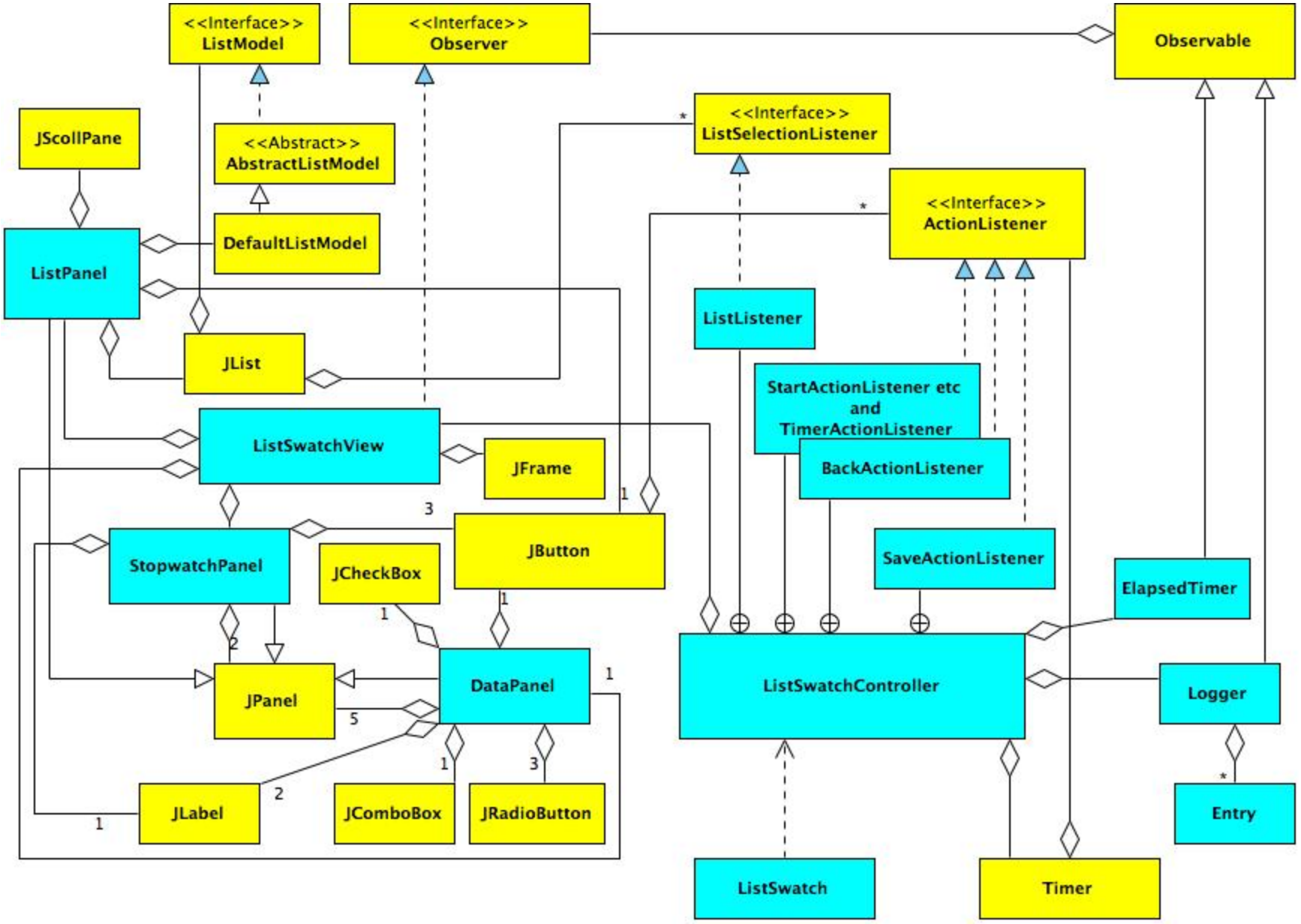
Lecture 8
Completing ListSwatch and introducing TableSwatch

ListSwatch - class diagram



ListSwatchController – declarations and constructor

```
public class ListSwatchController {
     . . .
    private ListSwatchView theView;
    private Logger theLogger;

    public ListSwatchController() {
        elapsedTimer = new ElapsedTimer();
        theView = new ListSwatchView(this);
        theView.setResetState();
        elapsedTimer.addObserver(theView);
        theLogger = new Logger();
        theLogger.addObserver(theView);
        aTimer = new Timer(50, new TimerActionListener());
    }
```

- The class now has a reference to a ListSwatchView (instead of LoggingStopwatchView)
- The constructor now creates the ListSwatchView instance, the rest of Constructor is the same of for LoggingStopwatchController.

ListSwatchController – ListListener inner class

```
public class ListSwatchController
  . . .
  public class ListListener implements ListSelectionListener {
   public void valueChanged(ListSelectionEvent le) {
      if (theState == LIST_DISPLAY && !le.getValueIsAdjusting()) {
        JList list = (JList) (le.getSource());
        lastIndexSelected = list.getSelectedIndex();
        if (lastIndexSelected != -1) {
          Object selection = list.getSelectedValue();
          System.out.println(selection + " at position ” + lastIndexSelected);
        }
      }
  }
  . . .
)
```

- For commentary on this, see next slide

ListSwatchController – ListListener inner class, explanation

- The *valueChanged()* method must be included to conform to the ListSelectionListener interface. It is called every time a list selection is made.
- The method *getValueAdjusting()* is called on the ListSelectEvent parameter *le* to determine if the selection was one of a rapid series of events. The method exits if this is the case.
- Otherwise first the JList event source is obtained by invoking *getSource()* on *le* (this returns an Object which needs to be cast to JList).
- Then *getSelectedIndex()* is invoked on the JList to obtain the *int* index value (starting from 0) of the selected item, and if the returned value is -1 then no selection has been made and the method exits.
- Otherwise *getSelectedValue()* is invoked on the JList, to obtain the actual content of the selection. This is returned as an Object. Printing it out using System.out.println(. . .) prints the String value to the console.

ListSwatchController – SaveActionListener inner class

```
public class ListSwatchController
  . . .
  public class SaveActionListener implements ActionListener {
     public void actionPerformed(ActionEvent event) {
           if (theState == STOPPED){
              . . .
              theLogger.log(elapsedTimer.getTime(),
                            theView.getName(),
                            theView.getClub(),
                            theView.getSpeedQuality(),
                            theView.getAccepted());
               elapsedTimer.reset();
               theView.setListDisplayState();
               theState = LIST_DISPLAY;
           }
     }
  }
```

- This is similar to SaveActionListener in LoggingStopwatchController, except here *setListDisplayState()* is invoked on the *theView* and *theState* changes to LIST_DISPLAY (see state diagram of Lecture 7)

ListSwatchController – BackActionListener inner class

```
public class ListSwatchController
    . . .
    public class BackActionListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
              if (theState == LIST_DISPLAY) {
                  theView.setResetState();
                  theState = RESET;
              }
          }
        }
    }
```

- This *actionPerformed(…)* method of this inner class is invoked when the Back button is pressed on the screen displaying the JTable (i.e. in the LIST_DISPLAY state)
- *setResetState()* is invoked on theView (which toggles display back to the Panel showing the main Stopwatch and Data panels), and *theState* transitions to RESET

The Logger class - 1

```
public class Logger extends Observable {
  private ArrayList<Entry> records;
  public Logger() {
    super();
    records = new ArrayList <Entry> ();
  }
  public void log(String time,String name, String club,
                          String quality, boolean accepted){
   . . .
    records.add(new Entry (time, name, club,quality,accepted));
    this.setChanged();
    this.notifyObservers(Properties.RECORDS);
  }
```

- This model class is a modified version of the Logger class used for LoggingStopwatch. It aggregates an ArrayList of Entrys each of which contains the basic data for each run. The constructor creates the ArrayList
- The *log()* method takes the basic data for the run, constructs an Entry object from this data and adds the Entry to the ArrayList
- The *setChanged()* and *notifyObservers(..)* are called, the latter with a property value that resolves to "records" to indicate to the observer (TableSwatchView) the notification type.

The Logger class - 2

```
  public Entry getLastEntry(){
      return lastEntry;
  }

  public Entry[] getRecords(){
      return records.toArray(new Entry[records.size()]);
  }
}//End of Logger
```

- The first method gets the last entry entered. This method will be called by the view (TableSwatchView) after it has been notified of the change via *notifyObservers(…)* – see previous slide
- This second method returns an array of all the Entrys added so far. It converts the internally stored ArrayList into an array using the *toArray(…)* method of ArrayList.

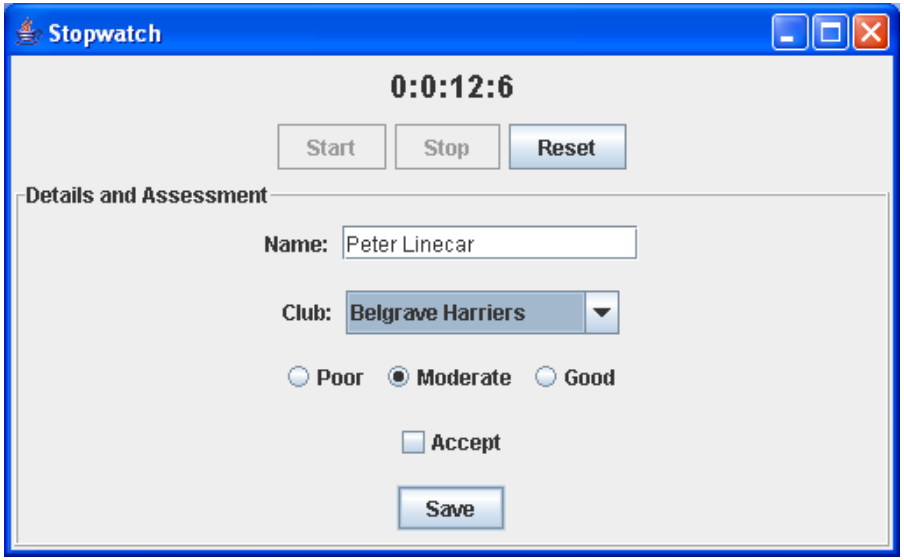
The Entry class -1

```
public class Entry {
    private String time ;
    private String name;
    private String club;
    private String quality;
    private boolean accepted;

    public Entry(String time, String name, String club,
                 String quality, boolean accepted) {
        this.time = time;
        this.name = name;
        this.club = club;
        this.quality = quality;
        this.accepted = accepted;
    }

    public String getTime() {
        return this.time;
    }
```

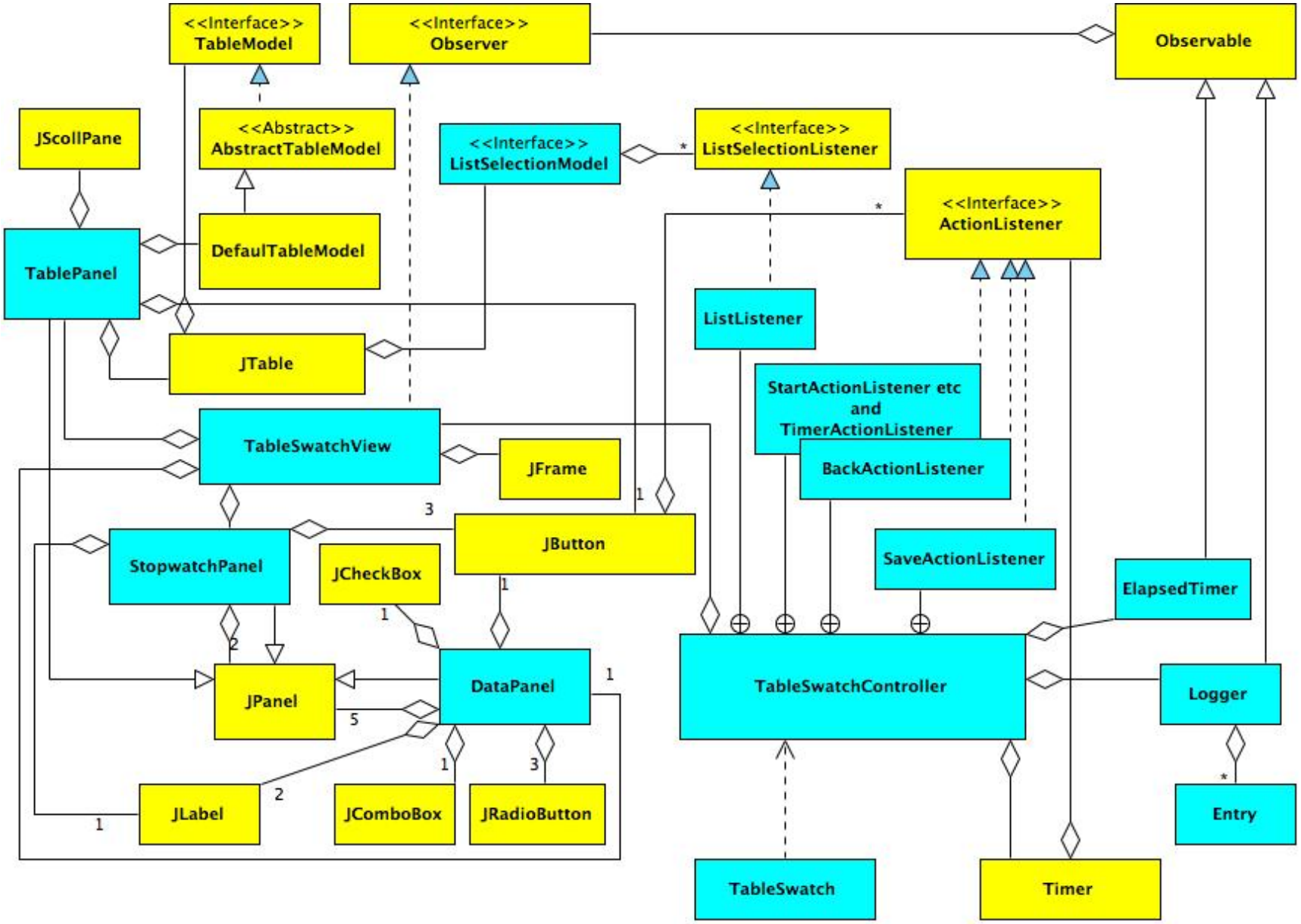
Entry class -2

```
    public String getName() {
        return this.name;
    }
    . . .
    public String getClub() {
        return this.club;
    }
    . . .
    public String getQuality() {
        return this.quality;
    }
    . . .
    public String getAccepted() {
        if (accepted){
            return "accepted";
        } else {
            return "not accepted";
        }
    }
    . . .//toString() method not shown here...
}//End of Entry
```

The *TableSwatch* artefact



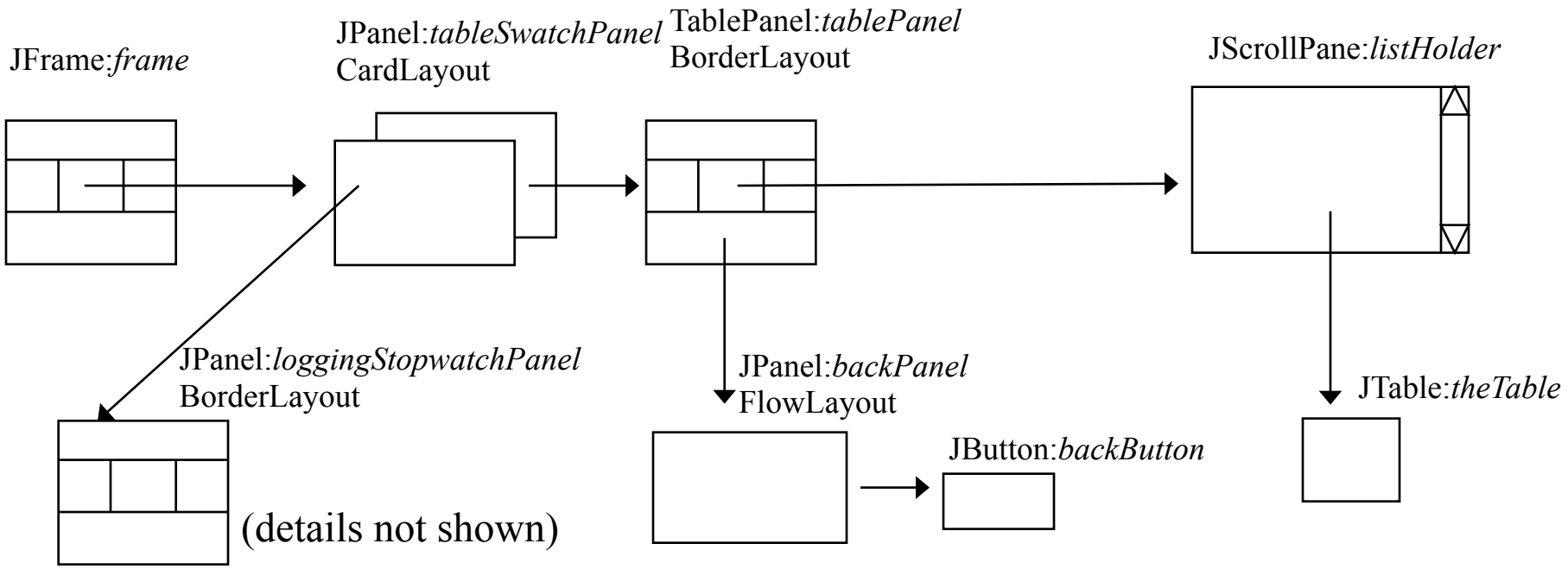
TableSwatch artefact - class diagram



TableSwatch artefact - class diagram cont'd

- The previous slide shows that the TableSwatch artefact has a very similar three layer structure to ListSwatch. The main difference is that TablePanel (which replaces ListPanel from ListSwatch) now has a JTable called *theTable*
- TablePanel also has a DefaultTableModel called *theTableModel*. This used to contain the "model" data behind the JTable. The association is made via JTable's method *setModel().*
- Having direct access to the "model" is the only way to add rows to the table. This is done by invoking the *addRow()* method of DefaultTableModel
- JTable is also connected internally to an object whose class implements ListSelectionModel - which keeps a record of the current row selected in the table
- A ListSelectionListener can be attached to the ListSelectionModel to detect whenever the selection is changed
- In this case the listener is actually an instance of ListListener – an inner class of TableSwatchController

TableSwatch - Layout management



- The layout management is very similar to that for ListSwatch - the JList is replaced by the JTable inside the JScrollPane

The JTable class

- Swing tables include rows and columns of cells and column headers.
- The JTable class, like other Swing components has an internal Model-View-Controller structure
- A JTable has an associated model, which it sees via the TableModel interface. This determines the data contents.
- The class for the model implements TableModel. A commonly used model is DefaultTableModel
  - which derives from AbstractTableModel which implements TableModel.

The JTable class - some methods

- `public void` **`setModel`**`(TableModel dataModel)`
  Sets the data model for this table to newModel and registers with it for listener notifications from the new data model.
- `public void` **`setPreferredScrollableViewportSize`**`(`
  `Dimension size)`
  Sets the preferred size of the viewport for this table. This will effect the size of the viewport (the visible area) of a JScrollPane into which the JTable is added
- `public void` **`setEnabled`**`(boolean enabled)`
  Sets whether or not this component is enabled. This method is in the parent class JComponent. The table is not editable if *enabled* is false.

The class DefaultTableModel - some of API

- `public DefaultTableModel(Vector columnNames, int numRows)`
  Constructs a DefaultTableModel with as many columns as there are elements in *columnNames* and *numRows* of null object values.
- `public void addRow(Vector rowData)`
  Adds a row to the end of the model. The class Vector is very similar to the class ArrayList
- `public void removeRow(int row)`
  Removes the specified row at the index *row*
- `public int getRowCount()`
  Returns the number of rows in the table
- `public void setDataVector(Vector vectorOfVectors, Vector columnIdentifiers)`
  Replaces all the data of the table. Each row is represented as a Vector of Object values. All these Vectors are then themselves contained as elements of the Vector called *vectorOfVectors*. The Vector called *columnIdentifiers* contains the names of the new columns.

TablePanel - header and declarations

```
public class TablePanel extends JPanel{
  . . .
    private JTable theTable;
    private DefaultTableModel theTableModel;
    private ListSelectionModel listSelectionModel;
    private Vector columnNames;
```

- The four extra fields are
  - a reference to a JTable called *theTable* on which the history of saved times/ data will be displayed
  - a reference to a DefaultTableModel called *theTableModel* which is the data "model" for *theTable*
  - a reference to a ListSelectionModel called *listSelectionModel* which stores information about the current selected row
  - A Vector called *columnNames* to hold the titles of the columns in the table

TablePanel - constructor 1

```
public TablePanel(TableSwatchController controller) {
    . ..
     theTable = new JTable();
     tableHolder = new JScrollPane(theTable,
             JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
             JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
     columnNames = new Vector();
     columnNames.add("Time");
     columnNames.add("Name");

     columnNames.add("Club");
     columnNames.add("Quality");
     columnNames.add("Accepted");
```

- As for ListPanel, the constructor gets passed, as parameter, an instance of the main controller class
- The JTable is instantiated and placed inside a newly constructed JScrollPane
- The *columnNames* Vector object is instantiated and filled with the names of the columns for the table (Vector is similar to ArrayList)

TablePanel - constructor 2

```
theTableModel = new DefaultTableModel(columnNames,0);
theTable.setModel(theTableModel);
theTable.setPreferredScrollableViewportSize(
                                    new Dimension(500,100));
theTable.setEnabled(true);
listSelectionModel = theTable.getSelectionModel();
listSelectionModel.setSelectionMode(
                    ListSelectionModel.SINGLE_SELECTION);
```

- The DefaultTableModel called *theTableModel* is constructed based on *columnNames* with initially 0 rows
- This newly constructed DefaultTableModel is associated with the JTable as its "model", such that updates in this "model" will get reflected in the "view" of the JTable in the operation phase
- The preferred size of the viewport is set and user interaction is enabled for the JTable
- The ListSelectionModel associated with the JTable is then obtained and stored in the field *listSelectionModel.* It is set up for single row selection.

TablePanel - constructor 3

```
    listSelectionModel.addListSelectionListener(controller.new ListListener());
    listSelectionModel.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);
    backButton = new JButton("Back");
    backPanel = new JPanel();
    backPanel.add(backButton);
    this.add(tableHolder, BorderLayout.CENTER);
    this.add(backPanel, BorderLayout.SOUTH);
    backButton.addActionListener(controller.new BackActionListener());
}//end of TablePanel constructor
```

- A new instance of ListListener (an inner class of ListSwatchController) is constructed and passed as the parameter to *addListSelectionListener(…)* of the ListSelectionModel. This will have the effect, in the operation phase, that the ListListener will have its *valueChanged(…)* method called whenever the user clicks on an item in the JList.
- *backButton* (an instance of JButton) is constructed and a new instance of BackActionListener (another inner class of ListSwatchController) is registered as its ActionListener.

TablePanel *addEntry()*

```
public void addEntry(Entry element) {
    Vector itemParts = new Vector();
    itemParts.add(element.getTime());
    itemParts.add(element.getName());
    itemParts.add(element.getClub());
    itemParts.add(element.getQuality());
    itemParts.add(element.getAccepted());
    theTableModel.addRow(itemParts);

}
```

Called whenever an entry is added to the Logger (via the MVC mechanism)

- A Vector is created and the Strings comprising the different parts of the Entry (supplied as parameter) are extracted and added to the Vector
- The Vector is then supplied as the parameter to *addRow(...)* method on the DefaultTableModel
- This will cause the visual component of the JTable (in which it has been inserted as the model) to be updated

TablePanel *getListSelection()*

```
public Object getListSelection() {
   if (listSelectionModel.getMinSelectionIndex() == -1){
     throw new RuntimeException(
      "there is no selected item in getListSelection()");
   }
   Vector dataVector = theTableModel.getDataVector();
   Vector row =(Vector)dataVector.elementAt(
   listSelectionModel.getMinSelectionIndex());
   StringBuffer sb = new StringBuffer();
   for (int i = 0;i<row.size();i++){ sb.append(row.elementAt(i) + " ");}
   return sb.toString();
}
```

- If there is currently no selection, A RuntimeException is thrown and the program exits since this method should not have been called under such a condition.
- Otherwise *dataVector* - the Vector of Vectors containing all the table data is obtained from *theTableModel*. The position in the table of the current selection is obtained from *listSelectionModel* via *getMinSelectionIndex()* and this index is used to extract the Vector containing the required row data from *dataVector*
- Each element of this row is then obtained and appended to a temporary StringBuffer whose String contents are finally returned to the caller

TableSwatchController and TableSwatchView

- The code for TableSwatchController and TableSwatchView is almost exactly the same as for ListSwatchController and ListSwatchView and is not shown here.
  - View the full code of the TableSwatch artefact for further understanding
- The code for the model classes ElapsedTimer and Logger are unchanged. This shows the advantages of modularity and reuse when we use the Model View Controller architecture to separate out presentation, event handling and application model functionality.