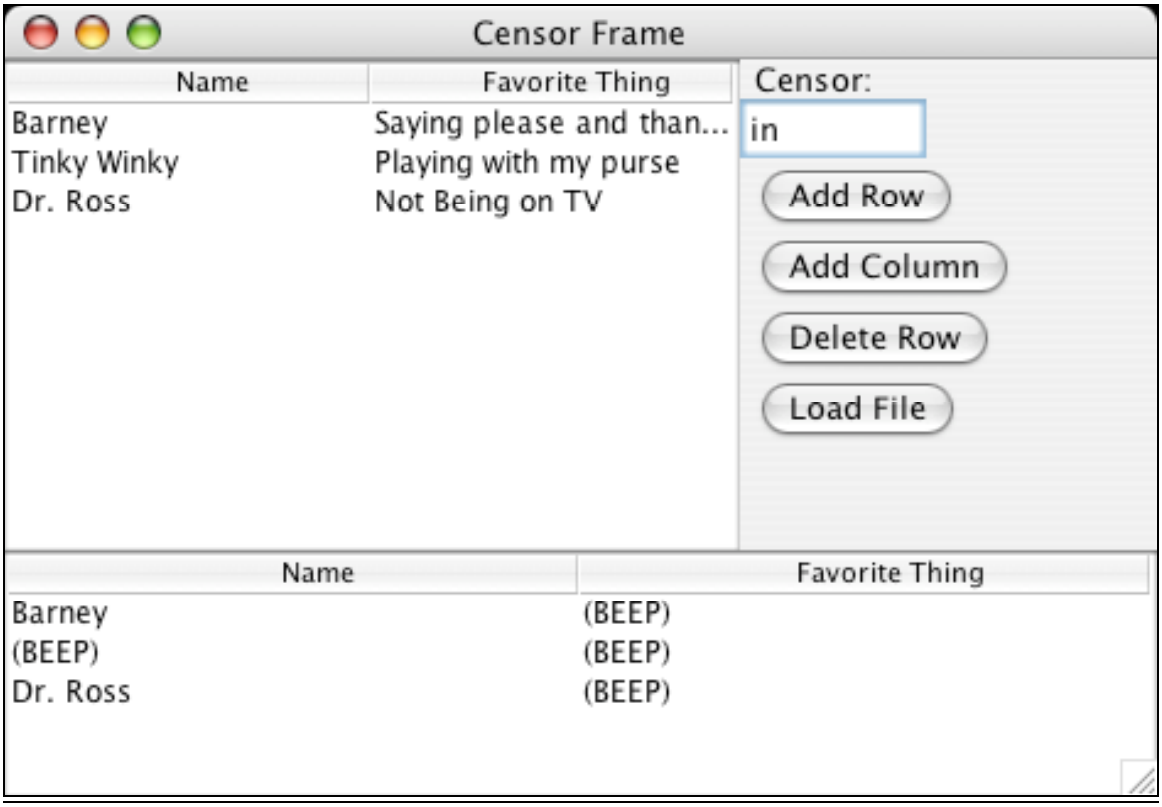
MVC Table / Delegation

Delegate MVC Table Censor Example

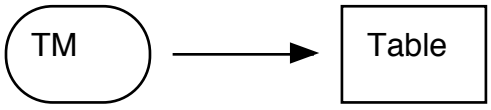
- Uses the delegate strategy to build a variant table model
- Uses table model listener logic in a complex way -- a good test of your understanding of MVC/listener design



Censor Table Example

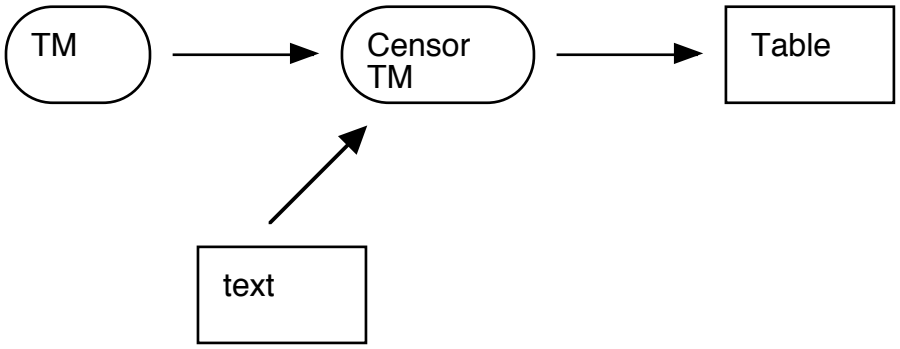
- Implement a TableModel that censors cells
- Use a BasicTableModel delegate to take care of the storage

1. Plain



- Data and data change notifications go from TM to table

2. "Delegate" Solution



Censor Strategy, Delegation

- Real model
  1. Have a pointer to the "real" TableModel that has the real data -- the delegate
- "delegate" strategy
  - delegate HASA vs. inheritance ISA
  - Do not store any data -- let the model delegate do that
  - "Pass back" requests you get to the model delegate (with a little adjustment)
- getValueAt()
  - Pass back to the real model, taking the censoring into account
- Listen for notifications from the real model
  - If it says something is changed, pass that notification forward.
  - e.g. if the real model says that cell (0,1) has changed, tell your client that cell (0,1) has changed
- censor text change -> fireTableCellUpdated(r, c)
  - When the censor text changes, iterate over the whole model, and generate cellUpdated(r,c) notifications for cells that are now different

CensorTableModel Code Points

- 1. ctor() -- register as listener to the real TM
- 2. getRowCount(), getValueAt() -- pass back to the real table model, do the filtering on the results
- 3. tableChanged() (notification from the real table model) -- pass the notification forward to our view
- 4. setCensor(String newCensor) -- sent by the GUI when the user changes the censor string. We need to recompute which cells are censored and which are not and fireXXX to update the GUI.

CensorTableModel.java

```
// CensorTableModel.java
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;


/*
 Presents a table model that censors out table cells that include
 the "censor" string.

 This is a "layered" or "delegate" solution.
 We do not store any cell data at all --
 we store a pointer to a delegate "real" table model that has the real data.
 In typical delegation fashion, when we receive a message to do something,
 we often pass the request back to the delegate for processing.
 We pass the data forward, but do the censoring on it.
 We listen to the delegate to hear about table model changes from it.
 We get setCensor() when the censor string changes, and in that case
 we have to re-think the censoring for every cell.
 This is a pretty complex use of MVC model-listener logic.
```

```
 Note: another solution would be to subclass off TableModel to create a censoring
 variant. The delegate strategy shown here is probably easier to get right.
*/
class CensorTableModel extends AbstractTableModel implements TableModelListener {
    private TableModel model;     // the "real" model
    private String censor;    // current censor string

    /*
     Sets up the censoring model based on the given model.
    */
    public CensorTableModel(TableModel model) {
        this.model = model;
        censor = "";

        // we listen to the real model for its changes
        model.addTableModelListener(this);
    }


    /*
     Standard table model messages -- we just pass them back
     to our delegate, the real table model.
     Except, getData, which does the censoring.
    */
    public String getColumnName(int col) { return model.getColumnName(col); }
    public int getColumnCount() { return model.getColumnCount(); }
    public int getRowCount() { return model.getRowCount(); }

    // This one is interesting: calls the delegate, then adjusts
    // the data.
    public Object getValueAt(int row, int col) {
        String elem = (String) model.getValueAt(row, col);
        if (isCensored(elem, censor)) return "(BEEP)";
        else return elem;
    }

    // We don't allow a censored cell to be edited
    public boolean isCellEditable(int row, int col) { return false; }
    // So we don't have to respond to the following...
    //public void setValueAt(Object value, int row, int col) {
    //  model.setValueAt(value, row, col);
    //}


    /*
     We get this notification when the real table model
     changes: column added, cell updated, etc.
     We just pass it along to our table, and it
     will generate getXXX() requests as it sees fit.
    */
    public void tableChanged(TableModelEvent e) {
        fireTableChanged(e);
    }


    /*
     Changes the censor string.

     Implementation: In this case, we need to
     check if any of the cells are now censored differently,
     and do fireTableCellUpdated() to notify
     the table of any cells that are changed.
     Compares what each cell looked like with the
     old censor string vs. the new censor string to
     decide if we need to tell our table to change or not.
    */
    public void setCensor(String newCensor) {
        String oldCensor = censor;
        censor = newCensor.toLowerCase();
```

```
        // if the strings are the same, just forget it
        if (censor.equals(oldCensor)) return;

        // look at every cell, compare old censoring to new
        for (int row = 0; row<model.getRowCount(); row++) {
            for (int col = 0; col<model.getColumnCount(); col++) {
                String elem = (String) model.getValueAt(row, col);
                if (elem != null) {
                    boolean old = isCensored(elem, oldCensor);
                    boolean now = isCensored(elem, censor);

                    // tell the table the cell changed if the
                    // new state is different from the old state.
                    if (old != now) fireTableCellUpdated(row, col);
                }
            }
        }
    }


    /*
     Private utility.
     Tests a cell string against a censor string to
     see if it should be censored.
     For censoring to happen, the censor string must
     not be the empty string.

     note: example of "bottlneck" strategy -- in this way,
     getValueAt() and setCensor() have *exactly the same*
     notion of censoring. Two methods like that getting
     out of sync for some edge case is a classic source of bugs.
    */
    private boolean isCensored(String cell, String censorString) {
        return(cell!=null && !"".equals(censorString) &&
            cell.toLowerCase().indexOf(censorString) != -1);
    }
}
```

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. Controller/change logic
  - Manage change between the model and view. Swing uses a listener structure to notify the view of model changes.

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 simultaneously showing one model (on one machine, or with some views over the network).

Advangage: Pluggable

- The Model and View are both already written -- can use either off the shelf, connected to a custom class.
- 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. (Perhaps the definition of modularity should be that X and Y classes are modular if you can add a feature to X, confident that your change will not disturb Y.)
- 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 mixed 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.

e.g. Model Substitution

- Have some 2-column list of dates and numbers to display. Want to present it in a 2-d GUI. Wrap your data up so that it responds to getColumnCount(), getValueAt(), etc....
- Build an off-the-shelf 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 Notification 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 valueInModel = newValue step, but then check if the new value is actually different from the old value. Only notify if the value is actually different. This solves some cases.

- Solution #2: have a "ignoreUpdates" boolean. Set it to true temporarily while making a change and doing notifications. Ignore notifications that come in while ignoreUpdates is true. Then set it to false after sendNotifications() returns.

Danger: Listener Inefficiency

- Suppose we are changing 100 rows in a TableModel -- could loop through, sending 100x setter messages.
- Each will trigger the whole fireXXX/getData cycle against all the views
- Might well be better to make all 100x changes to the model, and then do a single fireXXX to signal the changes all at once. This is, essentially, a manual form of coalescing.
- Could use a "combineUpdates" boolean in the model to bracket all the changes, and send the single coalesced notification on the combineUpdates=false transition.
- Could use a realtime clock, coalescing all updates that come in during, say, a 50 millisecond period.