printStackTrace();
27:         }
28:      }
29:      private void printLine(String s) {
30:         for(int c, i=0; i < s.length(); ++i) {
```

```
31:             c = s.charAt(i);
32:
33:             if(c == '\t') System.out.print("   ");
34:             else          System.out.print((char)c);
35:          }
36:          System.out.println();
37:       }
38:    }
```

Notice how I've constructed the line reader used in Example 3:

```
FileReader       frdr = new FileReader(filename);
LineNumberReader lrdr = new LineNumberReader(frdr);
```

The `LineNumberReader` decorator encloses another reader; in this case, the enclosed reader is an instance of `FileReader`. The line number reader forwards method calls, such as `read()`, to its enclosed reader and tracks line numbers, which can be accessed with `LineNumberReader.getLineNumber()`. Because `LineNumberReader` is a decorator, you can easily track line numbers for any type of reader.

Decorator statics

Decorators decorate an object by enhancing (or in some cases restricting) its functionality. Those objects are referred to as *decorated.* Figure 1 shows the static relationship between decorators and the decorated.

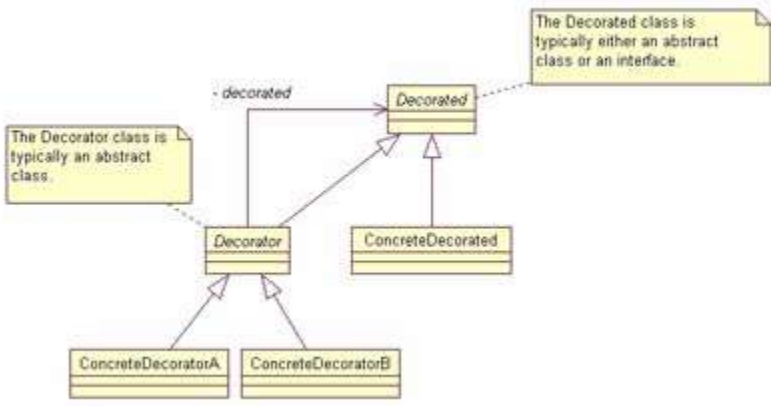

Figure 1. Decorator class diagram. Click on thumbnail to view full-size image.

Decorators extend the decorated class (or implement the decorated interface), which lets decorators masquerade as the objects they decorate. They also maintain a

reference to a `Decorated` instance. That instance is the object that the decorator decorates. As an example of how the classes in the Decorator pattern relate, Figure 2 depicts the static relationships among four decorators from the `java.io` package:

- `BufferedReader`
- `LineNumberReader`
- `FilterReader`
- `PushbackReader`

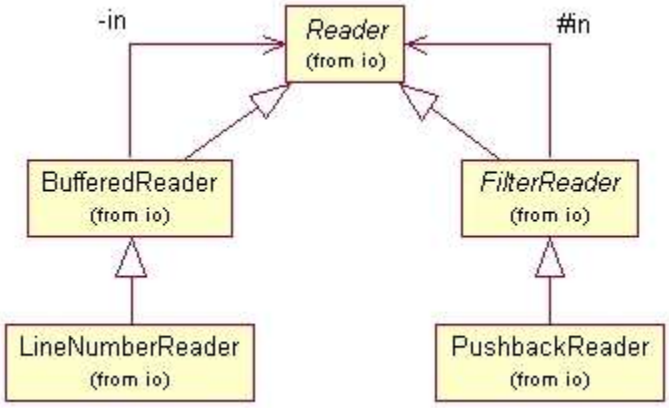

Figure 2. I/O decorators

`BufferedReader` and `FilterReader` are decorators just like the one shown in Figure 1. Both classes extend the abstract `Reader` class, and both forward method calls to an enclosed `Reader`. Because they extend `BufferedReader` and `FilterReader`, respectively, `LineNumberReader` and `PushbackReader` are also decorators.

Decorator dynamics

At runtime, decorators forward method calls to the objects they decorate, as shown in Figure 3.

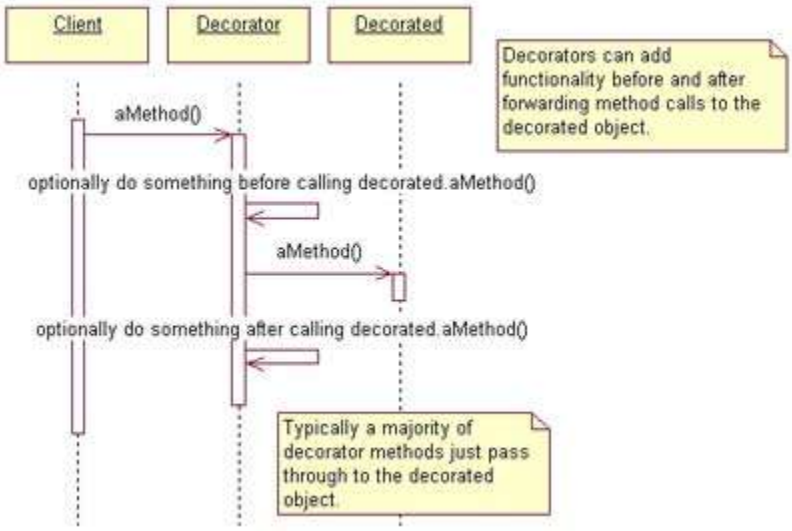

Figure 3. Decorator dynamics. Click on thumbnail to view full-size image.

Developers often refer to Decorators as *wrappers* because they wrap method calls to decorated objects. Figure 3 clearly depicts such wrapping. Figure 4 shows the dynamics of the code listed in Example 2.

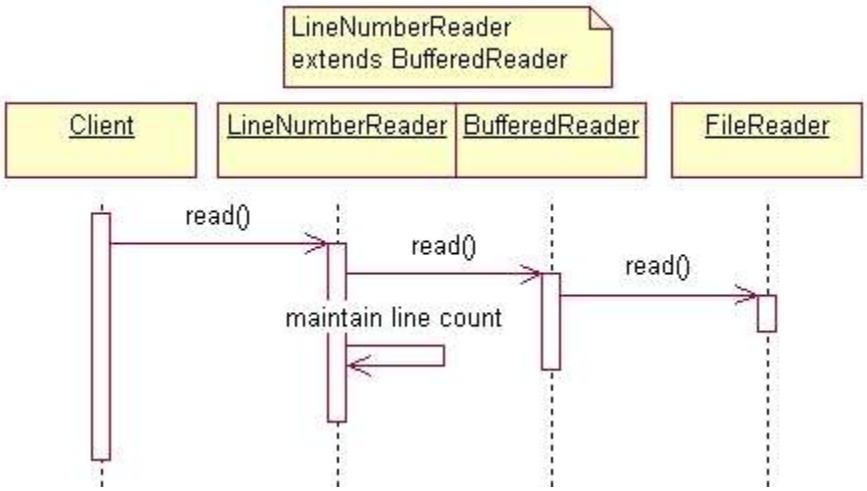

Figure 4. I/O decorator dynamics

Now that we have a high-level understanding of Decorator's statics and dynamics, let's use an example to examine the implementation of the Decorator pattern.

Sort and filter decorators for Swing tables

The Decorator pattern excels at attaching functionality, such as sorting and filtering, to Swing tables. In fact, the Decorator pattern is flexible enough that we can attach functionality to any Swing table at runtime. Before we can discuss how decorators enhance Swing tables in detail, we must have a basic understanding of Swing tables; so, let's take a short detour.

Swing tables

Swing components are implemented with the Model-View-Controller (MVC) design pattern. Each component consists of a model that maintains data, views that display the data, and controllers that react to events. For example, Swing tables, instances of `JTable`, are commonly created with an explicit model. Example 3 lists an application that does just that. (Try to picture the table in Example 3 before viewing it in Figure 5.)

Example 3. Create a Swing table

```
import javax.swing.*;
import javax.swing.table.*;

public class Test extends JFrame {
   public static void main(String args[]) {
      Test frame = new Test();
      frame.setTitle("Tables and Models");
      frame.setBounds(300, 300, 450, 300);
      frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      frame.show();
   }
   public Test() {
      TableModel model = new TestModel();
      getContentPane().add(new JScrollPane(new JTable(model)));
   }
   private static class TestModel extends AbstractTableModel {
      final int rows = 100, cols = 10;
      public int    getRowCount()    { return rows; }
      public int    getColumnCount() { return cols; }
      public Object getValueAt(int row, int col) {
         return "(" + row + "," + col + ")";
      }
   }
}
```

The application creates a table with a model. That table is placed in a scrollpane and added to the frame's content pane. Figure 5 shows the application listed in Example 3.

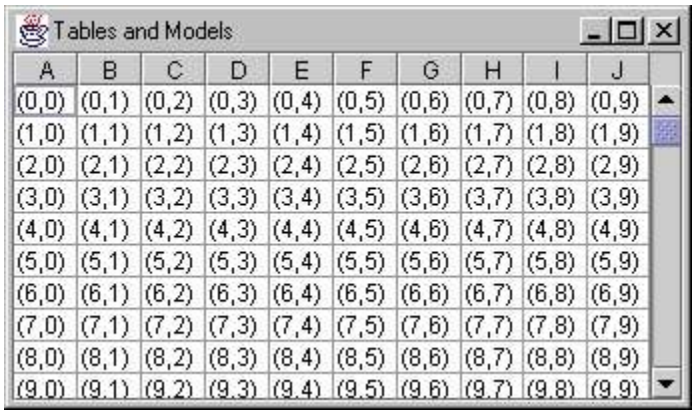

Figure 5. A simple Swing table

The application shown above creates a table model with 100 rows and 10 columns. When asked for a cell value, the model creates a string that represents the cell's row and column. The application uses that model to construct a Swing table (an instance of `JTable`). Table models produce a table's data, in addition to metadata such as the number of rows and columns.

It's important to understand that Swing tables possess models that maintain a table's data. Table models, as evidenced in Example 3, do not have to actually store any data -- they must produce a value for a given row and column.

A sort decorator

With a basic understanding of the Decorator pattern and Swing tables, we can now implement a table sort decorator. First, we'll see an application that uses the decorator, followed by a discussion of the decorator's implementation.

The application shown in Figure 6 contains a table with two columns: one for items and another for price. You can sort columns by clicking on column headers. When the application starts, nothing is sorted, as evidenced by the status bar in the bottom of the left picture in Figure 6. I took the middle picture in Figure 6 after I clicked the Item column header. I took the the right picture after I clicked the Price/Lb. column header.

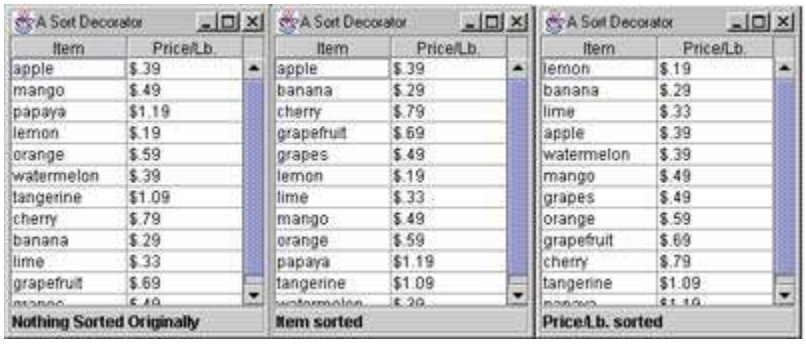

Figure 6. A sorting decorator. Click on thumbnail to view full-size image.

The application shown in Figure 6 replaces the table's model with a sort decorator. That application is partially listed in Example 4. (You can download the full source code from the Resources section below.)

Example 4. Sorting a Swing table

```
import java.awt.*;
import java.awt.event.*;

import java.util.Locale;
import java.util.ResourceBundle;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class Test extends JFrame {
   public static void main(String args[]) {
      SwingApp.launch(new Test(), "A Sort Decorator",
                                  300, 300, 200, 250);
   }
   public Test() {
      // Create the decorator that will decorate the table's
      // original model. The reference must be final because it's
      // accessed by an inner class below.
      final TableSortDecorator decorator =
new TableSortDecorator(table.getModel());

      // Set the table's model to the decorator. Because the
      // decorator implements TableModel, the table will never
      // know the difference between the decorator and it's
```

```
        // original model.
        table.setModel(decorator);
        ...

        // Obtain a reference to the table's header
        JTableHeader header = (JTableHeader)table.getTableHeader();

        // React to mouse clicks in the table header by calling
        // the decorator's sort method.
        header.addMouseListener(new MouseAdapter() {
           public void mouseClicked(MouseEvent e) {
              TableColumnModel tcm = table.getColumnModel();
              int vc = tcm.getColumnIndexAtX(e.getX());
              int mc = table.convertColumnIndexToModel(vc);

              // Perform the sort
              decorator.sort(mc);

              // Update the status area
              SwingApp.showStatus(headers[mc] + " sorted");
           }
        });
     }
     ...
}
```

In the preceding code, a sort decorator decorates the table's original model. When you click on a column header, the application calls the decorator's `sort()` method. The interesting code in this example is not the application, but the sort decorator. Before looking at the source to `TableSortDecorator`, let's look at its class diagram, shown in Figure 7.

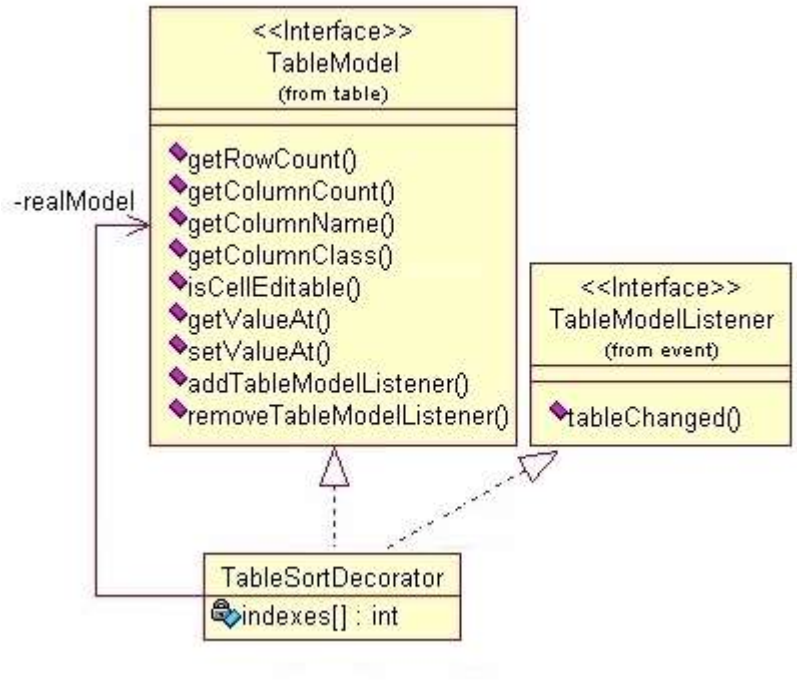

Figure 7. Sorting decorator class diagram

Since instances of `TableSortDecorator` decorate table models, `TableSortDecorator` implements the `TableModel` interface. `TableSortDecorator` also maintains a reference to the real model that it decorates. That reference can point to any table model. Example 5 lists the `TableSortDecorator` class:

Example 5. A table sort decorator

```
class TableSortDecorator implements TableModel,TableModelListener{
   public TableSortDecorator(TableModel model) {
      this.realModel = model;
      realModel.addTableModelListener(this);
      allocate();
   }

   // The following nine methods are defined by the TableModel
   // interface; all of those methods forward to the real model.
   public void addTableModelListener(TableModelListener l) {
      realModel.addTableModelListener(l);
   }
```

```
public Class getColumnClass(int columnIndex) {
   return realModel.getColumnClass(columnIndex);
}
public int getColumnCount() {
   return realModel.getColumnCount();
}
public String getColumnName(int columnIndex) {
   return realModel.getColumnName(columnIndex);
}
public int getRowCount() {
   return realModel.getRowCount();
}
public boolean isCellEditable(int rowIndex, int columnIndex) {
   return realModel.isCellEditable(rowIndex, columnIndex);
}
public void removeTableModelListener(TableModelListener l) {
   realModel.removeTableModelListener(l);
}
public Object getValueAt(int row, int column) {
   return getRealModel().getValueAt(indexes[row], column);
}
public void setValueAt(Object aValue, int row, int column) {
   getRealModel().setValueAt(aValue, indexes[row], column);
}

// The getRealModel method is used by subclasses to
// access the real model.
protected TableModel getRealModel() {
   return realModel;
}

// tableChanged is defined in TableModelListener, which
// is implemented by TableSortDecorator.
public void tableChanged(TableModelEvent e) {
   allocate();
}

// The following methods perform the bubble sort ...

public void sort(int column) {
   int rowCount = getRowCount();

   for(int i=0; i < rowCount; i++) {
      for(int j = i+1; j < rowCount; j++) {
         if(compare(indexes[i], indexes[j], column) < 0) {
            swap(i,j);
         }
      }
   }
}
private void swap(int i, int j) {
```

```
      int tmp = indexes[i];
      indexes[i] = indexes[j];
      indexes[j] = tmp;
   }
   private int compare(int i, int j, int column) {
      TableModel realModel = getRealModel();
      Object io = realModel.getValueAt(i,column);
      Object jo = realModel.getValueAt(j,column);

      int c = jo.toString().compareTo(io.toString());
      return (c < 0) ? -1 : ((c > 0) ? 1 : 0);
   }
   private void allocate() {
      indexes = new int[getRowCount()];
      for(int i=0; i < indexes.length; ++i) {
         indexes[i] = i;
      }
   }
   private TableModel realModel; // We're decorating this model
   private int indexes[];
}
```

Table sort decorators do not change their real model when they sort; instead, decorators maintain an array of integers representing sorted row positions. When a sort decorator, masquerading as a table model, is asked for a value at a specific row and column, it uses the row value as an index into its array, and returns the corresponding value from the array. In that way, sort decorators overlay sorting onto Swing table models *without modifying the original model in any way.*

`TableSortDecorator` also implements the `TableModelListener` interface and registers itself as a listener with the real model. When the real model fires a table changed event, the decorator reallocates its array of sorted index references. That allocation happens in `TableSortDecorator.tableChanged`

`TableSortDecorator` has 11 public methods. Nine of those methods forward to the real model. Of those nine methods, only two methods -- `getValueAt()` and `setValueAt()` -- provide any additional functionality. Such ratios are not unusual among decorators; most decorators add a small amount of behavior to a decorated object that already performs several functions.

Refactoring the sort decorator

The sort decorator discussed above works as advertised by decorating arbitrary table models with sorting functionality at runtime. This gives you flexibility because the decorated models do not need to be modified, which means that any table model can be decorated with sorting capabilities.

But the `TableSortDecorator` as implemented above is not designed for reuse because it implements two responsibilities for which it is not well suited. The first responsibility is forwarding to the real model. Because other table model decorators will need almost exactly the same code, that responsibility is too general, and should be moved up the class hierarchy. The second responsibility is sorting, which in this example is a bubble sort. That sorting algorithm is hardcoded into the sort decorator, which means sorting cannot be changed without rewriting the entire decorator. The sorting algorithm is too specific, so it should be moved down the hierarchy.

Figure 8 shows the result of refactoring the sort decorator as recommended in the preceding paragraph.

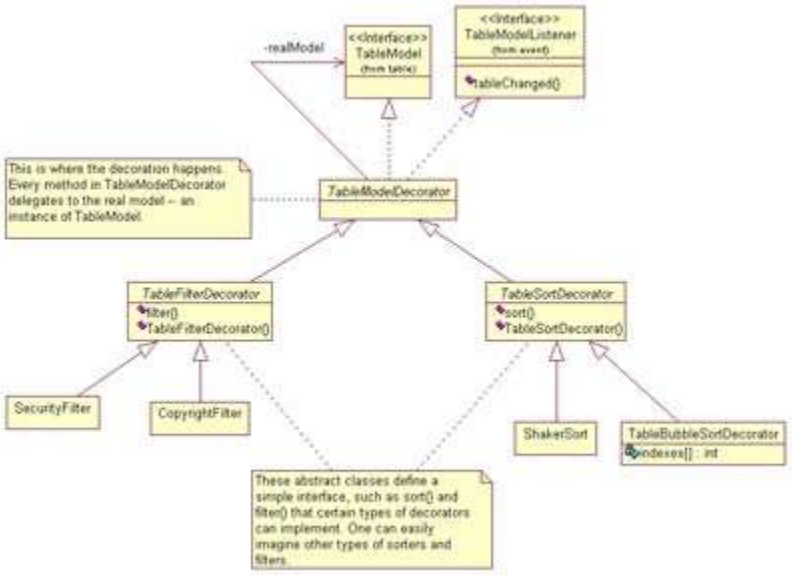

Figure 8. Refactored sorting decorator class diagram. Click on thumbnail to view full-size image.

The refactoring split the original `TableSortDecorator` into three different classes:

- **`TableModelDecorator`:** Implements `TableModel` by forwarding to the real model

- **`TableSortDecorator`:** Extends `TableModelDecorator` and adds an abstract `sort` method that subclasses must implement

- **TableBubbleSortDecorator:** Extends `TableSortDecorator` and implements the bubble sort

By refactoring the sort decorator, we can reuse the code that forwards to the real table model. Encapsulating that code in `TableModelDecorator` lets us easily develop other table model decorator types, such as the filter decorators -- which are not discussed in this article -- shown in Figure 8.

The abstract `TableSortDecorator` class defers the implementation of its `sort()` method to subclasses, making it trivial to implement sort decorators with a different sorting algorithm.

Example 6 lists the `TableModelDecorator` class.

Example 6. TableModelDecorator

```
import javax.swing.table.TableModel;
import javax.swing.event.TableModelListener;

// TableModelDecorator implements TableModelListener. That
// listener interface defines one method: tableChanged(), which
// is called when the table model is changed. That method is
// not implemented in this abstract class; it's left for
// subclasses to implement.
public abstract class TableModelDecorator
                        implements TableModel, TableModelListener
{
   public TableModelDecorator(TableModel model) {
      this.realModel = model;
      realModel.addTableModelListener(this);
   }

   // The following 9 methods are defined by the TableModel
   // interface; all of those methods forward to the real model.
   public void addTableModelListener(TableModelListener l) {
      realModel.addTableModelListener(l);
   }
   public Class getColumnClass(int columnIndex) {
      return realModel.getColumnClass(columnIndex);
   }
   public int getColumnCount() {
      return realModel.getColumnCount();
   }
   public String getColumnName(int columnIndex) {
      return realModel.getColumnName(columnIndex);
   }
```

```
   public int getRowCount() {
      return realModel.getRowCount();
   }
   public Object getValueAt(int rowIndex, int columnIndex) {
      return realModel.getValueAt(rowIndex, columnIndex);
   }
   public boolean isCellEditable(int rowIndex, int columnIndex) {
      return realModel.isCellEditable(rowIndex, columnIndex);
   }
   public void removeTableModelListener(TableModelListener l) {
      realModel.removeTableModelListener(l);
   }
   public void setValueAt(Object aValue,
                          int rowIndex, int columnIndex) {
      realModel.setValueAt(aValue, rowIndex, columnIndex);
   }

   // The getRealModel method is used by subclasses to
   // access the real model.
   protected TableModel getRealModel() {
      return realModel;
   }

   private TableModel realModel; // We're decorating this model
}
```

Notice that public `TableModelDecorator` methods do nothing but forward to the real model. That makes them good defaults for other classes to inherit when they extend `TableModelDecorator`. Example 7 lists `TableSortDecorator`.

Example 7. TableSortDecorator

```
import javax.swing.table.TableModel;

public abstract class TableSortDecorator extends
                                          TableModelDecorator {
   // Extensions of TableSortDecorator must implement the
   // abstract sort method, in addition to tableChanged. The
   // latter is required because TableModelDecorator
   // implements the TableModelListener interface.
   abstract public void sort(int column);

   public TableSortDecorator(TableModel realModel) {
      super(realModel);
   }
}
```

The abstract `TableSortDecorator` class defines one abstract method -- `sort()` -- that must be implemented by concrete subclasses. The `TableBubbleSortDecorator`, listed in Example 8, is one such class.

Example 8. TableBubbleSortDecorator

```
import javax.swing.table.TableModel;
import javax.swing.event.TableModelEvent;

public class TableBubbleSortDecorator extends TableSortDecorator {
   // The lone constructor must be passed a reference to a
   // TableModel. This class decorates that model with additional
   // sorting functionality.
   public TableBubbleSortDecorator(TableModel model) {
      super(model);
      allocate();
   }

   // tableChanged is defined in TableModelListener, which
   // is implemented by TableSortDecorator.
   public void tableChanged(TableModelEvent e) {
      allocate();
   }

   // Two TableModel methods are overridden from
   // TableModelDecorator ...
   public Object getValueAt(int row, int column) {
      return getRealModel().getValueAt(indexes[row], column);
   }
   public void setValueAt(Object aValue, int row, int column) {
      getRealModel().setValueAt(aValue, indexes[row], column);
   }

   // The following methods perform the bubble sort ...

   public void sort(int column) {
      int rowCount = getRowCount();

      for(int i=0; i < rowCount; i++) {
         for(int j = i+1; j < rowCount; j++) {
            if(compare(indexes[i], indexes[j], column) < 0) {
               swap(i,j);
            }
         }
      }
   }
   private void swap(int i, int j) {
      int tmp = indexes[i];
      indexes[i] = indexes[j];
      indexes[j] = tmp;
```

```
    }
    private int compare(int i, int j, int column) {
        TableModel realModel = getRealModel();
        Object io = realModel.getValueAt(i,column);
        Object jo = realModel.getValueAt(j,column);

        int c = jo.toString().compareTo(io.toString());
        return (c < 0) ? -1 : ((c > 0) ? 1 : 0);
    }
    private void allocate() {
        indexes = new int[getRowCount()];
        for(int i=0; i < indexes.length; ++i) {
            indexes[i] = i;
        }
    }
    private int indexes[];
}
```

With the refactoring of the original table sort decorator, we now have a hierarchy of table decorators that we can extend at will.

Decorator applicability
In Java, developers typically use inheritance to add functionality to objects. Base classes implement shared behavior, and extensions add functionality. For example, instead of the sort and filter decorators discussed above, you could implement `SortModel`, `FilterModel`, and, perhaps, `SortFilterModel` classes.

Decorators prove more flexible than inheritance because the relationship between decorators and the objects they decorate can change at runtime, but relationships between base classes and their extensions are fixed at compile time.

Of course, both inheritance and the Decorator pattern have their places; in fact, inheritance is used to implement the Decorator pattern. So when should you use the Decorator pattern? It's useful when:

- You have many simple behaviors you would like to combine in numerous ways at runtime. Implementing the functionality with inheritance would result in a subclass explosion for every possible combination of behaviors.

- You need to transparently add functionality to individual objects at runtime. *Transparently* means that existing classes need not have been modified to support the extra functionality; indeed, those classes are not aware that they are being decorated.

- You want to restrict the use of an object's public methods. Instead of forwarding calls to a restricted method, a decorator can veto a method by throwing an exception from the wrapper method.

For your Enjoyment: Not to be tested.

Abstract Subclasses (Mixins) - Composable Behaviors

References

- Mixin-basedInheritance
- Using C++ Templates to Implement Role-Based Designs
- Working code for the program of this lecture is in Mixin.java
- Archiveof the Java Genericity mailing list
- JavaSpecification Request: Generic Types
- Decoratorpattern
- Chainof Responsibility pattern

Aims

- Understanding how to express abstract subclasses for reusable behavioral extensions in languages such as Java thathave single inheritance and no genericity

Lecture

Genericity

Genericity is the ability to *parameterize classes by otherclasses*. A familiar example of genericity are templates in C++.However, templates lack the ability to constrain explicitly the classesused as formal template parameters and to check such constraints at thepoint of template instantiation. Instead, the capabilities of thoseclasses are implicit in the way their instances are used inside themethods of the class template. This frequently leads toincomprehensible error messages buried deep inside the class templatebody.

We now look at a hypothetical extension of Java with boundedgenericity. In this extension, which borrows from C++ template syntax,formal class parameters can be constrained by existing interfaces. Onlyclasses that implement these interfaces are accepted as actual classparameters to a generic class.

In the following example of a generic sorted list parameterized byitem type, only item types implementing the Comparable interface areallowed so that the compareTo method can be used in the implementationof the sorted list class.

```
class SortedList<Item implements Comparable> {
    public void insert(Item i) {
        ... i.compareTo(j)) ...
    }
}
```

Abstract subclasses as composable behaviors

Usually, (single) inheritance allows us to extend or specialize anexisting (single) superclass. We are required to decide on a specificsuperclass. However, we would get much higher reuse if we could specifysubclasses in such a way that we can apply them to differentsuperclasses as needed. Such subclasses are called *abstractsubclasses* or *mixins*. The following example illustratesthis idea.

We start out with a basic Collection interface and two typicalimplementations.

```
interface Collection {
    void insert(Object item);
```

```
    Object remove();
    void clear();
    boolean isEmpty();
}

class Stack implements Collection {
    public void clear() { while (! isEmpty()) remove(); }
    // ...
}

class Queue implements Collection { ... }
```

Based on this interface, we build several extensions that areparameterized by the superclass. Specifically, these extensions can beapplied to any superclass that implements the Collection interface. Thefirst extension called SizeOf adds to any class implementing Collectionthe ability to keep track of its number of elements.

```
class SizeOf<Base implements Collection> extends Base {
    private int count;
    public void insert(Object item) {
        super.insert(item);
        count ++;
    }
    public Object remove() {
        Object result = super.remove(item);
        if (result != null) {
            count --;
        }
        return result;
    }
    public int size() { return count; }
}
```

The second extension called Graphical adds to any class implementingCollection the capability to represent visually (on the screen) theitems it currently contains in such a way that the visualrepresentation is updated as items are inserted or removed.

```
class Graphical<Base implements Collection> extends Base {
    public void insert(Object item) {
        super.insert(item);
        // add item to the visual representation of Collection
    }
    public Object remove() {
        Object result = super.remove(item);
        // remove item from the visual representation of Collection
        return result;
    }
}
```

We can now compose the extensions with the existing Collectionimplementations. The following code builds a stack that keeps a countof its elements and updates its visual representation.

```
SizeOf<Graphical<Stack>> s = new SizeOf<Graphical<Stack>>();
s.insert("hello");
s.insert("world");
// visual representation now shows two items
System.out.println(s.size()); // prints 2
s.clear();
// visual representation now shows zero items
System.out.println(s.size()); // prints 0
```

Note that the clear method inherited from Stack automaticallyinvokes the right version of remove, namely the one that keeps track ofthe visual representation, which invokes the one that decrements thecount.

Implementation using C++ templates

It is straightforward to use C++ templates to implement thepreceding example. However, C++ templates do not provide the ability to constrain the base class explicitly.

Implementation in Java?

Java lacks genericity and multiple inheritance. How to simulateabstract subclasses?

- Single inheritance: if we fix the superclass, then the abstract subclass is not reusable.

- Composition/chain of responsibility: methods of the superclass are invoked instead of those in the abstract subclass. For example, if we make SizeOf a wrapper class aroundany Collection as in the chain-of-responsibility pattern, thenclear will invoke that Collection's remove instead of SizeOf'sremove!

```
class SizeOf implements Collection {
    Collection zuper;
    private int count;
    public SizeOf(Collection zuper) { this.zuper = zuper; }
    public void insert(Object item) {
        zuper.insert(item);
        count ++;
    }
    public Object remove() {
        Object result = zuper.remove(item);
        if (result != null) {
            count --;
        }
        return result;
    }
```

```
    public int size() { return count; }
    public void clear() {
        // ultimately invokes wrong remove :-(
        zuper.clear();
    }
    public boolean isEmpty() { return zuper.isEmpty(); }
}
```

This illustrates the crucial difference between inheritance and itsapproximation through composition: composition does not handle dynamicmethod binding correctly! This aspect makes the problem much deeper than a simple decorator or chainof responsibility problem.

A possible implementation that solves this problem by recreating the inheritance chain explicitly is given in Mixin.java.

An application of this technique to the interpreters in the simpleimperative language is given in SimpleImperativeMixin.java.