MVC / Tables

Model / View / Controller

Design
    A decomposition strategy where "presentation" is separated from data
      maintenance
    Smalltalk idea
    Controller is often combined with View, so have Model and View/Controller
Web Version
    Shopping cart model is on the server
    The view is the HTML in front of you
    Form submit = send transaction to the model, it computes the new state,
      sends you back a new view

Modularity

  Writing larger programs in teams, we're always on the lookup for a natural
  dividing line to help use separate our 1000 into two 500 line parts that are as
  independent as possible.
Separating the "Data Model" and "View" ideas is works well and is a common
  modularity strategy.

Model -- aka Data Model

"data model"
Storage, not presentation
Knows data, not pixels
Support data model operations
    cut/copy/paste, File Saving, undo, networked data -- these can be expressed
      just on the model which simplifies things
    e.g. can get the logic for file save or undo working, without worrying about
      pixels.

View

Presentation
Gets all data from model and draws or otherwise renders it for the user

---

Controller

The logic that glues things together.

Manage the relationship between the model and view

1. Most data changes are initiated by user events (keyboard, mouse gestures) that tend to happen on the view side. These are translated to messages to the model which does the actual data maintenance

2. Going the other direction, the view needs to hear about changes in the model. In Java, this is done with a Listener paradigm in Swing.

Usually, the controller is implemented in the view.

Model Role

Respond to getters methods to provide data

Respond to setters to change data

Manage a list of listeners

> Java uses the Model/Listener structure, and it's a good design, although there are other ways to do it. Alternately, the model could not know anything about the views (dumb), and instead there is some Controller that is smart about keeping the view and model in synch both ways.

When receiving a setData() to change the data, notify the listeners of the change (fireXXXChanged)

> Change notifications express the different changes possible on the model. (cell edited, row deleted, ...)

> Iterate through the listeners and notify each about the change.

View Role

Have pointer to model

Don't store any data

Send getData() to model to get data as needed

User edit operations (clicking, typing) in the UI map to setData() messages sent to model

Register as a listener to the model and respond to change notifications

On change notification, consider doing a getData() to get the new values to make the pixels up-to-date with the real data.

Swing Table Classes

JTable -- view

Uses a TableModel for storage

Has all sorts of built-in features to display tabular data.

TableModel -- Interface

The messages that define a table model -- the abstraction is a rectangular area of cells.

getValueAt(), setValueAt(), getRowCount(), getColumnCount(), ...

The table model establishes a co-ordinate system: 0..getRowCount()-1, 0..getColumnCount()-1. The model and the view(s) all use the model coordinate system to identify rows and columns.

TableModelListener -- Interface

Defines the one method tableChanged()

If you want to listen to a TableModel to hear about its changes, implement this interface.

```
public interface TableModelListener extends java.util.EventListener
{
    /**
     * This fine grain notification tells listeners the exact range
     * of cells, rows, or columns that changed.
     */
    public void tableChanged(TableModelEvent e);
}
```

AbstractTableModel

Implements some TableModel utility behavior.

Provides helper utilities for things not directly related to storage
    addTableModelListener(), removeTableModelListener(), ...

fireXXXChanged() convenience methods
    These iterate over the listeners and send the appropriate notification
    fireTableCellUpdated(row, col)
    fireTableRowDeleted(row)
    etc.

getRowCount(), getColumnCount(), and getValueAt() are **abstract** -- they must be provided by a subclass model that actually stores data.

DefaultTableModel

extends AbstractTableModel

Complete implementation with Vector

BasicTableModel Code Points

A complete implementation of TableModel using ArrayList

getValueAt()
    Pulls data out of the ArrayList of ArrayList implementation

setValueAt()
    Changes the data model and uses fireTableXXX (below) to notify the listeners

AbstractTableModel
    Has routine code in it to manage listeners -- add and remove.
    Has fireTableXXX() methods that notify the listeners -- BasicTableModel uses
        these to tell the listeners about changes.
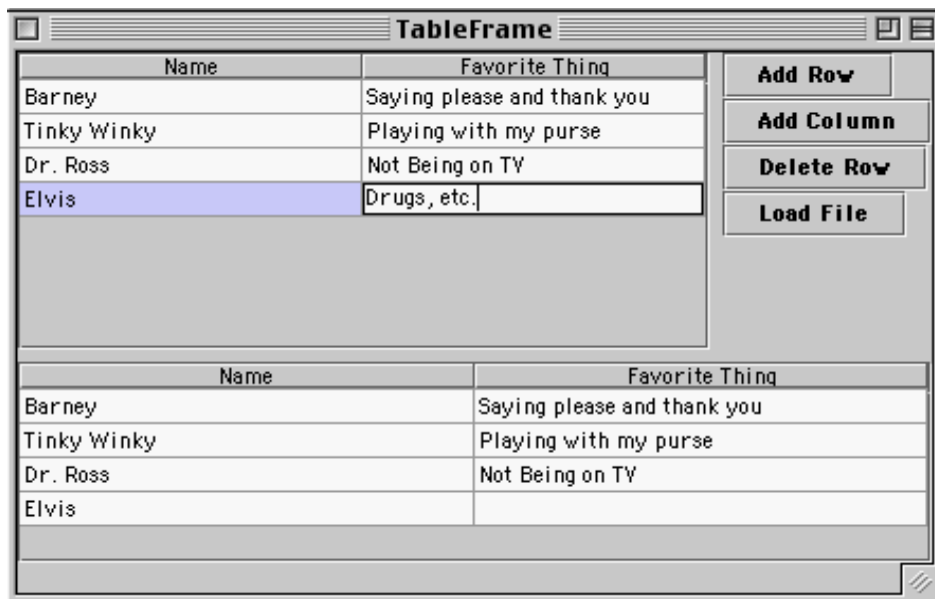
1. Passive Example
    1. Table View points to model
    2. View does model.getXXX to get data to display

2. Row Add Example
    1. Add row button wired to the model
    2. Model changes its state
    3. Model does fireRowAdded() which sends notification to each listener
    4. Listeners get the notification, call getData() as needed

3. Edit Example
    1. Table View1 points to model for its data and listens for changes
    2. Table View2 also points to the model and listens for changes
    3. User clicks/edits data in View1
    4. View1 does a model.setXXX to make the change
    5. Model does a fireDataChanged() -- notifies both listeners
    6. Both views get the notification of change, update their display (getXXX) if
        necessary
    View2 can be smart if View1 changes a row that View2 is not currently
        scrolled to see.

Name	Favorite Thing	
Barney	Saying please and thank you	Add Row
Tinky Winky	Playing with my purse	Add Column
Dr. Ross	Not Being on TV	Delete Row
Elvis	Drugs, etc.	Load File

Name	Favorite Thing
Barney	Saying please and thank you
Tinky Winky	Playing with my purse
Dr. Ross	Not Being on TV
Elvis	

In this case, "Elvis" has been entered, but the return key has not yet been hit for
  the "Sex, drugs, etc." entry

BasicTableModel.java
Demonstrates a complete implementation of TableModel -- stores the data
and generates fireXXXChanged() notifications where necessary.

```
// BasicTableModel.java

/*
 Demonstrate a basic table model implementation
 using ArrayList.

 A row may be shorter than the number of columns
 which complicates the data handling a bit.
*/
import java.awt.*;
import javax.swing.*;

import javax.swing.table.*;

import java.util.*;
import java.io.*;


class BasicTableModel extends AbstractTableModel {
    private ArrayList names;   // the label strings
    private ArrayList data;     // arraylist of arraylists


    public BasicTableModel() {
        super();

        names = new ArrayList();
        data = new ArrayList();
    }


    // Basic Model overrides
    public String getColumnName(int col) {
        return (String) names.get(col);
    }
    public int getColumnCount() { return(names.size()); }
    public int getRowCount() { return(data.size()); }
    public Object getValueAt(int row, int col) {
        ArrayList rowList = (ArrayList) data.get(row);
        String result = null;
        if (col<rowList.size()) {
            result = (String)rowList.get(col);
        }

        // _apparently_ it's ok to return null for a "blank" cell
        return(result);
    }


    // Support writing
    public boolean isCellEditable(int row, int col) { return true; }
    public void setValueAt(Object value, int row, int col) {
```

```
    ArrayList rowList = (ArrayList) data.get(row);

    // make this row long enough
    if (col>=rowList.size()) {
        while (col>=rowList.size()) rowList.add(null);
    }

    rowList.set(col, value);

    // notify model listeners of cell change
    fireTableCellUpdated(row, col);
}

// Adds the given column to the right hand side of the model
public void addColumn(String name) {
    names.add(name);
    fireTableStructureChanged();
    /*
     At present, TableModelListener does not have a more specific
     notification for changing the number of columns.
    */
}

// Adds an empty row, returns the new row index
public int addRow() {
    // Create a new row with nothing in it
    ArrayList row = new ArrayList();
    return(addRow(row));
}

// Adds the given row, returns the new row index
public int addRow(ArrayList row) {
    data.add(row);
    fireTableRowsInserted(data.size()-1, data.size()-1);
    return(data.size() -1);
}

// Deletes the given row
public void deleteRow(int row) {
    if (row == -1) return;

    data.remove(row);
    fireTableRowsDeleted(row, row);
}


/*
 Utility.
 Given a text line of tab-delimited strings, build
 an ArrayList of the strings.
*/
private static ArrayList stringToList(String string) {
    // Create a tokenize that uses \t as the delim, and reports
    // both the words and the delimeters.
    StringTokenizer tokenizer = new StringTokenizer(string, "\t", true);
    ArrayList row = new ArrayList();
    String elem = null;
    String last = null;
    while(tokenizer.hasMoreTokens()) {
        last = elem;
        elem = tokenizer.nextToken();
        if (!elem.equals("\t")) row.add(elem);
        else if (last.equals("\t")) row.add("");
        // We need to track the 'last' state so we can treat
```

```java
        // two tabs in a row as an empty string column.
        }
        if (elem.equals("\t")) row.add(""); // tricky: notice final element

        return(row);
    }

    /*
     Utility
     Given a collection of strings, writes them out as a line of text, separated by
tabs.
     Null strings are interpreted as a zero-length strings.
    */
    private static void writeStrings(BufferedWriter out, Collection strings
      throws IOException {
      Iterator it = strings.iterator();

      while (it.hasNext()) {
         String string = (String)it.next();
         if (string!=null) out.write(string);
         if (it.hasNext()) out.write('\t');
      }
      out.newLine();
    }




    /*
     Loads the whole model from a file.
    */
    public void loadFile(File file) {
        try {
            FileReader fileReader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(fileReader);

            // read the column names
            ArrayList first = stringToList(bufferedReader.readLine());
            names = first;

            // each line makes a row in the data model
            String line;
            data = new ArrayList();
            while ((line = bufferedReader.readLine()) != null) {
               data.add(stringToList(line));
            }

            // Send notifications that the whole table is now different
            fireTableStructureChanged();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }


    /*
     Saves the model to the given file as tab-delimited text.
    */
    public void saveToFile(File file) {
        try {
            BufferedWriter out = new BufferedWriter(new FileWriter(file));

            // write the column names
```

```
            writeStrings(out, names);

            // write all the data
            for (int i=0; i<data.size(); i++) {
                writeStrings(out, (ArrayList) data.get(i));
            }

            out.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

TableFrame.java

```
// TableFrame.java
/*
 Demonstrate a couple tables using one table model.
*/

import java.awt.*;
import javax.swing.*;
import java.util.*;

import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.filechooser.*;

class TableFrame extends JFrame {
    private BasicTableModel model;

    private JTable table;
    private JTable table2;

    JButton columnButton;
    JButton rowButton;
    JButton deleteButton;
    JButton loadButton;
    JButton saveButton;
    JComponent content;

    public TableFrame(String title) {
        super(title);
        content = (JComponent)getContentPane();
        content.setLayout(new BorderLayout(6,6));

        // Create a table model
        model = new BasicTableModel();

        // Create a table using that model
        table = new JTable(model);

        // there are many options for col resize strategy
        //table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        // JTable.AUTO_RESIZE_OFF

        // Create a scroll pane in the center, and put
        // the table in it
        JScrollPane scrollpane = new JScrollPane(table);
        scrollpane.setPreferredSize(new Dimension(300,200));
```

```
        content.add(scrollpane, BorderLayout.CENTER);

        // Create a second table using the same model, and put in the south
        JTable table2 = new JTable(model);
        scrollpane = new JScrollPane(table2);
        scrollpane.setPreferredSize(new Dimension(300,200));
        content.add(scrollpane, BorderLayout.SOUTH);

        // Create a bunch of controls in a box
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        content.add(panel, BorderLayout.EAST);

        rowButton = new JButton("Add Row");
        panel.add(rowButton);
        rowButton.addActionListener(
           new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                 int i = model.addRow();
                 table.clearSelection();
                 table.addRowSelectionInterval(i, i);
              }
           }
        );

        columnButton = new JButton("Add Column");
        panel.add(columnButton);
        columnButton.addActionListener(
           new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                 String result = JOptionPane.showInputDialog("What name for the new
column?");
                 if (result != null) {
                    model.addColumn(result);
                 }
              }
           }
        );

        deleteButton = new JButton("Delete Row");
        panel.add(deleteButton);
        deleteButton.addActionListener(
           new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                 int row = table.getSelectedRow();
                 if (row!=-1) model.deleteRow(row);
              }
           }
        );

        loadButton = new JButton("Load File");
        panel.add(loadButton);
        loadButton.addActionListener(
           new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                  //FileSystemView fsv = FileSystemView.getFileSystemView();
                 JFileChooser chooser = new JFileChooser(".");
                 int status = chooser.showOpenDialog(TableFrame.this);
                 if (status == JFileChooser.APPROVE_OPTION) {
                    model.loadFile(chooser.getSelectedFile());
                 }
              }
           }
        );
```

```
        saveButton = new JButton("Save File");
        panel.add(saveButton);
        saveButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    JFileChooser chooser = new JFileChooser(".");
                    int status = chooser.showSaveDialog(TableFrame.this);
                    if (status == JFileChooser.APPROVE_OPTION) {
                        model.saveToFile(chooser.getSelectedFile());
                    }
                }
            }
        );

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }


    static public void main(String[] args)
    {
        new TableFrame("TableFrame");
    }
}
```

MVC Summary

MVC is used in Swing in many places, and it is also a pattern you will see in
  other systems.
1. Data model -- storage
    Deals with storage. Algorithmic code can send messages to the model to get,
      modify, and write back the data
2. View -- presentation
    Gets data from the model and presents it. Translates user actions in the view
      into getters/setters sent to model. Presentation could be pixels, HTML,
      PDF, ...
3. Listener logic
    A design used in Swing: Model/view use a listener system to update the
      view(s) about changes in the model.

Advantage: Modularity

2 small problems vs. 1 big problem
Provides a natural decomposition "pattern"
    You will get used to the MVC decomposition. Other Java programmers will
      also. It ends up providing a common, understood language.
Isolate coding problems in a smaller domain
    Can solve GUI problems just in the GUI domain, the storage etc. is all quite
      separate. e.g. don't worry about file saving when implementing scrolling
      and visa-versa.

Networking / Mult Views

The abstraction between the model and view works well for a networked
  version: the model is on the central machine, the view is on the client machine.
The abstraction between the model and view can support multiple views all
  looking at one model (on one machine, or with some views over the network).

Use 50% Off The Shelf

The Model and View are both already written -- can customize one or the other
e.g. Substitute, say, your own Model, but use the off the shelf View.

e.g. File Save, or Undo

File save can be implemented/debugged just against the model. If the view
  worked before, it should still work.

undo() can just be implemented on the model -- it has to interact with far fewer lines of code than if it were implemented on top of some sort of combined model+view system

e.g. Web Site

Suppose you are implementing a calendaring web site.

model -- complex data relationships of people, times, events

view -- web pages, javascript, etc. that present parts of the model

Need to support multiple views simultaneously, and perhaps different types of view -- web page, PDF, IM message, ...

MVC: the data model team and the view team should be separate as much as possible -- don't want choices about pixels to interfere which choices of whether to store events in a hash map vs. a binary tree.

This is what the modern Servlet/JSP style does. The servlet does the data model "business logic", and the JSP just sends getter messages and formulates the results in to HTML or whatever (take CS193i).

e.g. Model Substitution

Have some 2-d data. Want to present it in a 2-d GUI. Wrap your data up so that it responds to getColumnCount(), getDataAt(), etc....

Build a JTable, passing it pointer to your object as the data model and voila. The scrolling, the GUI, etc. etc. is all done by JTable.

e.g. Wrap Database

Similar example -- suppose you have a table in an SQL database. Wrap it in TableModel class that makes the data appear to be in row/col format. getValueAt() requests are translated into queries on the database. Note that the JTable is insulated from knowing how you get the data, so long as you respond to the TableModel messages -- that's a nice use of OOP modularity.

Danger: Listener Storm

Suppose you have objects A, B, and C

Suppose they are listening to each other

Can get a sort of infinite loop where A changes and notifies B, which changes as a result and notifies C, which changes as a result, and notifies A, which ...

Solution #1: on change notification, do the data = model.getData(), but then check if the data value is actually different from the old value. Only notify if the value is actually different. This solves some cases.

Solution #2: have a "isupdating" boolean. Set it to true while making a change and doing notifications. Ignore notifications that come in while isupdating==true.