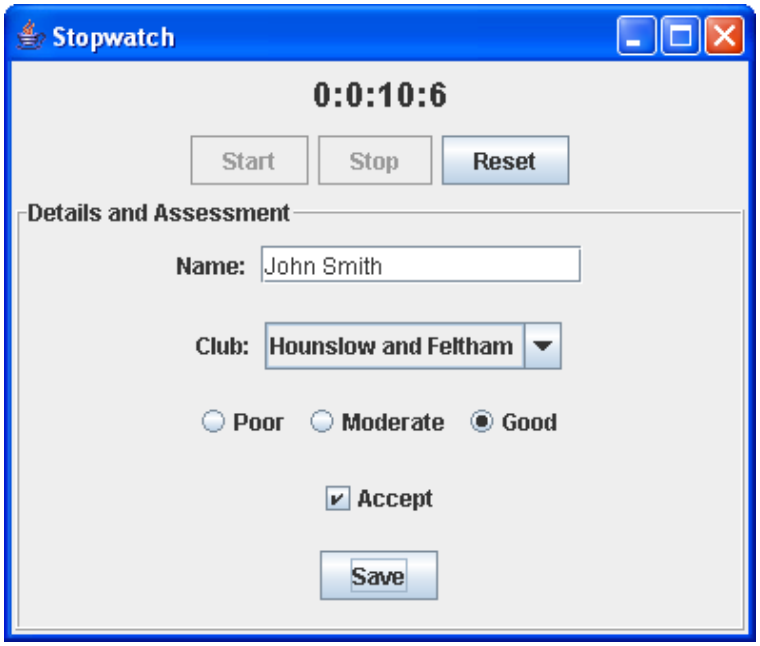
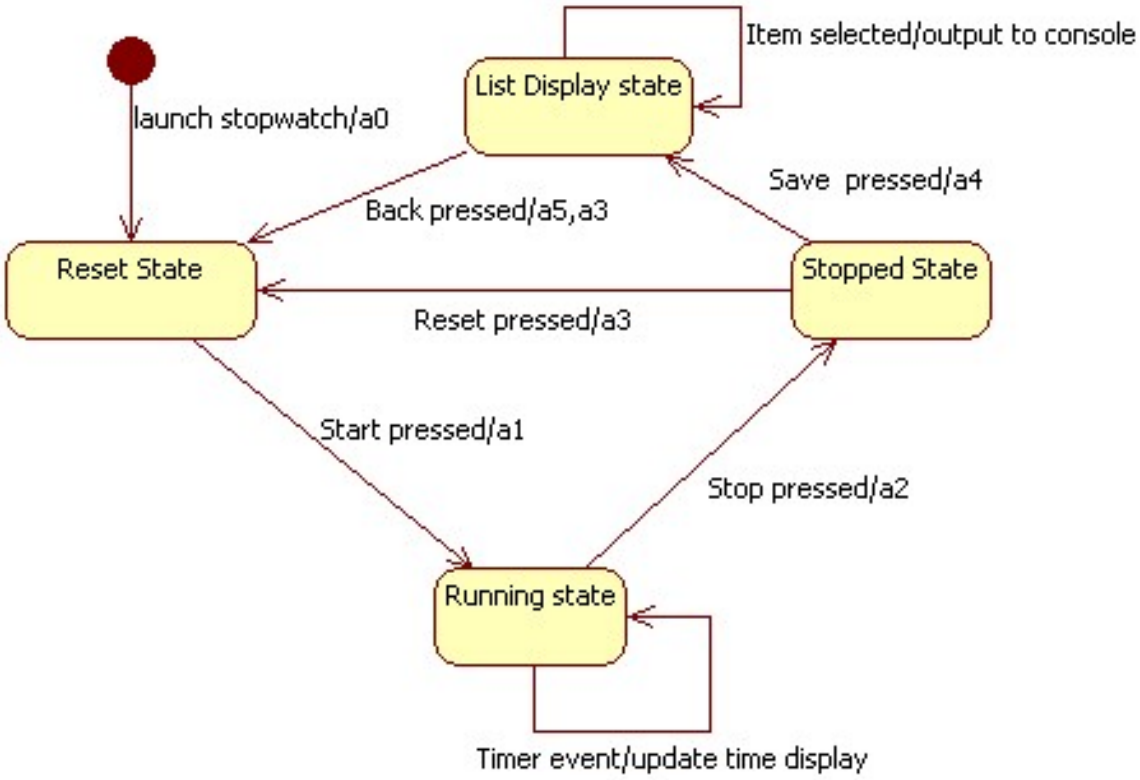
Lecture 7
The ListSwatch artefact
- CardLayout, JScrollPane, JList

The *ListSwatch* artefact



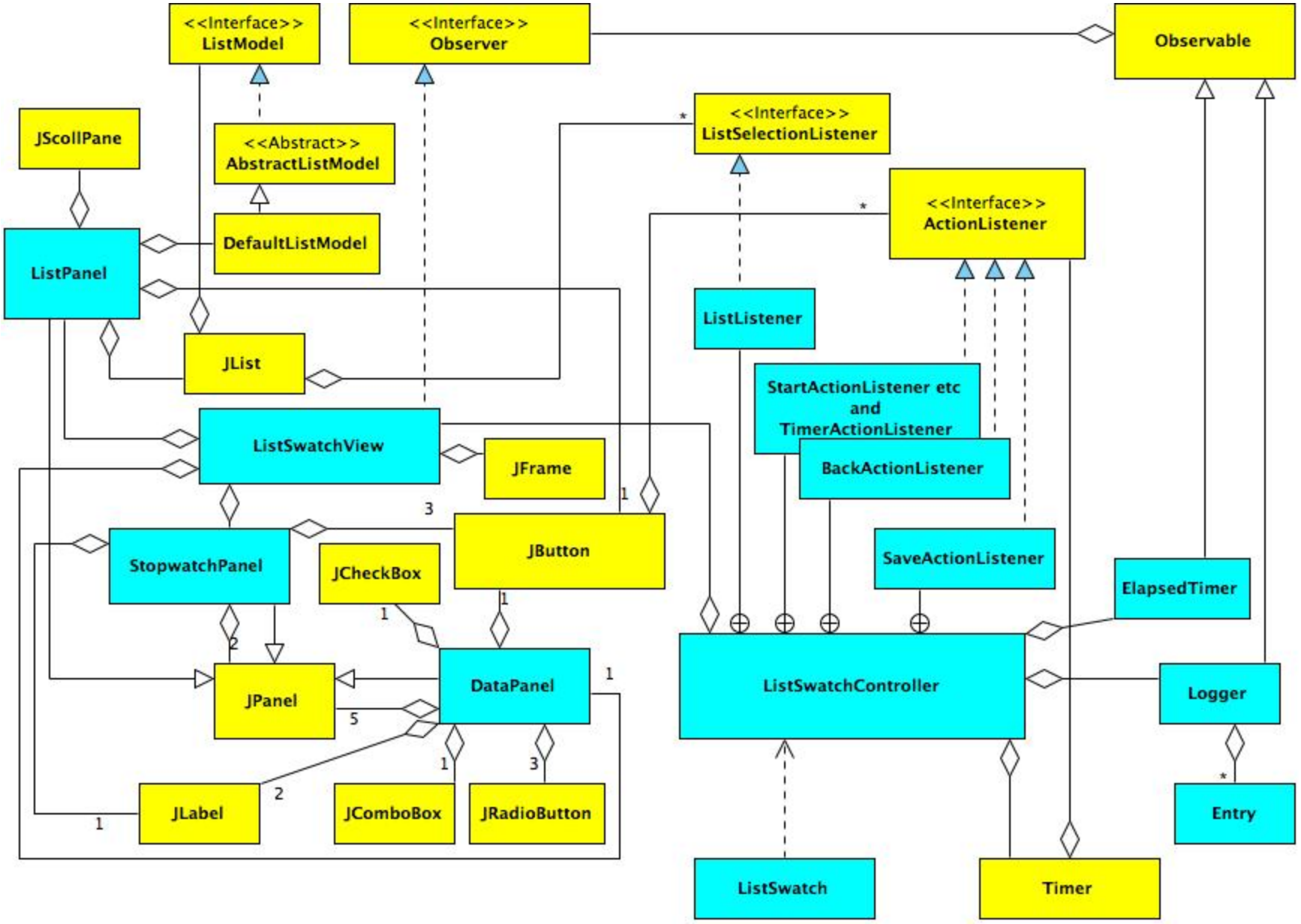
- When the user presses the Save button, the screen switches to the second screen, showing a JList inside a JScrollPane - giving a list of the timings carried out so far
- When the user clicks on one of the entries, text related to the entry is output to the console window
- The user can go back to the main screen by pressing the Back button

ListSwatch state transition diagram



a0: enable Start, disable Stop, Reset, Save, data entry components
a1: start timing, enable Stop, disable Start, Reset, data entry components
a2: enable Reset, data entry components, disable Start, Stop
a3: reset dispaly, enable Start, disable Stop, disable Reset, data entry components
a4: switch to data summary screen
a5: switch to main screen

ListSwatch - class diagram



ListSwatch artefact - class diagram cont'd

- The previous slide shows a similar three layer structure to Stopwatch and LoggingStopwatch. A new class ListPanel (a subclass of JPanel) is aggregated by ListSwatchView.
- The class Logger is has been changed to be a subclass of Observable and now stores the results of runs.
- Logger aggregates Entrys which contain the data for each run
- Logger notifies the Observer(s), in this case just ListSwatchView, via *notifyObservers(…)* in the superclass Observable, whenever a new piece of run data is added.

ListSwatch artefact - class diagram cont'd 2

- ListPanel aggregates a JList (a Swing class). It also aggregates a DefaultListModel (also a Swing class). This is used to contain the "model" data behind the JList. The association is made via JList's method *setModel().*
  - This is an example of the internal "Model View Controller" architecture of Swing components
- Having direct access to the "model" is the only way to add items to the list. This is done by invoking the *addItem()* method of DefaultListModel.
- Also shown is in extra inner class ListController of the main controller class. It implements another interface ListSelectionListener and the JList has ListController registered as a listener for ListSelectionEvents.
- This lecture will examine ListPanel and changes to ListSwatchView and the next lecture will examine ListSwatchController and the new version of Logger.

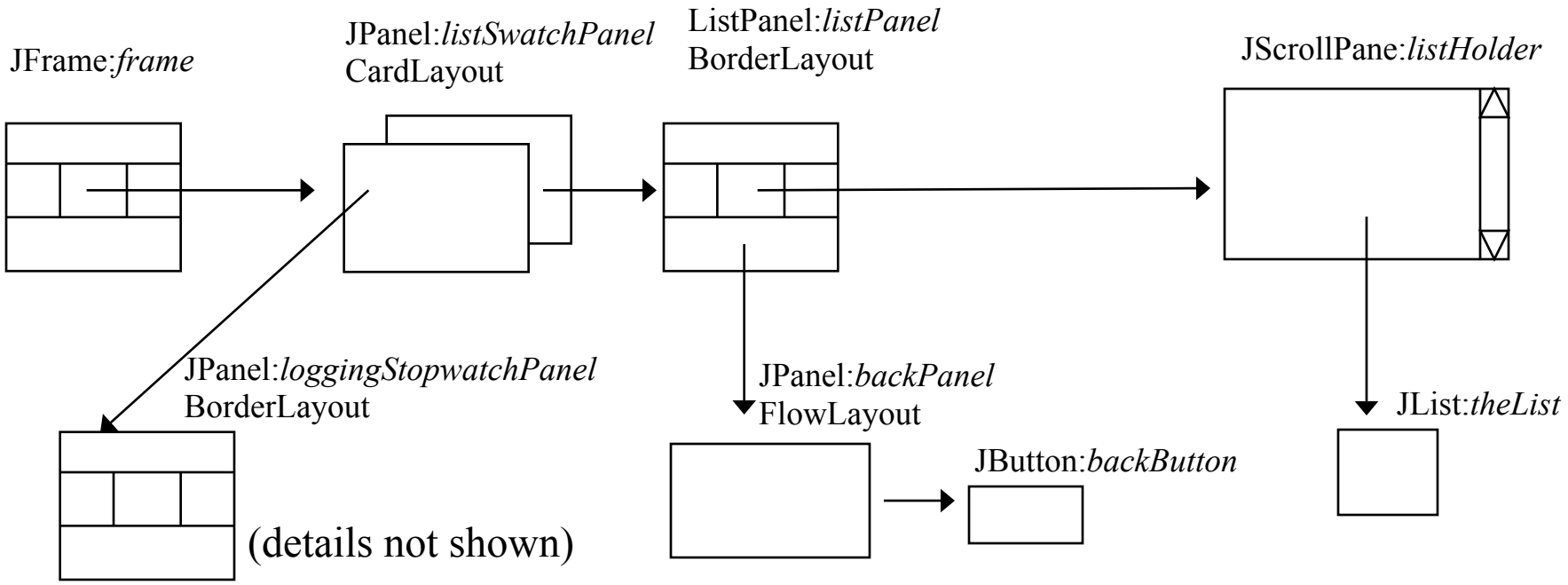
The interface ListSelectionListener

- This interface has a single method declaration:

```
public void valueChanged(ListSelectionEvent e);
```

- Any class that implements ActionListener must implement its method *actionPerformed(…)* This is the method that gets called when, for example, a JButton to which the listener has been registered is pressed in the operation phase, an ActionEvent being passed as its parameter.
- Similarly any class that implements ListSelectionListener must implement its method *valueChanged(. . .),* a ListSelectionEvent being passed as its parameter. This method will get called whenever, in the operation phase, the user makes a selection on the JList to which the ListSelectionListener has been registered

ListSwatch artefact - Layout management



- Shown above is a JPanel laid out using the CardLayout layout manager. This enables panels to be laid out on top of one another with only one being visible at any one time. In this case there are two. This JPanel sits on the JFrame that is part of ListSwatchView.
- Also shown is a JScrollPane that gives the contained JList scrolling behaviour
- Details of the *loggingStopwatchPanel* are not shown here

CardLayout layout manager 1

- Setting a Container to be managed by CardLayout is similar to setting up other layout managers. In ListSwatchView, in the constructor (preparation phase) a field called *cards* references a CardLayout object:

```
cards = new CardLayout();
```

- Then (again in the constructor) it is associated with a JPanel:

```
listSwatchPanel = new JPanel();
listSwatchPanel.setLayout(cards);
```

- Then (again in the constructor for ListSwatchView) JPanels are added to the parent CardLayout-managed JPanel with strings associated with each sub-JPanel:

```
listSwatchPanel.add(loggingStopwatchPanel,"Stopwatch");
listSwatchPanel.add(listPanel,"History");
```

  - in the general case several sub-components can be added to such a parent Container managed by CardLayout

CardLayout layout manager 2

- In the operation phase the method *show()* of CardLayout can be called to cause either one or the other sub-JPanel to be brought to the front:

```
cards.show(listSwatchPanel,"Stopwatch");
```

or

```
cards.show(listSwatchPanel,"History");
```

The class DefaultListModel and the interface ListModel

- This is one of the classes that can be used to supply the "model" data to a JList. An instance of it can be plugged into a JList at run time.
- DefaultListModel extends the abstract class AbstractListModel which in turn implements the interface ListModel.
- JList "sees" its associated data model via the ListModel interface which means that different data models can be plugged into a JList.
- DefaultListModel has, as some of its methods:

void addElement (Object obj)

Adds the specified component to the end of this list. As a result the visual display of the JList into which the DefaultListModel is plugged will have a line added

void clear ()

Removes all content from the list. As a result the visual display of the JList into which the DefaultListModel is plugged will be cleared

ListSwatchView –field declarations

```
public class ListSwatchView implements Observer {   . . .
   . . .
   private StopwatchPanel stopwatchPanel;
   private DataPanel dataPanel;
   private ListPanel listPanel;
   private JPanel listSwatchPanel;
   private JPanel loggingStopwatchPanel;
   private CardLayout cards;
    private JFrame frame;
    . . .
```

- Declared, but not instantiated, here are
  - the extra JPanels, as indicated on the layout management diagram
  - *cards,* the CardLayout layout manager that will be used to lay out the top level JPanel
  - *frame* – the JFrame on which the top level JPanel *listSwatchPanel* sits

ListSwatchView – the constructor

```
public ListSwatchView(ListSwatchController controller) {
      . . .
    frame = new JFrame("Stopwatch");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    stopwatchPanel = new StopwatchPanel(controller);
    dataPanel = new DataPanel(controller);
    loggingStopwatchPanel = new JPanel();
    loggingStopwatchPanel.setLayout(new BorderLayout());
    loggingStopwatchPanel.add(stopwatchPanel, BorderLayout.NORTH);
    loggingStopwatchPanel.add(dataPanel, BorderLayout.SOUTH);
    listPanel = new ListPanel(controller);
    . . .
```

- The StopwatchPanel and the DataPanel are constructed based on the supplied ListSwatchController parameter
- These are then placed in the NORTH and SOUTH of *loggingStopwatchPanel*, the newly constructed JPanel laid out using BorderLayout
- The ListPanel is then constructed, based on the supplied ListSwatchController parameter.

ListSwatchView – the constructor (cont'd)

```
public ListSwatchView(ListSwatchController controller) {
        . . .
     cards = new CardLayout();
     listSwatchPanel = new JPanel();
     listSwatchPanel.setLayout(cards);
     listSwatchPanel.add(loggingStopwatchPanel, "stopwatch");
     listSwatchPanel.add(listPanel, "history");
     frame.add(listSwatchPanel);
     frame.pack();
}
```

- The CardLayout *cards* is constructed
- The top level ListSwatchPanel is then constructed and laid out using this CardLayout
- Each of the panels *loggingStopwatchPanel* and *listPanel* are then added as cards labelled "stopwatch" and "history" to *listSwatchPanel*
- Finally *listSwatchPanel* (which is the top level panel) is added to the JFrame which is then *pack()*ed.

ListSwatchView – the the *update()* method

```
  @Override
  public void update(Observable observable, Object arg) {
    if (arg.equals(Properties.TIME)) {
         stopwatchPanel.setTime(((ElapsedTimer)observable).getTime());
    } else if (arg.equals(Properties.RECORDS)) {
            listPanel.addEntry(((Logger)observable).getLastEntry());
        }
}
```

- As in the original version of Stopwatch, if the value of the second parameter ***arg*** is "time" (which is the value of Properties.*TIME*), then *getTime()* is called on the object referenced by the first parameter, and the returned value is passed as parameter to *setTime(..)* called on the StopwatchPanel (updating the time display)
- If the value of the second parameter ***arg*** is "records" (which is the value of Properties.*RECORDS*), then the first parameter is cast to Logger, and the method *getLastEntry()* is called on it. The return value, an instance of Entry, is passed as the parameter to *addEntry(…)* called on the ListPanel, to add a new value to the the on screen list of records.

StopwatchView *setListDisplayState()* and *setResetState()*

```
public void setListDisplayState() {
    . . .
    cards.show(listSwatchPanel,"History");
}

public void setResetState() {
    cards.show(listSwatchPanel,"Stopwatch");
    . . .
}
```

- When these methods are called from ListSwatchController, they bring to the front either the JPanel containing the stopwatch and data fields or else the JPanel containing the JList of timings so far.

ListPanel - fields

```
public class ListPanel extends JPanel {
    private JPanel backPanel;
    private JScrollPane listHolder;
    private DefaultListModel listModel;
    private JList list;
    private JButton backButton;
    . . .
```

- The JPanel *backPanel* will be the panel on which "Back" button will be placed
- The JScrollPane *listHolder* will be used to contain the JList to give it scrolling behaviour
- The DefaultListModel *listModel* will be used to hold the data model (as part of the internal Swing MVC architecture) of the JList
- The JList *list* contains the list of data entries
- The JButton *backButton* will be used to navigate back to the main timing and data entry panel.

ListPanel constructor

```
public ListPanel(ListSwatchController controller) {
  this.setLayout(new BorderLayout());
  list = new JList();
  listHolder = new JScrollPane(list, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
          JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  listModel = new DefaultListModel();
  list.setModel(listModel);
  . . .
```

- The layout manager for the ListPanel is BorderLayout
- The JList is constructed
- The JScrollPane is then constructed. The first parameter of the constructor call indicates that it will hold the JList, giving it scrolling behaviour. The second and third parameters indicate that the vertical and horizontal scroll bars will always be displayed (i.e. not only when needed)
- The DefaultListModel is then constructed
- In the last line this DefaultListModel replaces the ListModel that is part of the JList when it is constructed. The reason for doing this is that now the program can directly access the ListModel – which, as we shall see, it needs to do.

ListPanel constructor (cont'd)

```
public ListPanel(ListSwatchController controller) {
    . . .
    backButton = new JButton("Back");
    backPanel = new JPanel();
    backPanel.add(backButton);
    this.add(listHolder,BorderLayout.CENTER);
    this.add(backPanel,BorderLayout.SOUTH);
    backButton.addActionListener(controller.new BackActionListener());
    list.addListSelectionListener(controller.new ListListener());
}
```

- The "Back" JButton is constructed and placed on a newly constructed JPanel *backPanel* (centred because of default FlowLayout layout manager).
- The newly constructed JScrollPane is placed in the CENTER of the ListPanel and *backPanel* is placed in the SOUTH
- A new instance of the inner class BackActionListener (inside ListSwatchController) is constructed and registered as the listener for the "Back" JButton
- Finally a new instance of the inner class ListListener (inside ListSwatchController) is constructed to act as a listener for user selection events in the operation phase

ListPanel *addRecord(…)*

```
public void addEntry (Entry entry){
     listModel.addElement(entry);
 }
```

This method is called from the *update(…)* method of ListSwatchView – when the Logger model class has an entry added to it.

•Its parameter is an Entry –which stores a set of data taken from a run

•The Entry is supplied as the parameter to the *addElement(…)* method of the internal model maintained by the JList (which was built and inserted into the JList earlier).

•Because of the active MVC internal structure of JList (and all Swing components), altering the internal model of the component directly will result in the dependent internal view of the JList being updated. In this case the String version of the Entry (resulting from applying *toString()* to it) will end up added to the end of the JList on screen.