Chapter 12

WORKING WITH TABLE VIEWS

CHAPTER 12 WORKING WITH TABLE VIEWS

Goal

To study the common techniques for using an NSTableView including the design of a custom data source.

Prerequisites

An understanding of delegation and notification.

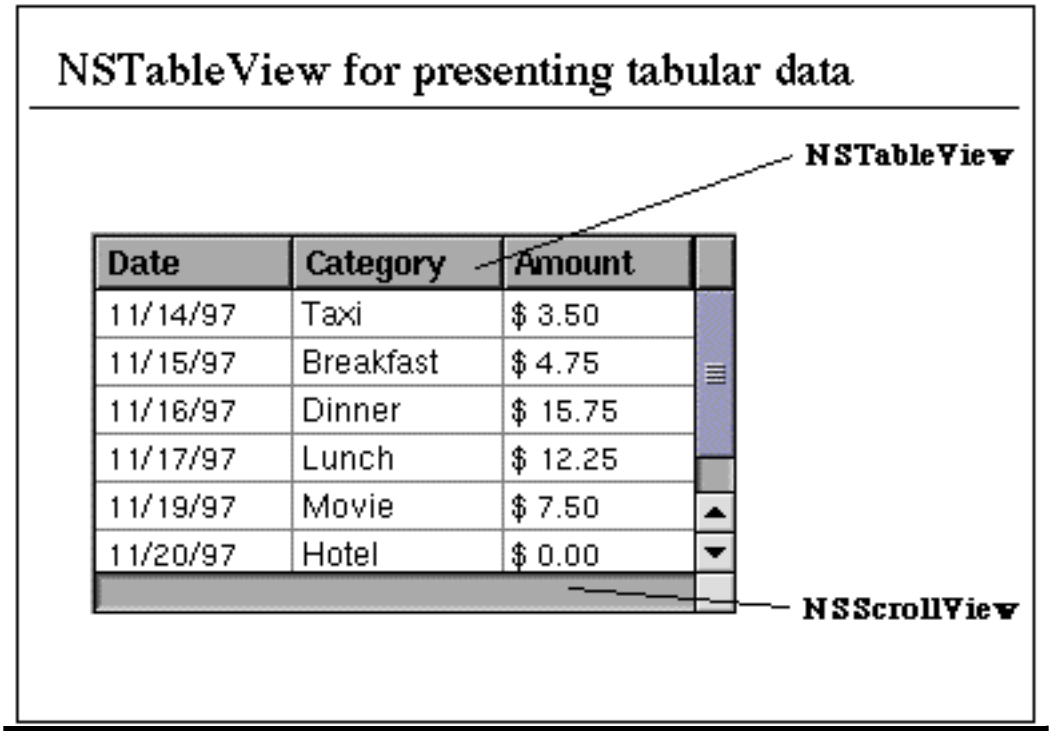
Objectives

At the end of this section, you will be able to:

- Write a custom data source object that works with a multi-column table view
- Make use of common NSTableView methods for reloading, selecting and tracking changes

Reading

NSTableView class reference in the Application Kit

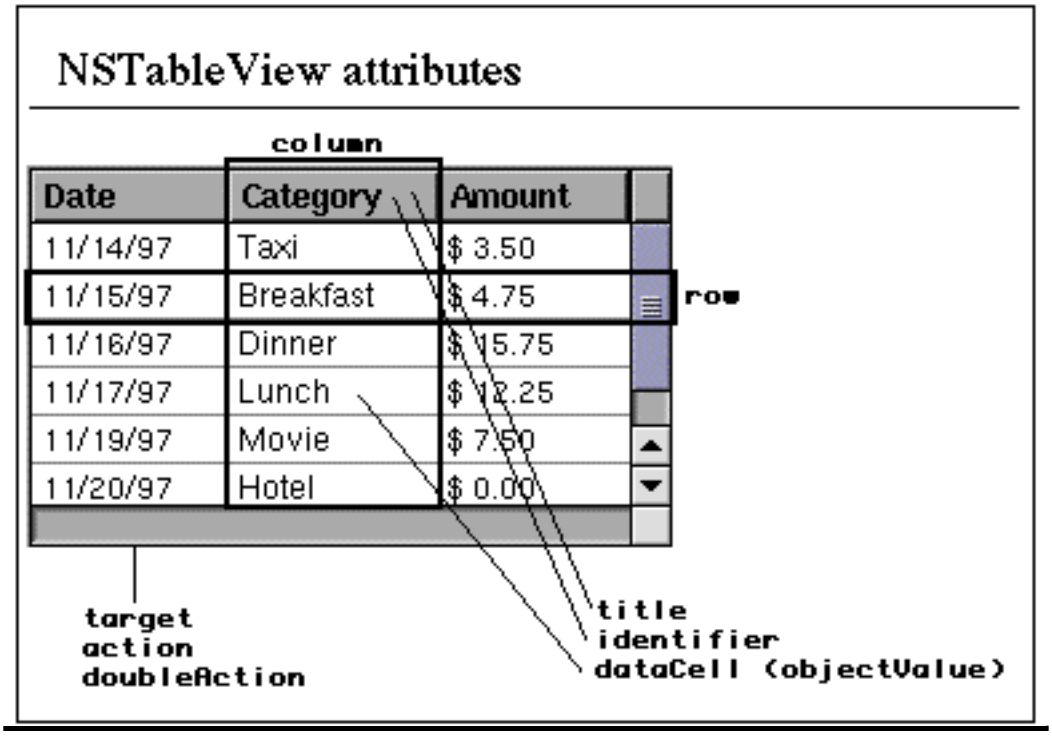


NSTableView for presenting tabular data

An NSTableView displays a set of related records, typically business objects. A row maps to a single object. Each column maps to an attribute of the object for that row. Most often, the number of rows and or columns is variable and exceeds the dimensions of NSTableView real estate so NSTableViews are usually surrounded by an instance of NSScrollView. Take note of this. When making connections in Interface Builder, be sure you are pointing to the table view, not the ScrollView.

The user selects rows or columns in the table by clicking and edits individual cells by double-clicking. The user can rearrange columns by dragging the column headers and can resize the columns by dragging the divider between headers. You can configure the table in various ways programmatically or using Interface Builder. A table view can be configured for multiple selection or no selection, or prohibit the user from editing or rearranging particular columns, and so on.

Though designed for multiple columns, NSTableView is convenient even for a single column list of items. In either case, the programmatic interface is the same.



NSTableView attributes

Some basic vocabulary is useful when dealing with NSTableViews:

» **row**—a row usually maps to one object in your data model.

» **column**—a column maps to an attribute of the object for that row. Some columns may hold derived or calculated values. Often, only a subset of the object's attributes appear in the table view. Each column sports a handful of configurable attributes:

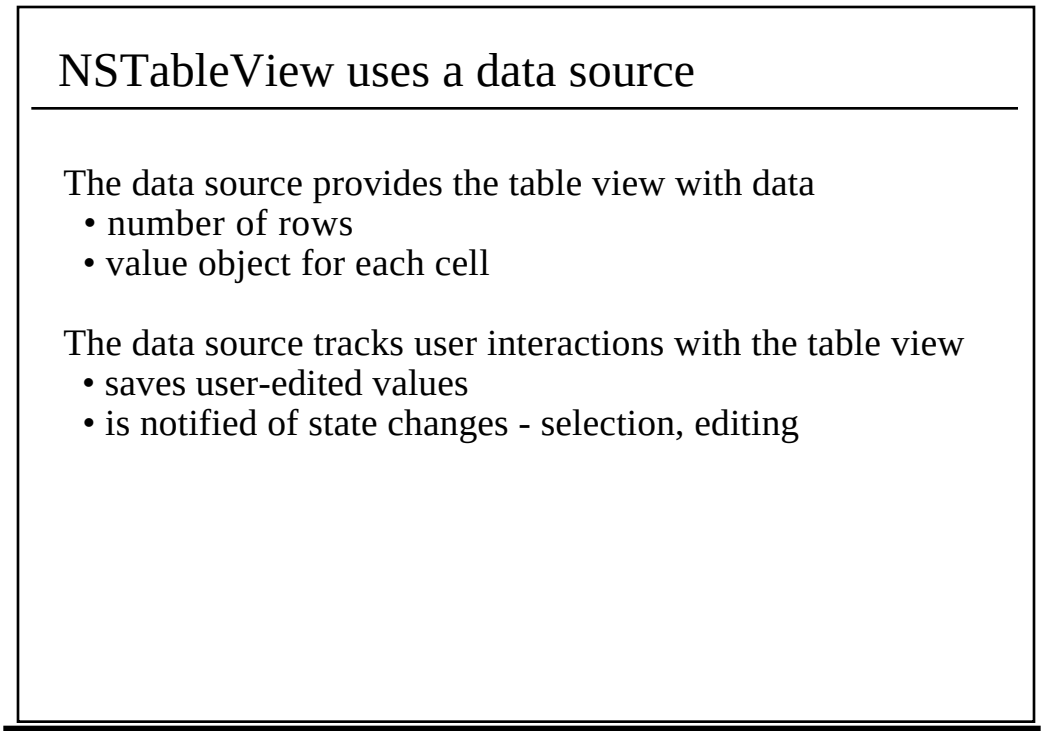
title—the column heading in the table.

identifier—an object value, most often a string, that is used to map a column to an attribute in the object. It might be the attribute name o a number like a tag.

dataCell—a single object value within a table view. It is possible to configure a column to use a custom cell if your application needs to use one. By default, it uses NSTextFieldCell.

formatter—like a text fields, a table view column can use a formatter. The formatter automatically applies to all cells in the column.

Like most user interface elements, NSTableView is a subclass of NSControl. As such, it supports a target/action connection. In addition, you can set a **doubleAction** which will be sent when the user double-clicks on anything other than an editable cell. This cannot be set in Interface Builder: use **setDoubleAction:**(SEL)action.



NSTableView uses a data source

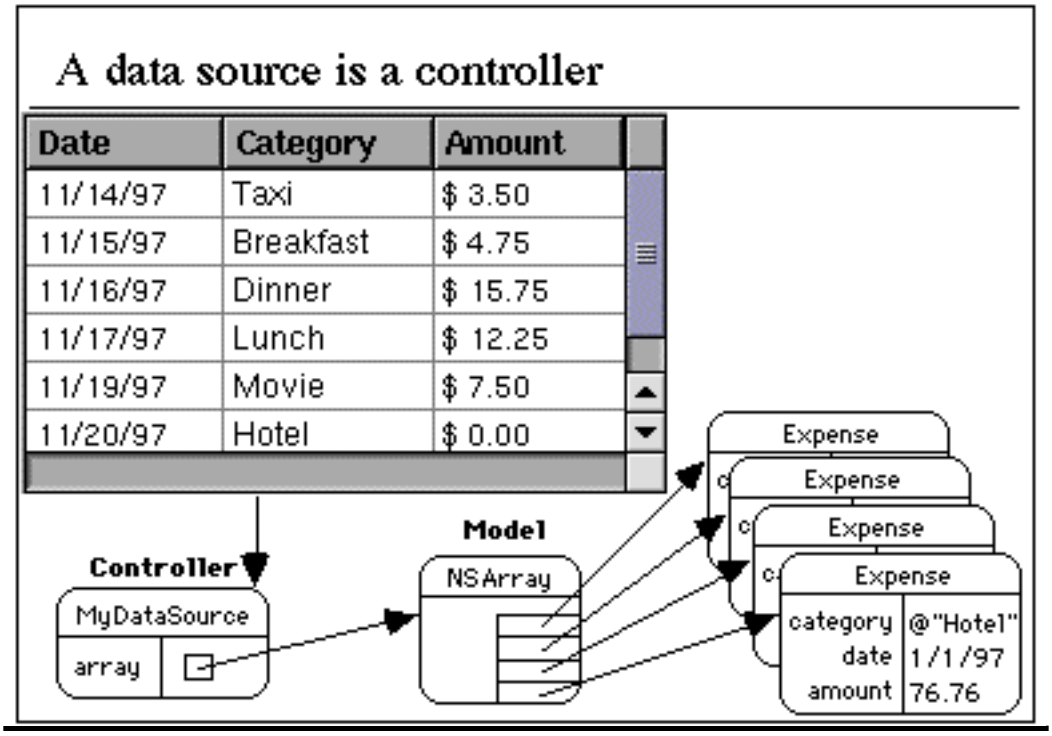
Unlike more simple NSControls, NSTableView doesn't store the data it displays. Instead, it cooperates with a custom helper object that you provide—its data source.

Your data source can manage its data in any way, but it must be able to map data instances to an integer index—a row—and must implement methods to provide the following:

» How many records the data source contains—rows.

» What the value is for a particular record's attribute—columns.

» A method for changing a value of an attribute in the data source. If your table view is configured to allow editing, changes must be saved back to the data source.

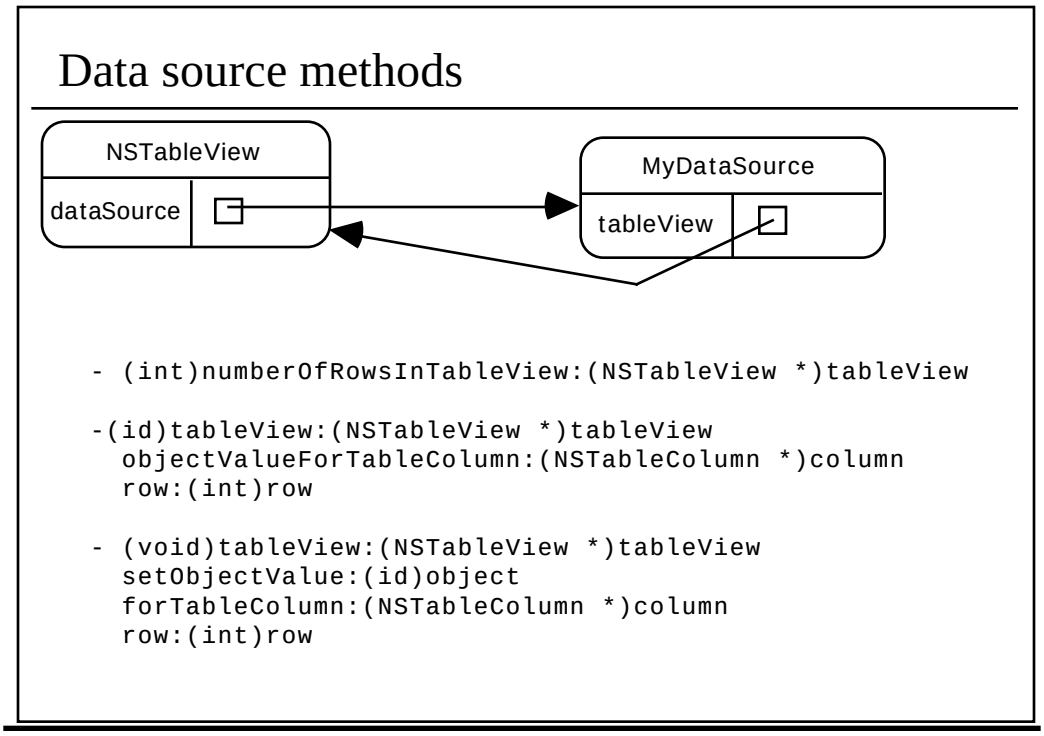


A data source is a controller

This is another example of the Model-View- Controller pattern:

- model—the business objects being displayed
- view—the NSTableView user interface object
- controller—your custom data source

Typically, a data source points to an array of business objects. This is the true source of data used to fill the NSTableView cells.



Data source methods

The NSTableView sends several messages to access the contents of its data source. These methods are implemented by you in your data source:

- **numberOfRowsInTableView:**—return the number of rows to display
- **tableView:objectValueForTableColumn:**—return a single value object corresponding to a specific row/column pair. Cells are filled lazily, one at a time, and only if they are actually visible.
- **tableView:setObjectValue:forTablecolumn:row:**—works in the reverse direction—the table view is passing the data source a user-modified value from the user interface. It needs to be written back to the data source.

The data source must implement the first two methods to work properly with a table view. If the table view is read-only, you don't need the third.

Getting the number of rows

```
- (int)numberOfRowsInTableView:(NSTableView *)tableView
{
    return [expenseArray count];
}
```

Getting the number of rows

When a table view is told to display itself, it is sent a **reloadData** message, it first asks the data source how many items will be displayed. The data source consults its model, such as an NSArray of objects, and returns the **count**.

Getting a value

```
- (id)tableView:(NSTableView *)tableView
  objectValueForTableColumn:(NSTableColumn *)column
  row:(int)row
{
    Expense *expense;
    NSString *identifier;

    expense = [expenseArray objectAtIndex:row];
    identifier = [column identfier];
    if ([identifier isEqualToString:@"Category"])
        return [expense category];
    else
        return [expense date];
}
```

Getting a value

Equipped with the proper number of rows, the table view now asks the data source for individual cell values for the rows that are currently visible. As the user scrolls around, new cells come into view and need to have their values fetched from the data source. Each time the table view messages the data source with **tableView:objectValueForTableColumn:row:**.

The table view is expecting a dynamically typed object as a return value. What kind of object is appropriate? Just like a text field, the object can be any value-type object that responds to the **description** method. Ultimately, the table view needs a string value to display in the cell. ObjectValues are typically NSStrings, NSCalendarDates, NSNumbers or custom value objects of your own design.

How do you know which attribute goes with which column? Remember that the user can rearrange the columns so the column number is not helpful. This is where the **identifier** is used. This is an object value that you choose to be helpful in this exact situation. You can set a string value in Interface Builder. You can programmatically set any value object you wish. In this example, a string value is used—a tag that names the attribute in the business object.

Saving a change (optional)

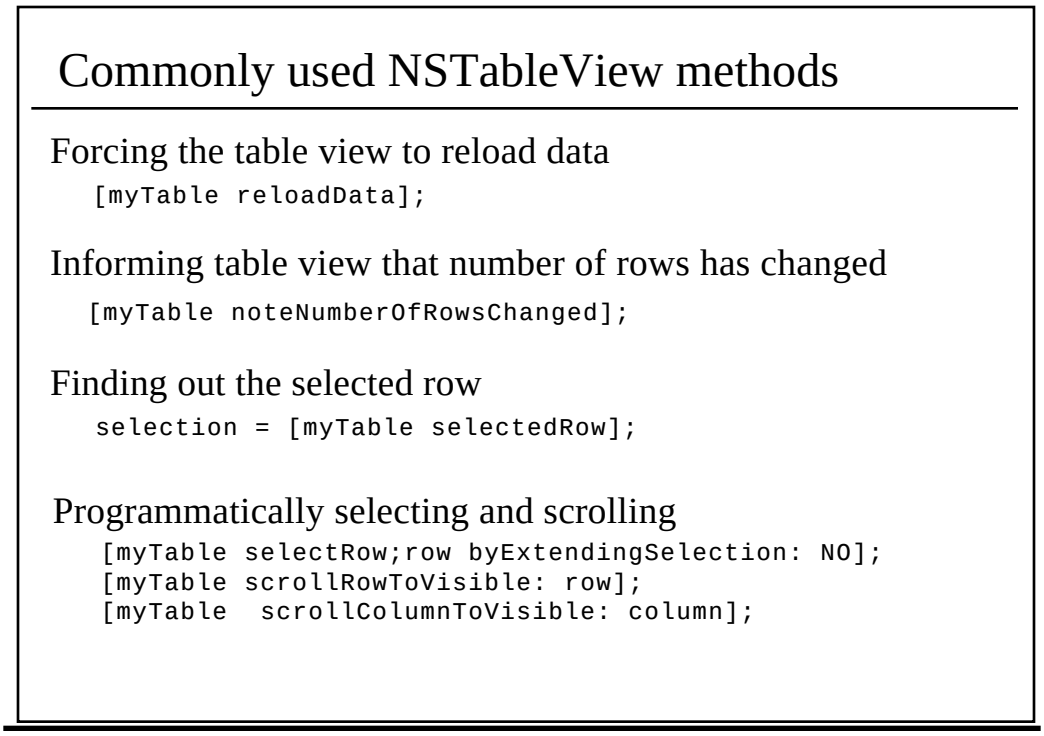
```
- (void)tableView:(NSTableView *)tableView
  setObjectValue:(id)object
  forTableColumn:(NSTableColumn *)column
  row:(int)row
{
    Expense *expense;
    NSString *identifier;

    expense = [expenseArray objectAtIndex:row];
    identifier = [column identifier];
    if ([identifier isEqualToString:@"Category"])
        [expense setCategory:object];
    else
        [expense setDate:object];
}
```

Saving a change (optional)

If the user edits a value in the table, the table view informs the data source so that the table view and the data model are kept in sync. If your table view is read-only, you are not required to implement this.

Your document component design should track changes to the data so that attempts to close without saving generate warnings. This is an ideal place to do it. But this is not the only change typical of a table view. Your interface might allow the user to add or delete a row. Such actions would be possible through other controls on the window that contact your data source or controller object directly. Inside each control's action message, you track the modification for future use.

So how would you add or delete a row dynamically? Change the data model—add or remove an object from the NSArray—then tell the table view something has changed. Ask the table view it to reload itself with **reloadData**.



Commonly used NSTableView methods

You have learned the methods that the table view uses in your data source object. Messages will flow in the opposite direction too. The data source needs to message the table view to bootstrap data loading, inform it of changes in the data, programmatically select rows and cells and a variety of other possibilities.

Here are some of the most commonly used NSTableView methods:

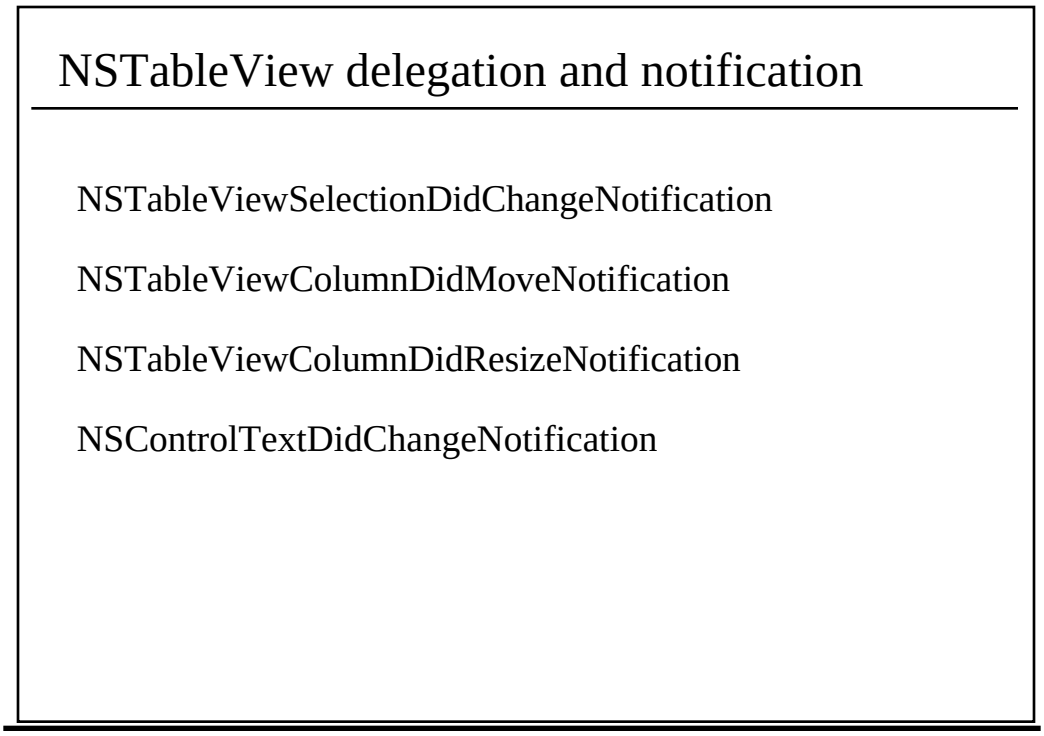
» **reloadData**—forces the table view to reload from the data source.

» **noteNumberOfRowsChanged**—informs the table view that the size of the data source—the number of objects—has changed.

» **selectedRow**—returns the index of the currently selected row.

» **selectRow:byExtendingSelection:**—programmatically selects a row, with a provision for either appending to the currently seleted rows or clearing the previous selection.

» **editColumn: row: withEvent: select:**—like NSTextField's **selectText** this allows you to put an individual cell into edit mode programmatically.



NSTableView delegation and notification

Your data source can closely follow the user's interaction with the table view by registering as an observer for these and other notifications. Note, unlike some objects, attaching yourself as a data source or delegate does not automatically register you for all notifications.

It is likely whenever a user modifies the object value of a table view cell, your document should be marked as edited in preparation for saving it when the time is right. You could track such changes in two different ways:

- NSControlTextDidChangeNotification
- **tableView:setObjectValueForTableColumn:row:**

Your data source must implement the latter if you expect to save the user's changes to the document. The former might be a useful addition depending on your design.

> **Note:** NSTableViewSelectionDidChangeNotification is posted when the selected row changes, not necessarily the column. You can think of it as the selected object changed since one row typically maps to one object in your data source model.

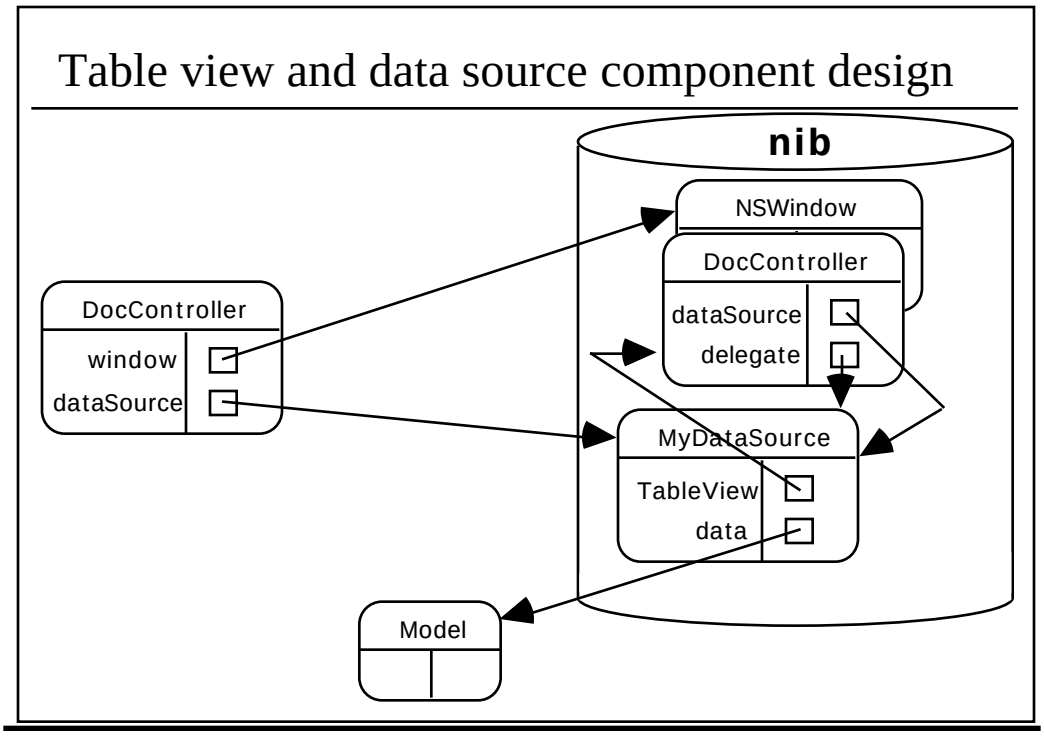


Table view and data source component design

This approach separates the document controller from the data source since the former controls the document as a whole while the latter controls one of possibly many views on the document window. The document controller is still File's Owner and is created outside the nib file. It manages the common reusable aspects of a document controller—loading the nib, tracking modificatn status, handling window delegation, and dealing with Open and Save panels and pathnames.

The data source, on the other hand, is instantiated inside the nib file since it is tightly coupled with the table view itself. It is the controller in a Model-View-Controller trio with the table view (view) and the data (model). Its job is to synchronize the view with the model. It is possible to imagine some general and reusable designs for the data source itself, such as a data source that takes a simple array of objects.

Some amount of communication must happen between the document controller and the data source—the document controller will likely pass a filename or even the unarchived data model to the data source. The data source must inform the controller that the data has been modified; the document should be considered edited.

An alternate design might combine the document controller and table view data source into a single object. While this is more simple, in the long run it is a less flexible design.

Important ideas from this section

» NSTableViews rely on their data source for:

number of rows

objectValues for each cell

saving user changes back to the model

» A data source is much like a delegate, a "helper" for customizing the table view for your application.

» A data source is required to implement the following:

numberOfrowsInTableView

tableView:objectValueforTablecolumn:

» NSTableView posts a number of useful notifications. You must explicitly register your observer to get them. This would typically be your data source object.

Class featured in this section

NSTableView

REVIEW WORKING WITH TABLE VIEWS

1. Why should table views use data sources?

2. In what way is a Table View radically different from a NSControl?

3. How do you know which attribute in your data object maps to which column in a table view?

4. Why might a table view with a single column be useful?

EXERCISE 12.1 EXPENSE REPORT—ADDING A TABLE VIEW TO THE DOCUMENT

In this exercise you evolve the multiple-document template application into a simple expense report application. Expenses are listed in a scrollable table view. The expense report is built around a simple business model object, Expense, which is fully implemented for you as part of the exercise materials. The data for the expense report is an array of Expense objects. Your custom table view data source will map the data into the user interface and reflect user edits back into the data.

Objectives

After completing this exercise, you will be able to:

- Display tabular data using NSTableView
- Write a data source for a table view
- Modify tabular data via NSTableView and a data source

Exercise—Stage 1

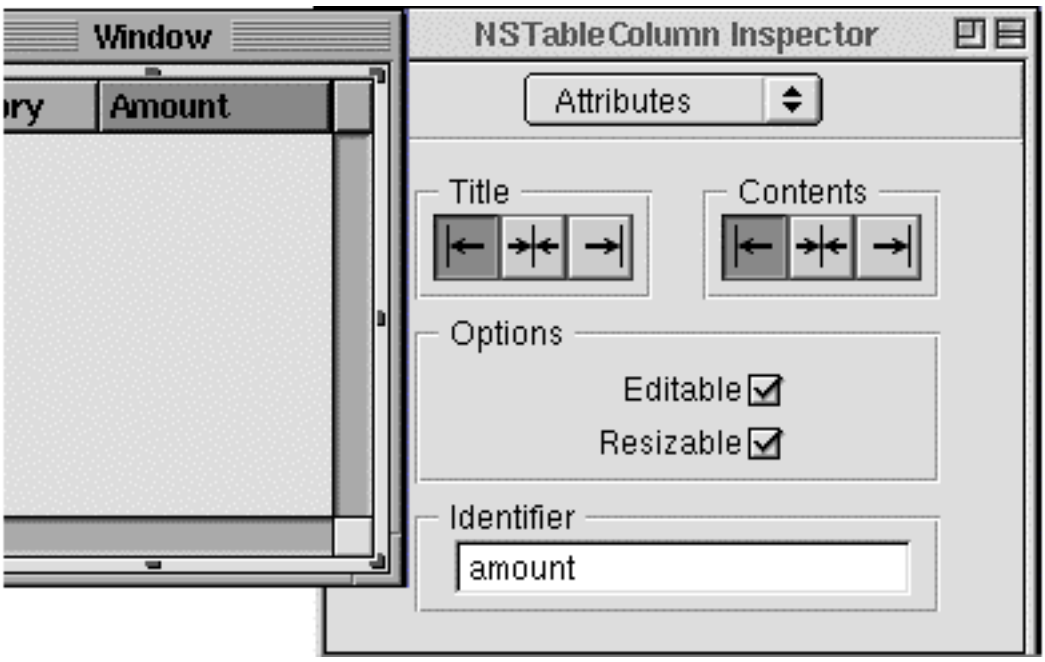
1. Make a copy of the previous project to keep a reference copy. Open the new project and change the project name to **Expenses**, in order to distinguish it from earlier versions. Save it.

2. You will need the Expense object and the skeleton table view data source. These files are all in the **ExerciseMaterials** area—**Expense.h**, **Expense.m**, **ExpenseDataSource.h** and **ExpenseDataSource.m**.

    - Drag these files into the Classes suitcase of your Documents subproject.
    - Study **ExpenseDataSource.m** carefully to understand some of the support implemented for you. The **init** method builds a dummy array of Expense objects so you have some initial test data to work with. In future exercises, you will evolve the application to open a previously saved file or bring up a new, empty expense report.

3. Open the Document nib file in Interface Builder and modify the interface:

    - Delete the old modify button.
    - Add a table view from the palette. It should have three columns. You can add columns by selecting one, copying and pasting it.Change the column titles of the table view to be **Date**, **Category** and **Amount**.



    - For each column, set the **identifier** attribute using the inspector to the corresponding Expense object attribute names—**date**, **category** and **amount**. This allows the columns of the table view to be identified by something other than their titles.

Hint: These identifiers are used in the data source methods. They can be used to generate selectors that communicate directly with Expense objects. Other mappings are possible.

4. Next, you provide the data source for the table view and connect it to the other objects in the nib.

    - Drag **ExpenseDataSource.h** into Interface Builder's class browser so you can use the class. Create an instance.
    - Connect the table view to the ExpenseDataSource instance which is both its delegate and its data source. Be sure to connect from the table view and not the scroll view which surrounds it.
    - Connect the ExpenseDataSource **tableView** outlet to the table view.
    - Implement the two required data source methods used to fill the table view. The data source has a instance variable for the array of Expense objects called **array**. Table view rows correspond to array indicies. You will have to map a column identifier to an instance variable in the correct Expense object instance. You can use the identifier to generate the accessor method selector. Your data source already implements the method **accessorFromIdentifier**. Use it like so:

```
SEL sel = [self accessorFromIdentifer: identifier];

id value = [expenseInstance performSelector:sel]
```

- The data source needs to load the table view in **awakeFromNib**.

5. Make sure everything is saved and then build the project. At this stage you have an application that should display the test data when you open a new document. You cannot change any of the data yet. Stage 2 adds this functionality.

Stage 2

1. The next part of the exercise is to allow changes to the underlying data. To do this, the data source must be able to accept changes from the table view, sent via the **-tableView:setObjectValue:forTableColumn:row: method**.

    - You can make use of the existing **setAccessorFromIdentifier:** method to obtain the required method selector or just do a simple hard-coded comparison using the column identifier.

```
SEL sel = [self setAccessorFromIdentifer: identifier];

[expenseInstance performSelector:sel withObject:value];
```

Hint: When the user edits anything in the table, they are changing the document. You should include code to notify the document of this change. This can be done either by messaging the document window's delegate, which you can find via the table view, or by updating the window's **isDocumentEdited** flag directly.

2. Rebuild the project and see if you can edit items. Does the document window indicate that the document is modified? If you had implemented it earlier, are you still warned if you try to close a modified document?

Stage 3

1. The last stage is to make your expense report extensible. The picture above shows the required user interface controls—**Add** and **Delete** buttons.

2. Make the necessary changes to your ExpenseDataSource, including the addition of the two action methods:

   - **-add:**—create a new Expense object and add it to the **array** instance variable. You should mark the document as modified, as you did earlier when the user edited a field.
   - **-delete:**—delete the currently selected object from the **array** instance variable. You can determine which row is selected by asking the **tableView**. The document status also needs updating in this method.
   - After any change in the number of rows, you should tell the table view to **reloadData** so that the changes are reflected in the display.

3. Make the changes to the interface:

   - Drag the new **ExpenseDataSource.h** file into Interface Builder to get the new outlets and actions.
   - Position the Add and Delete buttons on the document window.
   - Drag connections from the two buttons to the appropriate actions in the ExpenseDataSource.

4. Save everything and build the project. Check that you can add and delete expense items.

Enhancements

» The delete button is always enabled. What if there are no rows? What if no row is currently selected? You will need an outlet to the Delete button so you can toggle its enabled status. See the table view delegate method **tableViewSelectionDidChange:** for more information.

» When you add a new row with the Add button, it should be automatically selected and scrolled into view.

» Add a total field to the interface and implement the logic in the data source to keep it up-to-date.