New DataSet Widget
User's Guide & Cookbook

The purpose of this document is to give you a comprehensive overview of the features and facilities provided by *New DataSet Widgets* and how to access them programmatically. It starts with an introductory chapter. Subsequently step-by-step instructions show how to build applications employing new data set widgets.

Contents

Introduction

New DataSet Widget is a re-implementation of the existing data set widget as being shipped with VisualWorks. Originally, the sole reason for this re-implementation was the need for multiple selection facilities in data set views. Unlike the data set implementation shipped with VisualWorks this new implementation is based on a *View* class derived from *MultiSelection-SequenceView*, thus inheriting support for multiple selections. However, *New DataSet Widget* not only supports multiple selection of entries but also conforms to the method interface of the old implementation. Thus, the new data set widgets can …

a) be created and configured in Interface Painter as before, and
b) be used in place of any old data set.

In the following sections we will discuss the enhanced features provided by this new implementation, when compared to the old one. We will also have a look at how this affects the user interface and the programming interface of data set views.

New features compared to old implementation

Browsing and Editing mode

An application using a new data set view can switch the widget between browsing behavior and editing behavior at runtime. In browse mode, the widget behaves very much like an ordinary *SequenceView*, except for the tabular layout, the column labels and the resizable columns. If multiple selection facility is enabled, you can select more than one row in this mode.

In edit mode, there's always only one cell selected and active in the data set view. You can edit the text in this cell. As in the old implementation, you can exclude individual columns from editing by making them read-only.

Manipulating appearance

NewDataSet widgets provide manifold ways to manipulate their appearance at runtime of an application. In particular, an application can arrange for enabling or disabling …

- horizontal and vertical grid lines
- column labels
- row selectors

- decorated cell editors (vs. plain editors without borders)
- column dragging
- sorting entries by columns

The last two issues will be discussed in detail in the following sections.

Sorting entries

A user can sort the entries in a new data set view by clicking on a column's label. This facility works in a general and adaptive way, without requiring any explicit actions from the application programmer. It simply uses the column values retrieved by the column aspect as being configured in the interface painter.

If the values retrieved for a column respond to the comparison operator message '<', this operator is used to provide a type specific sorting. If this message is not supported directly, the entries' display strings are used. E.g. if a column displays date values, the entries will be correctly sorted by this column, since class Date implements '<'. This wouldn't be the case if display strings would be used for dates.

Changing the column layout

Besides column resizing, new data set also supports interactive modification of the column order by clicking on a column label and dragging it to the new position. This is possible for non-frozen columns only. The column layout (column widths and order) can be stored when a window is closed and restored on opening a window containing a new data set.

Automatic appending of new entries

New data set widgets can be configured to automatically insert a new entry when a user moves the cell editor past the last edited entry. Application programmers initialize a new data set view with a so called „auto new block“ which is responsible for returning an entry to be used as the template.

Support for Drag & Drop

New data set view inherits support for drag & drop from its superclass *SequenceView*. A developer can follow the steps described in the VisualWorks Cookbook to enable drag & drop for new data set views.

User Interface

The look and feel of new data set views differs from that of the old implementation. It adopts to that of table controls in Windows 95 or NT 4.0. This has the result of presenting the end user with a more familiar user interface. Part of this are some new keyboard shortcuts supported.

Sorting entries

A user can sort the entries in a new data set view by clicking on a column's label. The first click on a label sorts the entries by the contents of the selected column in an ascending order. Subsequent clicks on the same column label reverses the sort order. An icon is displayed in the column label to indicate the sort order.

Column dragging

A user can reposition a column by clicking on a column label and dragging it to the new position. Clicking and releasing the mouse button without moving the mouse invokes the sorting function. When the mouse is moved horizontally a short distance, the mode changes to column dragging. This is indicated by the column label following the mouse pointer visually.

In the process of dragging the column label horizontally the labels' order will change gradually as the label is moved beyond the position of the other columns. The label can be moved a vertically within a certain 'corridor'. If the vertical distance exceeds a certain amount, however, the label following the mouse will disappear, indicating that the dragging operation will be cancelled. The label snaps back to its old position.

Appending new entries
(AutoAppend)

New data set widgets can be configured to automatically insert a new entry when a user moves the cell editor past the last edited entry. This feature works like comparable facilities, e.g. that in MS Access table controls: A template entry is always appended as the last entry. If row selectors are enabled, this entry is marked with a special asterisk icon ('*').

When the user edits the template entry and subsequently moves the editor to the next entry using Tab or Return, a new template entry is automatically appended. However, no new templates are added if no changes are made to an already existing template entry. The editor won't move past an unchanged template entry.

Keyboard Shortcuts

Keyboard shortcuts available in editing and browsing mode:

- **Up / Down** moves the selection to the previous / next entry
- **Page Up / Page Down** moves the selection one page up / down

Keyboard shortcuts available in browsing mode only:

- **Home / End** moves the selection to the first/last entry in the list. (In editing mode these keys move the cursor to the begin/end of the edited text.)

Keyboard shortcuts available in editing mode only:

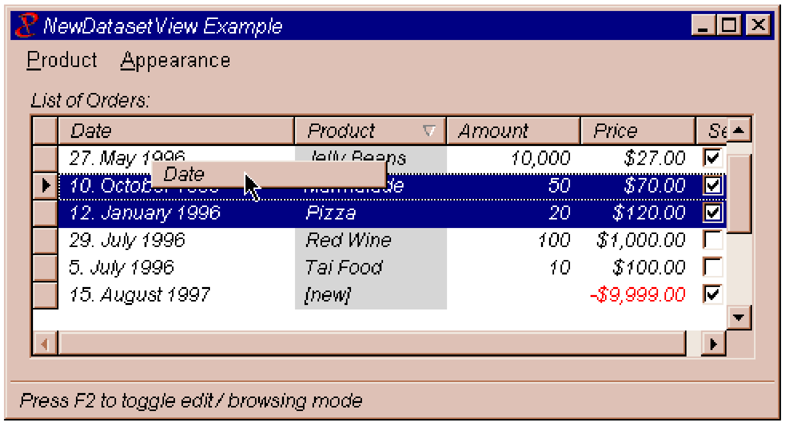
- **Tab / Shift Tab** moves the editor to the next/previous cell.
- **Return / Enter** moves the editor to the next cell.
- **Ctrl-Tab / Ctrl-Shift-Tab** moves keyboard focus to the next/previous widget.

Differences in appearance

The appearance of new data set views is quite different from that of old ones. It is more closer to table controls in Windows 95. The most obvious differences in appearance are:

- Button-like row selector and column labels.
- Optional use of non-decorated (without border) cell editors.

Except for the tabular layout, the entries in a new data set widget are displayed just as they would be displayed in a list box. Together with the browsing mode supported by new data set views, this gives it a more 'entry-oriented' character than the cell-oriented character of the old appearance. The following picture shows a new data set view in browsing mode.



Date	Product	Amount	Price	Se
27. May 1996	Jelly Beans	10,000	$27.00	✓
10. Octob	de	50	$70.00	✓
12. January 1996	Pizza	20	$120.00	✓
29. July 1996	Red Wine	100	$1,000.00	☐
5. July 1996	Tai Food	10	$100.00	☐
15. August 1997	[new]		-$9,999.00	✓

Press F2 to toggle edit / browsing mode

Programming Interface

From a programmer's point of view, a new data set view behaves very much like a SequenceView. In fact class NewDataSetView is derived from SequenceView and inherits most of its functionality — and hence its programming interface — from its superclass. Moreover, the very most of the messages of the old implementation's interface is supported too. This allows a developer to replace an old data set view by a new one without changing code, that was already written to access the widget.

New messages are provided for additional features, that are not supported neither by the old data set view nor by sequence views. These features are described above. The next chapter provides examples which show how to use the new features in own applications.

Integration with Interface Painter

The conformance to the old DataSetView interface also allows the new widget to be created and configured within the Interface Painter exactly as you would do this with old data set widgets. In particular you can add a NewDatasetView to your canvas from PaletteTool and configure it using the familiar settings in PropertyTool.

NewDatasetView provides you with some additional pages and properties. Most of those settings are inherited from SequenceView, such as the check box which tells the widget whether to enable multiple selections or not, and the Drag&Drop properties page.

Summary of Features

These are the pro's of the new implementation that are important for you as a developer:

- Allows you to select between single selection or multiple selection behavior.
- Allows you to switch between browsing mode and editing mode at runtime.
- Inherits most of its behavior — and hence its programming interface — from MultiSelectionSequenceView (including Drag&Drop!).
- Conforms to the old DataSetView interface (→ in principal you can use it anywhere you used to use the old widgets)

- Acts more efficiently concerning display, browsing, and editing.
- Seamless integration with Lens Objects and Database Tools (Canvas Composer, …)

And these are the pro's important for you as an end user:

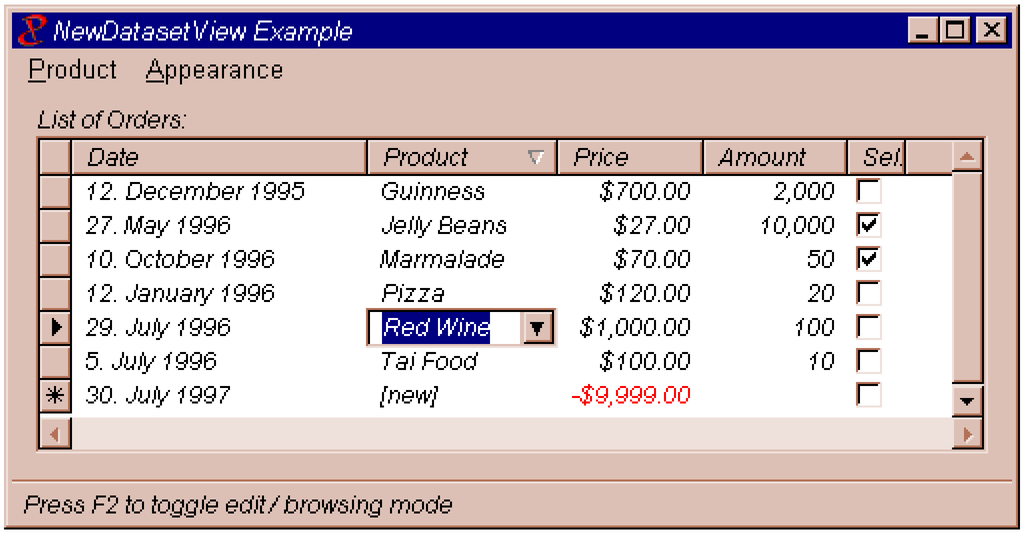
- Adopts the Look and Feel of Windows 95 table controls,
- Allows the user to resize columns in real time using the column labels.
- Allows the user to reposition columns by dragging them to a new position.
- Allows the user to sort the rows by any column using the labels.
- Supports automatic appending of new entries.
- Acts more efficiently concerning display, browsing, and editing.

Cookbook:
How to employ new data set views

In this chapter we will describe how to employ new data set widgets in a VisualWorks application. We will thereby concentrate on the new features introduced. To learn more about the basic steps, you may refer to the corresponding chapter in your VisualWorks documentation (Cookbook, Chapter. 11). It will be shown how a new data set view is added to a canvas, how to toggle between browsing and editing mode, how auto-append blocks can be provided, etc.

The code shown in the single steps is provided in the tutorial application class *NewDatasetExample*. This class is itself derived from *OldDatasetExample*. The latter contains all the code that is not specific for new data sets but depends on the common interface of old and new implementation. If you start it, it shows up with an old data set. NewDatasetExample employs a new data set view. It inherits from its superclass all the basic code to employ a data set view, whether it is an old or a new one. Additionally, new methods are provided to address the new features.

The separation between the two example classes clearly shows the distinction between the common interface, defined by the old implementation and also supported by new data sets, and the new interface provided to address the new features introduced.



List of Orders:

	Date	Product	Price	Amount	Sel.
	12. December 1995	Guinness	$700.00	2,000	☐
	27. May 1996	Jelly Beans	$27.00	10,000	☑
	10. October 1996	Marmalade	$70.00	50	☑
	12. January 1996	Pizza	$120.00	20	☐
▶	29. July 1996	Red Wine	$1,000.00	100	☐
	5. July 1996	Tai Food	$100.00	10	☐
✳	30. July 1997	[new]	-$9,999.00		☐

Press F2 to toggle edit / browsing mode

Adding a New Data Set View

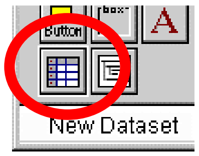
Strategy

Since NewDataSetView supports the interface of the old implementation, the basic steps to add a data set to a canvas remain the same. A new data set also uses an instance of *SelectionInList* to hold the list of objects to be displayed, along with a selection holder. However, if you enable the multiple selection mode, you should provide an instance of *MultiSelectionInList* instead.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. Use a Palette to add a new data set widget to the canvas.



All the remaining steps are the same as those being listed in VisualWorks Cookbook in chapter 11, section „Adding a Dataset“.

Variant A: Replacing an old data set view by a new one

If you want to modify an existing application to use a new data set view, you can do this by replacing „DataSetSpec“ with „NewDataSetSpec“ in the application’s window spec. There should principally be no further changes to your code necessary.

1. Use a system browser to edit your application’s *windowSpec* method. Search for string „**DataSetSpec**“ and replace it with „**NewDataSetSpec**“.

2. Apply the changes and start up the application to test if it works.

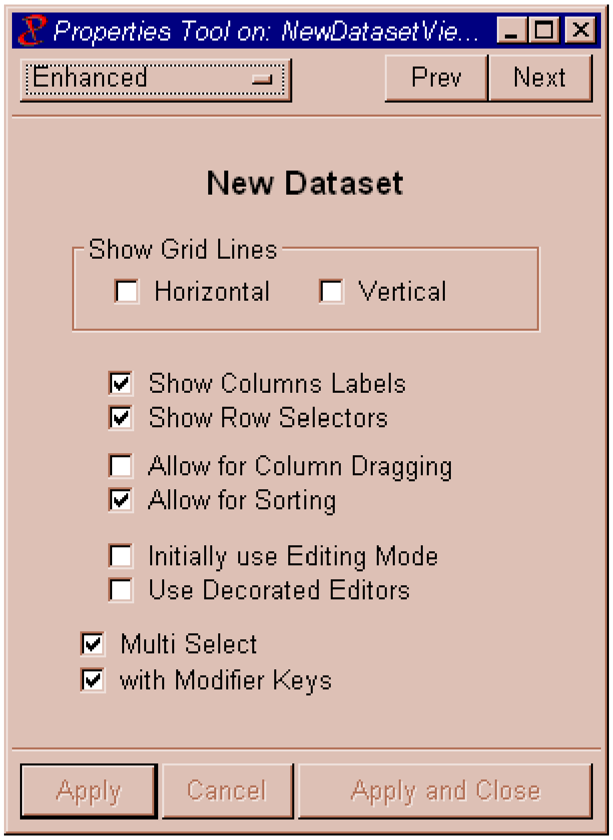
3. You can now edit the modified window spec in Canvas Painter, using all the new properties.

Variant B: Enabling or disabling display of row and column labels

Per default a new data set is pre-configured to display columns labels and row selectors. You can disable this selectively in the Properties Tools.

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. In the Properties Tool's *Enhanced* Page, turn on the data set's **Show Column Labels** and **Show Row Selectors** properties appropriately.



Properties Tool on: NewDatasetVie...

Enhanced | Prev | Next

New Dataset

Show Grid Lines

- [ ] Horizontal
- [ ] Vertical

- [x] Show Columns Labels
- [x] Show Row Selectors
- [ ] Allow for Column Dragging
- [x] Allow for Sorting
- [ ] Initially use Editing Mode
- [ ] Use Decorated Editors
- [x] Multi Select
- [x] with Modifier Keys

Apply | Cancel | Apply and Close

Variant C: Enable Displaying of Grid Lines

You can configure a new data set to display grid lines horizontally, vertically or both.

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. In the Properties Tool’s *Enhanced* Page, turn on the data set’s **Show Grid Lines** properties appropriately.

Variant D: Enabling editing mode initially

If you have replaced an old data set widget with a new one as described in the steps before, the application will start up with the new data set being in browsing mode. Since your application hasn’t any code to enable editing mode, you may want to configure the data set to use editing mode initially.

1. Open the modified window spec in Canvas Painter and select the data set widget.

2. In the Properties Tool’s *Enhanced* Page, turn on the data set’s **Initially use Editing Mode** property.

Variant E: Use decorated editors

New data set widgets are configured to use non decorated editors as the default. This means that no borders are displayed around the input fields used to edit the contents of cells in editing mode. This allows the display and traversal of new data sets to be much faster than that of the old ones that always display a border. However, you can also configure a new data set widget to display bordered editors.

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. In the Properties Tool’s *Enhanced* Page, turn on the data set’s **Use Decorated Editors** property.

Variant F: Disabling interactive sorting and/or column dragging

The built-in facilities for interactive sorting and column dragging are initially enabled when a new data set is added to a canvas. However, you can switch off these facilities.

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. In the Properties Tool’s *Enhanced* Page, turn on/off the data set’s **Allow for Sorting** and/or **Allow for Column Dragging** properties.

Allowing for Multiple Selections

Strategy

In order to enable multiple selection facility for your new data set, you first turn on the appropriate properties in the Properties Tool and then provide an instance of *MultiSelectionInList* instead of *SelectionInList* in the data set's aspect method. The properties to turn on for multiple selection are the same as in case of list boxes (*Multi Select*, *Use Modifier Keys for Multi Select*).

Basic Steps

Tutorial Example: *NewDatasetExample*

1. In the Properties Tool's Details Page, turn on the data set's **Multi Select** property. (Leave **Use Modifier Keys for Multi Select** selected for standard behavior.)

2. In the application model initialize the widget's aspect variable with an instance of *MultiSelectionInList* instead of *SelectionInList*:

```
entries

    entries isNil
        ifTrue:
            [entries := MultiSelectionInList new.
            entries selectionIndexHolder
                compute: [:v | self row value: entries selection]].
    ^entries
```

Analysis

Note that normally, a data set view (at least old ones) expect the aspect to be an instance of *SelectionInList*. You can recognize this by having a closer look at the aspect method above: To ensure compatibility with old data sets, a new data set also uses a second aspect, a so called row holder which holds the selected entry (Generally, you don't care about this row holder). This row holder is updated by a *BlockValue*, which is computed each time the selection changes. Traditionally this happens via a message like *rowHolder value: entries selection*. In this message *entries selection* refers to a single selection, which is normally not supported by instances of *MultiSelectionInList.*

The implementation of new data sets however, include minor enhancements to *MultiSelectionInList* to unify its interface with that of *SelectionInList*. In fact these enhancements allow you to have a new data set view using single selection mode initialized with an instance of MultiSelectionInList as its aspect. Thus, when the data set is in editing mode, you can securely use *selection* or *selectionIndex* to access the only selected entry (During editing no multiple selections are possible). However, when a data set allowing multiple selections is in browsing mode, you should use *selections* or *selectionIndexes* instead.

Toggle editing and browsing mode

Strategy

If an application is to be designed to use a new data set view in both editing and browsing mode, it has to be equipped with code to switch between the two modes. This is done by sending the widget's controller corresponding messages. These messages are:

startEditing to enable editing mode when the widget is in browsing mode. If the widget is already in editing mode, this message has no effect.

stopEditing to return to browsing mode when the widget is in editing mode. If changes have been made to the currently edited entry, these changes are committed. If the widget is already in browsing mode, this message has no effect.

abortEditing to return to browsing mode when the widget is in editing mode. Changes to the currently edited entry are canceled. If the widget is already in browsing mode, this message has no effect.

toggleEditing to switch to editing mode when the widget is in browsing mode or to switch to browsing mode when the widget is in editing mode. If it was in editing mode and changes have been made to the currently edited entry, these changes are committed.

In our example application starting and ending editing is done via a key press on F2 (conforming to file name editing in Windows 95 explorer). The steps described below show how this is achieved.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. Use a system browser to edit the application model's *postBuildWith:* method as shown below.

```
postBuildWith: aBuilder

    aBuilder keyboardProcessor
        keyboardHook: [:ev :ctrl | self keyPress: ev].
    super postBuildWith: aBuilder
```

2. Use a system browser to implement method *keyPress:* to handle key presses as follows: F2 toggles between browsing and editing mode, Return starts editing if we are in browsing mode, Esc aborts editing and returns to browsing mode if the widget is in editing mode, and F12 accepts changes and returns to browsing mode.

```
keyPress: aKeyboardEvent

    | keyValue ctrl |
    keyValue := aKeyboardEvent keyValue.
    ctrl := (builder componentAt: #entries) widget controller.
    keyValue == #F2              ifTrue: [ctrl toggleEditing] ifFalse: [
    keyValue == Character cr[1]  ifTrue: [ctrl startEditing]   ifFalse: [
    keyValue == Character esc    ifTrue: [ctrl abortEditing]   ifFalse: [
    keyValue == #F12             ifTrue: [ctrl stopEditing]
    ifFalse: [^aKeyboardEvent]]]].
    ^nil
```

Validating changes in edited cells

Frequently, an application will have to validate the changes made in an edited cell. The basic steps below show how this can be done by means of *change validation* and *exit validation* callbacks. Change validation prevents input from being passed to the cell's value model unless it is valid. Exit validation prevents the user from moving the focus out of the cell until either an invalid input is corrected or the editing is aborted. Most often, both kinds of validation will have to be performed, usually with the same callback method.

Validation as described in this section can be applied to both, the old and the new implementation. However, since the VisualWorks documentation doesn't talk about this with respect to Data Sets, we will provide some

[1] In fact, the test for this is *(keyValue == Character cr and: [ctrl isBrowsing])*, since presses on Return should only be processed by the application if we are in browsing mode. Otherwise this event would not be passed to the widget's controller, which itself handles Return as *next field* when in browsing mode.

explanation here. You may want to compare the explanations given here with the more comprehensive explanations on validation for Input Fields in VisualWorks Cookbook, Chapter 6.

Basic Steps

Tutorial Example: *OldDatasetExample*

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. Select the column you want to validate input for.

3. In the Properties Tool's *Validation* Page, enter a method selector for both, change validation and exit validation. In this example, the same method (*validateProductName:*) is used for both.

4. Use a system browser to implement the validation callback method corresponding to the selector you specified in step 1 (*validateProductName:*). Return true if the input is valid. Note that, because we specified a method selector with a colon, the method takes a controller as an argument.

```
validateProductName: aController

    | newName oldName |
    newName := aController editValue.

    „If no changes were made, we don't need to perform the
    validation tests"

aController textHasChanged ifFalse: [^true].

    newName isEmpty
        ifTrue:
            [Screen default ringBell.
            Dialog warn: 'You must enter a name for that product!'.
            ^false].

    ^true.    „Ok, validation succceeded ..."
```

Variant: Testing for duplicates

We enhance the validation tests shown in basic step 2 by an additional test for duplicates. If a duplicate is detected, we ask the user if he really wants to use the name he entered. If he confirms this, the validation returns true.

```
validateProductName: aController

        [...]

        „It is ok, if the name was edited but remains the same ...“
        oldName := row value at: 2.
        newName = oldName ifTrue: [^true].

        „ Otherwise we have to look for duplicates ...“
        entries list detect: [:each | (each at: 2) = newName] ifNone: [^true].


        „ A duplicate was found; ask the user if he insists
        on this product name ...“

        Screen default ringBell.
        (Dialog
                confirm:
'There''s already an entry with product name ''', newName, '''
Are you sure you want to insert another one?')
        ifFalse: [^false].


        „The user confirmed the duplicate name. We’ll have to reset
        the change flag and accept the changes here, otherwise the
        user will be asked twice.“

        aController textHasChanged: false.
        aController accept.

        ^true.        „Ok, validation succcceeded ...“
```

Storing and restoring the column layout

Strategy

If a user can change the order and the widths of the columns in a data set view interactively, he or she would certainly want the application to remember these changes and use the same column layout, the next time the application is started. New data set view supports this by the new messages *columnLayout* to retrieve the current column layout information and *columnLayout:* to restore a formerly stored column layout.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. Add a class variable (in this example we call it *ColumnLayout*) to your application model.

2. Use a system browser to create/edit the application model's *noticeOfWindowClose:* method. This method is always called when an application's window is closed. In the method's implementation use *columnLayout* to get the current column widths from the data set view. Store the answer in the class variable provided in step 1 (*ColumnLayout*).

```
noticeOfWindowClose: aWindow

    ColumnLayout := (builder componentAt: #entries)
        widget columnLayout.
    ^super noticeOfWindowClose: aWindow
```

3. Use a system browser to create/edit the application model's *postBuildWith:* method. In the method's implementation use *columnLayout:* to set the data set view's column widths from the class variable provided in step 1 (*ColumnLayout*).

```
postBuildWith: aBuilder

    | widget |
    widget := (builder componentAt: #entries) widget.
    ColumnLayout notNil
        ifTrue: [widget columnLayout: ColumnLayout].
    super postBuildWith: aBuilder
```

Sorting entries programmatically

Strategy

As described in the introduction, new data set widget supports automatic sorting of entries without requiring any additional code to be provided for this purpose. However, sometimes you may want to sort the entries programmatically, such as in result to some menu commands (e.g. *Sort by …*). You can do this by sending message *orderBy:* to a data set view. This message expects a column index as the argument. Index 1 refers to the first data column, row selector columns are ignored.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. Open the application's window spec in Canvas Painter and select the data set widget.

2. In the Properties Tool's Basics Page, fill in the widget's **Menu** aspect (*listMenu*).

3. Apply changes and save the canvas.

4. Use a system browser to add an instance variable to the application model to hold a menu for the aspect selector (*listMenu*).

5. Use a system browser to create an aspect method (*listMenu*) that builds a menu with entries for the columns to sort by. The items' labels are the column names, the items' values are the column indexes.

```
listMenu

    listMenu isNil
        ifTrue:
            [| mb |
            (mb := MenuBuilder new)
                beginSubMenuLabeled: 'Sort by ...\b';
                    add: 'Date' -> 1;
                    add: 'Name' -> 2;
                    add: 'Amount' -> 3;
                    add: 'Price' -> 4;
                endSubMenu.
            listMenu := mb menu asValue].
    ^listMenu
```

6. Use a system browser to create/edit the application model's *postBuildWith:* method. In the method's implementation provide a block to be used as the new data set controller's menu performer. This block takes a menu value — in case of the *listMenu* shown above, these are column indexes — as its argument and uses this index to call *orderBy:*.

```
postBuildWith: aBuilder

    | widget |
    widget := (builder componentAt: #entries) widget.
    widget controller performer: [:i | widget orderBy: i].
    super postBuildWith: aBuilder
```

Variant A: Specifying the initial sort column

When an application employing a new data set view starts up, the first data column is used as the initial sort column to sort the entries in an ascending order. If you want to specify another column as the initial sort column, you can use message *sortColumn:* to do so. This message expects a column index as the argument. Index 1 refers to the first data column, row selector columns are ignored.

1. Use a system browser to edit the application model's *postBuildWith:* method. In the method's implementation send *sortColumn:* with a column index to the data set view.

```
postBuildWith: aBuilder

    | widget |
    widget := (builder componentAt: #entries) widget.
    widget controller performer: [:i | widget orderBy: i].
    widget sortColumn: 2.
    super postBuildWith: aBuilder
```

Variant B: Providing a default sort block

When a data set's entries are sorted by a certain column, only the values in this column are used to determine the entries' order. Sometimes however, you may want the entries to be sorted with a secondary sort criterion, such that a range of entries with equal values in the primary sort column is sorted according to that secondary criterion. New data set view supports this by means of a default sort block.

1. Use a system browser to edit the application model's *postBuildWith:* method. In the method's implementation send *defaultSortBlock:* with a sort block to the data set view. The sort block takes two arguments each one standing for an entry to compare for ordering. The block's return value is expected to be a boolean which tells whether the first argument is considered less than the second.

```
postBuildWith: aBuilder

    | widget |
    widget := (builder componentAt: #entries) widget.
    widget controller performer: [:i | widget orderBy: i].
    widget sortColumn: 2.
    „Provide a default sort block which compares two entries’
    product names ...“
    widget defaultSortBlock: [:a :b | (a at: 2) < (b at: 2)].
    super postBuildWith: aBuilder
```

Configuring for auto-appending new entries

Strategy

New Data Set Widgets can be configured to automatically insert a new entry when a user moves the cell editor past the last edited entry. This feature works like comparable facilities, e.g. that in MS Access table controls. If this feature is used, a template entry is appended as the last entry. If row selectors are enabled, this template entry is marked with a special asterisk icon (‘*’). Application programmers initialize a new data set view with a so called „auto new block“ which is responsible for returning an entry to be used as the template.

When the user edits that entry and subsequently moves the editor to the next entry using Tab or Return, a new template entry is automatically appended. However, no new templates are added if no changes are made to an already existing template entry. The editor won’t move past an unchanged template entry.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. Use a system browser to edit the application model’s *postBuildWith:* method. In the method’s implementation send *autoNewBlock:* to the data set view. This message expects a block as its argument. The block’s return value is expected to be an entry with template values — these values can be empty — to be appended the end of the list each time a new entry is required.

```
postBuildWith: aBuilder

    | widget |
    widget := (builder componentAt: #entries) widget.
    widget autoNewBlock: [self templateEntry]
    super postBuildWith: aBuilder
```

2. Use a system browser to create method *templateEntry* which is responsible for creating the template entries.

```
templateEntry

    ^(Array new: 5)
        at: 1 put: Date today;
        at: 2 put: '[new]';
        at: 3 put: nil;
        at: 4 put: -9999;
        at: 5 put: false;
    yourself.
```

Analysis

Note that the template entries must each be a different object, since they might be changed when a user edits that entry in order to add a new one. You may not use objects stored class variables or literal objects (e.g. a literal array). The template entries will be compared by the data set view using the equality operator (=) in order to detect if the entry has changed and a new entry should be appended, if the user tries to move the editor past the currently last entry.

Adding/Removing columns programmatically

Strategy

In certain application you might need to arrange for dynamic adding and removing of separate columns. One can distinguish two different cases: One case are applications in which all columns are known and can be pre-configured in Canvas Painter. Single columns are then to be hidden and re-displayed dynamically during runtime of the application. Another case are applications in which you don't know about all the columns which might be displayed later on. On contrary, your application will need to construct and add certain columns based on evaluation of some specific configuration in your application model dynamically.

In the tutorial example we will add a menu command "Show Prices" which dynamically shows or hides the Prices column in the data set. Method *showPrices: aBoolean* will arrange for this.

Basic Steps

Tutorial Example: *NewDatasetExample*

1. In your application code send *insertColumn:at:* to the *TreeView* to add a column at a certain position given by index. Send *remove-ColumnAt:* to remove a column.

```
showPrices: aBoolean

    | dataset |
    dataset := builder componentAt: #entries.
    aBoolean
        ifTrue: [dataset widget insertColumn: (dataset spec columns at: 5) at: 4]
        ifFalse: [dataset widget removeColumnAt: 4]
```

Analysis

In the code shown above, a certain column is added or removed depending on the parameter *aBoolean*. Both methods, *insertColumn:at:* and *removeColumnAt:*, expect an index denoting the position at which the column is to be inserted or removed. This index refers to data columns, the row label column is ignored. Furthermore, *insertColumn:at:* expects either an instance of *NewDataSetColumn* or of *DataSetColumnSpec*. In the example it is the latter which is obtained from the widget's spec. Both method take care for appropriate update of the view's display.

Variant A: Hiding and redisplaying pre-configured columns

The given implementation of *showPrices:* could be modified to store the instance of *NewDataSetColumn* in an instance variable before hiding it, and then using this instance variable for redisplaying the column, instead of doing this via the column spec.

1. Add an instance variable *pricesColumn* to class *NewDatasetExample*.

2. Modify implementation of method *showPrices:* as shown below:

```
showPrices: aBoolean

    | dataset |
    dataset := builder componentAt: #entries.
    aBoolean
        ifTrue: [dataset widget insertColumn: pricesColumn at: 4]
        ifFalse: [pricesColumn := dataset widget removeColumnAt: 4]
```

In the code you see that *removeColumnAt:* returns the instance of *New-DataSetColumn* which has been removed. This instance can be used to re-insert that very column subsequently.

Variant B: Dynamically construct and display new columns

If the more complicated situation arises, that you need to construct and add completely new columns based on some dynamic state in your application model, you would provide an instance of *DataSetColumnSpec* as a parameter to *insertColumn:at:*. You will configure this instance with all the necessary attributes, such as column aspect, label, editor spec, fonts and column width.

1. Modify implementation of method *showPrices:* to use a separate method *pricesColumn* to dynamically construct the appropriate instance of *DataSetColumnSpec*.

```
showPrices: aBoolean

    | dataset |
    dataset := builder componentAt: #entries.
    aBoolean
        ifTrue: [dataset widget insertColumn: self pricesColumn at: 4]
        ifFalse: [dataset widget removeColumnAt: 4]
```

2. Implement *pricesColumn* to dynamically construct the appropriate instance of *DataSetColumnSpec*.

```
pricesColumn

	"This could as well be taken from the widget spec, such as in:
	((builder componentAt: #entries) spec columns at: 5)
	Note: index 1 = rowSelector"

	^(DataSetColumnSpec new
		model: #'row 4';			"The column's aspect ..."
		width: 72;				"and its width"
		label: 'Price';				"The label ..."
		labelFont: #DataSetLabel;		"and its font (text style)"
		editorType: #InputField;		"The editor spec ..."
		type: #fixedpoint;
		alignment: #right;
		font: #DataSetEntries;
		formatString: '$#,##0.00;[Red]-$#,##0.00' )
```

If you are not sure how to specify certain attributes of a column spec, it is a good idea to first paint a dummy data set with a corresponding column spec in Canvas Painter and then look up all the attributes. Usually, all you have to do is removing the leading # characters from the attribute names to obtain the appropriate method selectors.

Allowing or disallowing for column dragging and sorting

Strategy

At runtime, an application can configure a new data set widget to allow or disallow for column dragging and interactive sorting. This can be done in an all or none way for all the columns in a data set or on a per-column basis. The basic steps below lists the messages to be sent to a *NewDataSetView* or to a *NewDataSetColumn* to fulfil these tasks.

Basic Steps

Use the messages listed below for enabling/disabling ...

Column Dragging:

```
aNewDataSetView allowForColumnDragging: aBoolean
aNewDataSetColumn canBeDragged: aBoolean
```

Interactive Sorting:

```
aNewDataSetView allowForSorting: aBoolean
aNewDataSetColumn allowForSorting: aBoolean
```

Analysis

In general all these methods expect a boolean as a parameter denoting whether to enable or disable the respective facility. All methods provide for appropriate invalidation and updating. An example for using these messages from within an application model methods is:

```
(builder componentAt: #myDataSet) widget allowForSorting: false.
((builder componentAt: #myDataSet) widget columnAt: 2)
    canBeDragged: false.
```

Disabling column dragging or interactive sorting on the *NewDataSet-View* for all columns has precedence over per-column enabling of these features. E.g., if you would want to disallow for column dragging on certain columns, you would generally allow for column dragging on the *NewDataSetView* (the default for non-frozen columns) and then disallow for column dragging on the desired columns.

Programmatically modifying the appearance

Strategy

At runtime, an application can arrange for enabling or disabling …

- horizontal and vertical grid lines
- display of column labels
- display of row selectors
- decorated cell editors (vs. plain editors without borders)
- column dragging
- sorting entries by columns
- alignment of column labels

The basic steps below lists the messages to be sent to a *NewDataSet-View* to fulfil these tasks.

Basic Steps

Use the messages listed below for enabling/disabling ...

Display of Grid lines:

aNewDataSetView ***showHorizontalLines:*** *aBoolean*
aNewDataSetView ***showVerticalLines:*** *aBoolean*
aNewDataSetView ***showLines:*** *aBoolean*

Display of Column labels:

aNewDataSetView ***showLabels:*** *aBoolean*

Display of Row selectors:

aNewDataSetView ***useRowSelectors:*** *aBoolean*

Decorated Editors:

aNewDataSetView ***useDecoratedEditors:*** *aBoolean*

Analysis

In general all these methods expect a boolean as a parameter denoting whether to enable or disable the respective facility. All methods provide for appropriate invalidation and updating. An example for using these messages from within an application model methods is:

```
(builder componentAt: #myDataSet) widget showLines: true.
```

Setting a column label's text programmatically

Strategy

An application model can override the column label texts as being configured in the window spec at any time.

Basic Steps

1. In a method of your application model use method *labelAt:put:* to change the label's text.

```
(aBuilder componentAt: #entries) widget labelAt: 2 put: 'New Title'.
```

Analysis

The methods *labelAt:* and *labelAt:put:* allow to access/modify a column label's displayed text. Both methods expect an index denoting the column's position. This index refers to data columns, the row label column is ignored.

Setting alignment of column labels programmatically

Strategy

An application model can change the column labels' (default: left) programmatically at runtime.

Basic Steps

1. In a method (e.g. *postBuildWith*) of your application model use method *aNewColumnLabelVisual align:* to change the label's text.

```
((aBuilder componentAt: #entries) widget columnAt: 2)
    labelVisual align: #center.
```

Analysis

The method *aNewColumnLabelVisual align:* allows to change a column label's alignment. The parameter is expected to be one of *#left*, *#right* or *#center*.

Setting the line grid programmatically

Strategy

New data set automatically determines its line spacing from the text style used to display the entries. Sometimes however, you may want to specify a different line grid explicitly.

Basic Steps

1. Open the window spec containing the new data set in Canvas Painter and select the widget.

2. In the Properties Tool's Basics Page, fill in the ID property (e.g. with *#entries*).

3. Use a System Browser to create/edit a *postBuildWith:* method in your application model class as follows:

```
postBuildWith: aBuilder

    (aBuilder componentAt: #entries) widget lineGrid: 24.
```

Customizing rendering of cell contents

Strategy

NewDataSet provides support for custom painting of cell contents by means of so called *display blocks*. E.g. this can be used to paint a cell’s background in a specific colour before continuing with the rendering of the cell’s contents.

Basic Steps

Tutorial Example: *NewDataSetExample*

1. Use a System Browser to set the widget’s display blocks in the ApplicationModel’s *postBuildWith:* method:

```
postBuildWith: aBuilder

    | widget |
    widget := (aBuilder componentAt: #entries) widget.

    widget visualBlockAtColumn: 1
        put:
            [:view :cell :gc :box |
            TreeView folderIcon displayOn: gc at: box origin + (2@0).
            box left: box left + 14].

    widget selectedVisualBlockAtColumn: 1
        put:
            [:view :cell :gc :box |
            TreeView folderIcon displayOn: gc at: box origin + (2@0).
            box left: box left + 14].
```

Analysis

The visualBlock and selectedVisualBlock are somewhat 'abused' in NewDataSetView to hold either a single visual block or an array of vb's, one for each column. In contrast to SequenceView the display blocks don't return a visual but provide for direct rendering using a graphics context as a block parameter.

A display block is evaluated in method *NewDataSetView displayVisualAt:...* The block may return false if it displays the cell's contents completely. Otherwise the standard cell contents display method *(NewDataSetView displayCellAt:...)* is executed after the block returns. In the example, the visual block displays an icon in the cell's upper left corner and changes the cell's left bound temporarily, such that the default display method will display the cell contents with an horizontal offset of 14 pixels.

Customizing the labels’ background color

Strategy

Per default, NewDataSetView uses *SymbolicPaint pushButtonBackground* to display the background of column labels and row selectors. This looks fine on UI Looks like Motif, OS/2, Windows 95, etc. On Windows 3.x and Mac UI Looks however, the background color is white. You can change this by re-configuring class *NewRowVisual.*

Basic Steps

1. Use a Workspace to type in and evaluate this:

```
NewRowVisual backgroundColor: Color veryLightGray.
```

Customizing the overall behavior

Strategy

Several overall settings can be customized with class methods. In particular you can customize ...

- selection hiliting
- handling of re-selecting an entry
- the minimum allowed column width

Customizing Selection Hiliting

Display of selection hiliting can be customised selectively in class *EnhancedSequenceView* and its subclasses (*TreeView*, *NewDataSetView*) by sending *useStandardHiliting: aBoolean* to the respective class. The default is to display the selected entry/entries in a reverse appearance regardless of whether the widget has the input focus or not (*useStandardHiliting: true*). This is also the default behaviour of list boxes in VisualWorks.

You can configure class *EnhancedSequenceView*, and hence all its subclasses — namely *TreeView* and *NewDataSetView* —, or each subclass selectively to adopt the Win95-style behaviour of hiliting the selected entry only if the respective widget has the input focus by sending *useStandardHiliting: false*.

Customizing handling of re-selecting an entry

Handling of repeated selection can be customised selectively in class *EnhancedSequenceView* and its subclasses (*TreeView, NewDataSetView*) by sending *deselectOnReselection: aBoolean* to the respective class. The default is to deselect a selected entry if it is re-selected, with the subclasses sharing this configuration with *EnhancedSequenceView*.

Adjusting the minimum allowed column width

You can adjust the minimum allowed column width to be kept on column resize. This can be adjusted system-wide by evaluating something like:

```
NewDataSetView minCellWidth: 12.
```

Using new data sets in Lens Applications

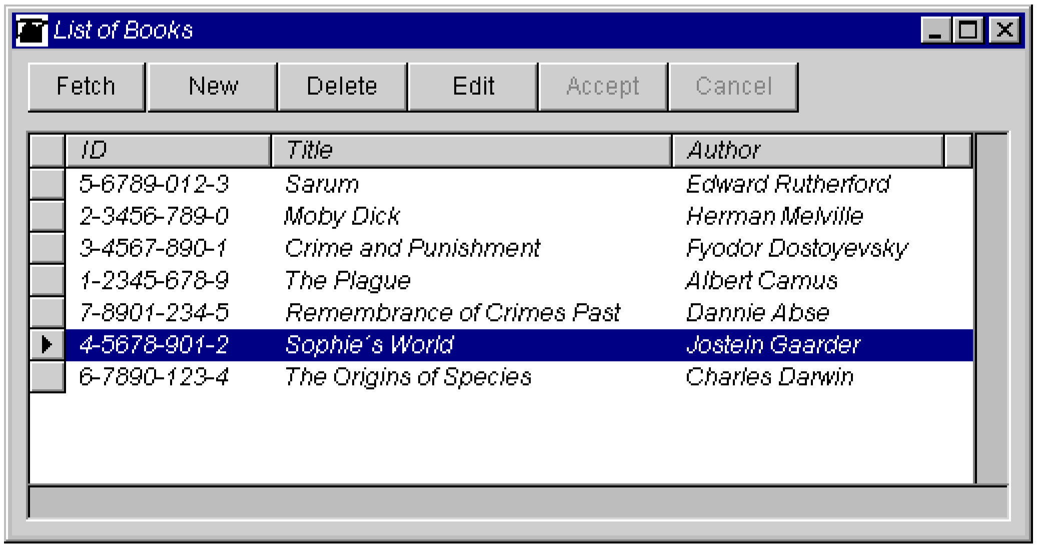
Strategy

Data sets are frequently used in database applications. Object Lens, VisualWorks persistence framework, comes with specialized application models for database applications (*LensApplicationModel*, *LensDataManager*). In particular, *LensDataManager* provides a comprehensive functionality to use data set views for displaying and editing sets of data either in forms one by one, or in data sets. Furthermore, advanced tools, such as *CanvasComposer* are provided to automatically generate applications and user interfaces based on a given lens data model.

New data set includes a subclass of LensDataManager which is specialized on using new data set views in a tabular database editor form. *LensDataSetManager* — this is the name of that subclass — seamlessly integrates new data sets into the lens machinery described above. It enhances certain methods of its superclass to automatically switch the data set to editing or browsing mode. Such as:

- When the application starts up and records are fetched, the data set is initially in browsing mode.
- When the **Edit** button is pressed — or when editing is started in general—, the data set is switched to editing mode.
- When the **New** button is pressed — or when a new row is added in general—, the new row is selected and the data set is switched to editing mode.
- When the **Accept** or **Cancel** button is pressed — or when editing is stopped in general—, the data set is returns to browsing mode.

All this is done in a transparent way. The lens application is doesn't have to notice that it is derived from *LensDataSetManager* instead of *Lens-DataManager*. The picture below shows an example of a Lens Application employing a new data set view.



ID	Title	Author
5-6789-012-3	Sarum	Edward Rutherford
2-3456-789-0	Moby Dick	Herman Melville
3-4567-890-1	Crime and Punishment	Fyodor Dostoyevsky
1-2345-678-9	The Plague	Albert Camus
7-8901-234-5	Remembrance of Crimes Past	Dannie Abse
4-5678-901-2	Sophie´s World	Jostein Gaarder
6-7890-123-4	The Origins of Species	Charles Darwin

Basic Steps

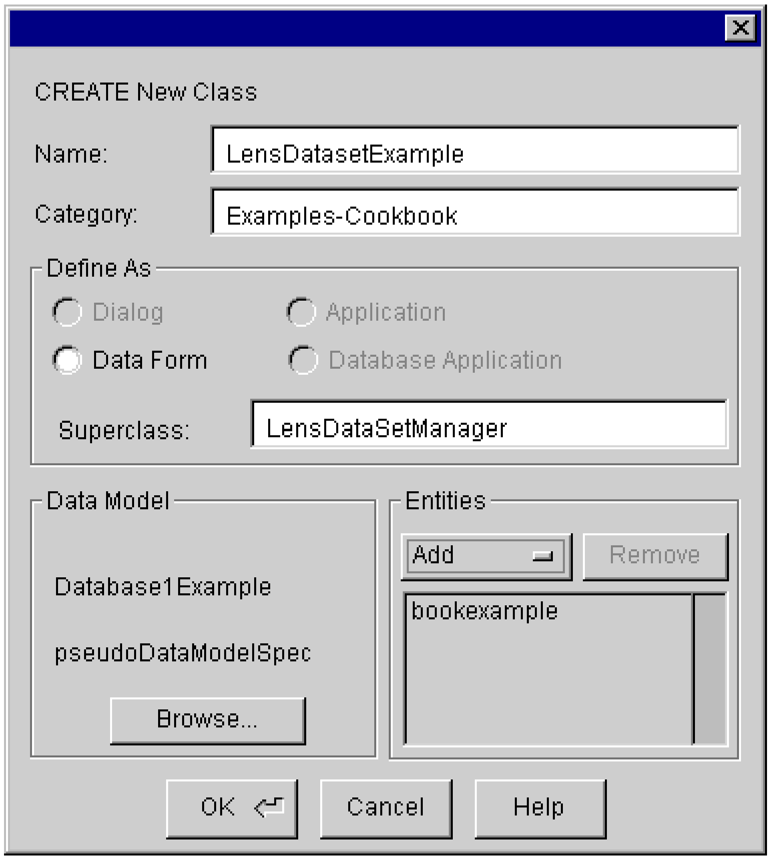
Tutorial Example: *LensDatasetExample*

1. Select *Database* ® *New Data Form…* from the VisualLauncher menu bar to open a dialog which helps you to create a new subclass of LensDataManager.

2. In the dialog's **Name** field type in the name of the new class (*Lens-DatasetExample*).

3. In the dialog's **Superclass** field type *LensDataSetManagaer* as the name of the superclass to derive from.

4. Complete the steps as described in the Database Cookbook (select a data model, entities, etc.).



CREATE New Class

Name: LensDatasetExample

Category: Examples-Cookbook

Define As

Superclass: LensDataSetManager

Data Model

Database1Example

pseudoDataModelSpec

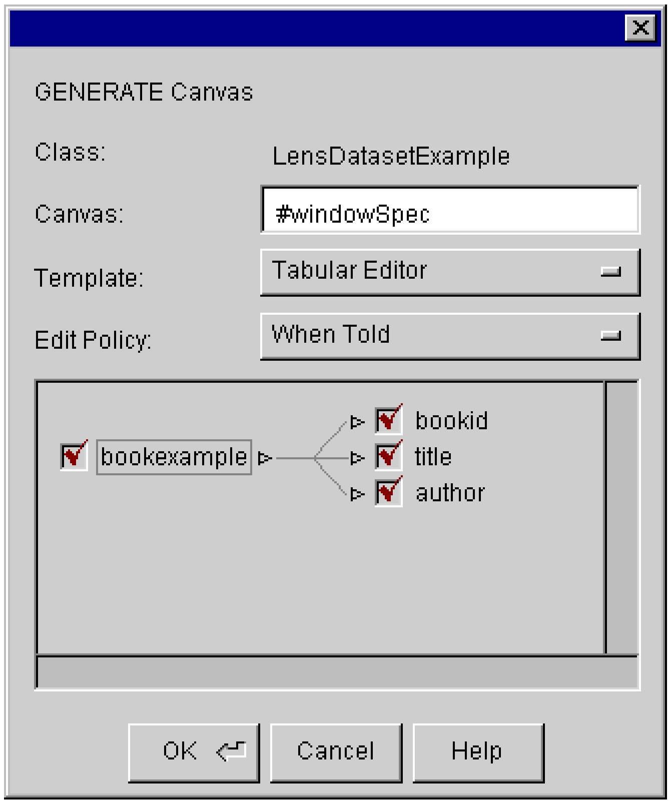
Entities

bookexample

5. Press OK to generate the class. ⇒ The class is created and a canvas composer dialog shows up to specify the user interface.

6. Select „Tabular Editor“ or „Tabular Viewer“ as the template to use.

7. Complete the steps as described in the Database Cookbook (select an edit policy, select the columns to display, etc.).


GENERATE Canvas

Class: LensDatasetExample

Canvas: #windowSpec

Template: Tabular Editor

Edit Policy: When Told

✓ bookexample
- ✓ bookid
- ✓ title
- ✓ author

8. Press OK to generate the interface. ⇒ A canvas displaying the new interface shows up. It already contains a fully configured new data set widget. You can immediately test drive your new Lens application by selecting **Open** from the canvas tool.

Analysis

Note that the data set view's **ID** property in the Properties Tool's Basics Page must contain the symbol *#list*. This is because LensDataSetManager must access the new data set view, and it does so by means of its ID. It assumes the data set's ID to be named *#list*. However you can choose another ID (say *#rows*), if you re-implement method *dataSetID* in your application model like this:

```
dataSetID
    ^#rows
```

Index