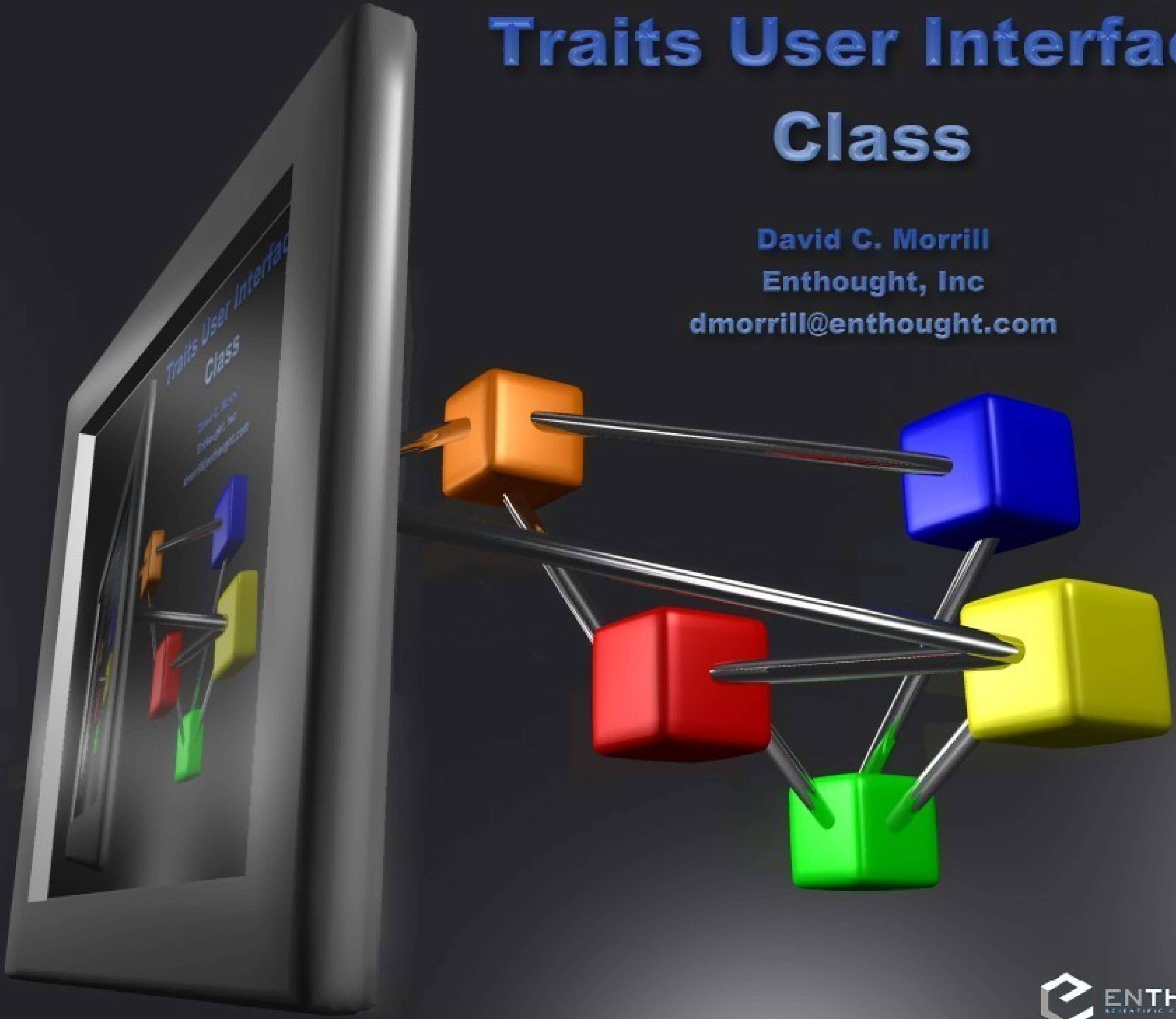
Traits User Interface Class

David C. Morrill
Enthought, Inc
dmorrill@enthought.com



ENTHOUGHT

What We’re Going To Cover…

- The MVC programming model
- Creating Traits UI Views
- Creating Traits UI Handlers
- Defining Editors

GUI Design 101: The Model, View, Controller Approach

- The Traits UI supports, and is based on, the MVC design pattern.
- The MVC pattern defines a UI in terms of three components:
  - Models
  - Views
  - Controllers

GUI Design 101: Models

- A model is the data and behavior that represents (i.e. models) some aspect of a particular problem domain.
- For example: A well log contains measurements that model a particular physical property of a lithological column.
- Models do not have an inherent visual representation.

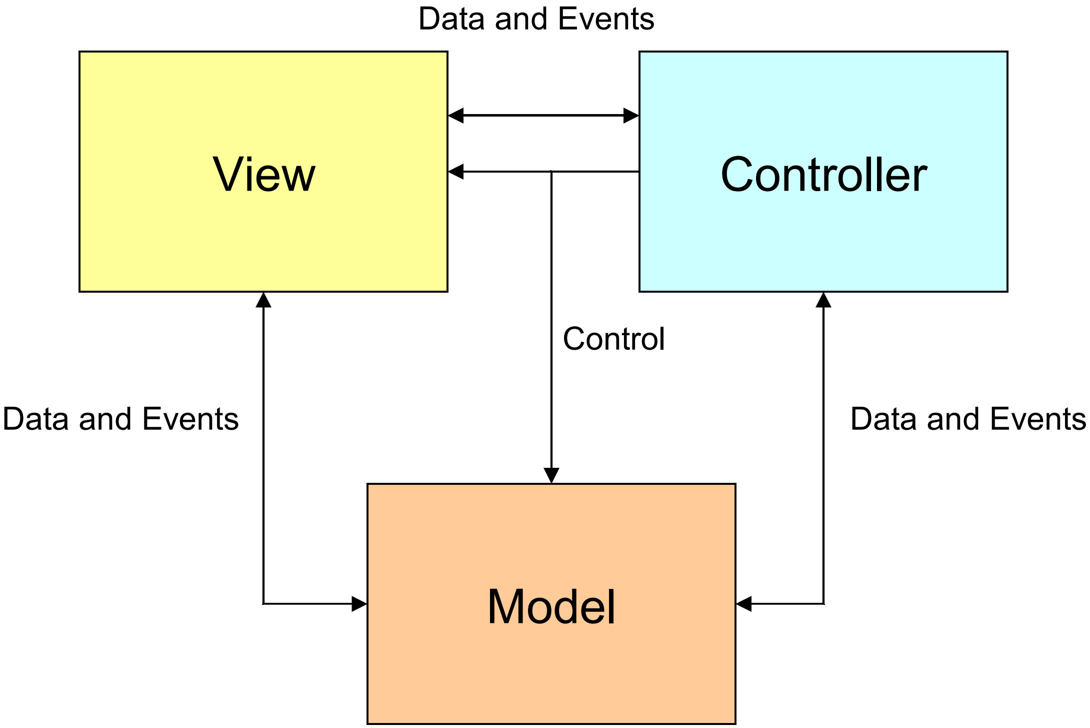
GUI Design 101: Views

- A view is the visual representation of some aspects of a model.
- For example: A line plot can be used as a view of a well log model.
- It’s possible, and common, for a model to have more than one view (e.g. a plot and spreadsheet view of a well log model).

GUI Design 101: Controllers

- A controller defines behavior that mediates and controls the flow of information between the user, the view and the model.
- Having a controller helps prevent the model from becoming cluttered with view specific details and code.

GUI Design 101: The MVC Big Picture



Data and Events

View

Controller

Control

Data and Events

Data and Events

Model

Defining a Model

- Defining a model using Traits is easy…
- Any object with traits is a model that can be used with the Traits UI.
- The object’s traits define the data and events that the model provides.

Defining a View

- A Traits UI defines an MVC view using View objects.
- A View object defines the general characteristics of a particular user interface view.
- A View also contains content which describes the individual items displayed in the view.
- A View does not reference a model directly.
- In general, a View is a reusable object that can be used to create multiple, simultaneous views for different model object instances.

The Kinds of Views

- There are seven basic types of views supported by the Traits UI:
  - Modal
  - Live
  - Livemodal
  - Nonmodal
  - Wizard
  - Panel
  - Subpanel

Modal Views…

- Suspend the application until the user dismisses them.
- Are displayed in a separate dialog window.
- Always make a copy of the model, interact with the copy, then update the original model from the copy when the user clicks OK. If the user cancels, no changes are made to the original model.

Live Views…

- Allow the user to continue working with other parts of the application (they are not modal).
- Are always displayed in a separate window.
- Interact directly with the specified model. Changes made to the model in a view are immediately seen by other parts of the application.

Livemodal Views…

- Are a cross between live and modal views.
- Suspend the application until the user dismisses them (i.e. they are modal).
- Always appear in a separate window.
- Do not make copies of the model. Changes made to the model by the view are immediately seen by other parts of the application (even though the user cannot interact with those parts).

Nonmodal Views…

- Are the inverse of livemodal views…
- Allow the user to continue working with other parts of the application (they are not modal).
- Are always displayed in a separate window.
- Always make a copy of the model, interact with the copy, then update the original model from the copy when the user clicks OK. If the user cancels, no changes are made to the original model.

Wizard Views…

- Organize their contents into a series of “wizard” pages which the user must process sequentially.
- Suspend the application until the user dismisses them (i.e. they are modal).
- Are displayed in a separate window.
- Operate directly on the model.

Panel Views…

- Are displayed as part of a containing window or panel.
- Allow a non-Traits UI window to intermix Traits UI and non-Traits UI elements together (e.g. within a **wx.Frame** window).
- Can contain the normal buttons that the other **View** kinds allow.
- Operate directly on the model.

Subpanel Views…

- Are nearly identical to panel views.
- The only difference is that a subpanel view *never* displays any of the standard Traits UI buttons, even if the **View** object specifies them.

VET: View Editing Tool

- VET is a tool for building Traits user interfaces.
- We’ll be using it today to interactively illustrate many of the Traits UI features…

View Buttons

- Views (except for **subpanel**) support a standard set of optional buttons:
  - **OK**: Allows the user to close a view after having successfully edited the model.
  - **Cancel**: Allows the user to close a view, discarding any changes made to the model.
  - **Undo/Redo**: Allows the user to undo or redo any or all changes made to the model.
  - **Revert**: Allows a user to undo and discard all changes made to the model without closing the **View** window.
  - **Apply**: Allows the user to apply all changes made to a copy of the model to the original model without closing the **View** window.
  - **Help**: Allows the user to display **View** specific help information.

Controlling a View Window’s Appearance

- A **View** defines several traits that can be used to control the appearance of a window:
  - **width, height**: Window size. It can be:
    - an absolute pixel value (e.g. width = 400 means width is 400 pixels)
    - a fraction of the screen size (e.g. width = 0.5 means 0.5 * screen width)
  - **x, y**: Window position. It can be:
    - an absolute pixel value (e.g. x = 50 means left edge is 50 pixels from the left edge of the display, and x= -50 means the right edge is 50 pixels from the right edge of the display)
    - a fraction of the screen size (e.g. x = 0.1 means the left edge is at 0.1 * screen width, and x = -0.1 means the right edge is 0.1 * screen width from the right edge of the display).

Controlling a View Window’s Appearance (cont.)

- Additional **View** traits that affect the window’s appearance are:
  - **title**: Specifies the contents of the window’s title bar.
  - **resizable**: If **False** (the default), the window has a fixed size. If **True**, the window will have a sizing border.

Specifying a View’s Contents

- A **View** has three primary types of content:
  - The actual editors (i.e. widgets) that appear in the view, specified using **Item**, **Group** and **Include** objects.
  - A menu bar, specified using **Menu**, **Action** and **Separator** objects.
  - A tool bar, specified using **Action** and **Separator** objects.
- It can also have various optional buttons (e.g. Undo, OK, Cancel) specified using the **buttons** trait on the **View** itself.

The Item Object

- An **Item** defines a single editor or control that appears in a view.
- In some cases, an **Item** corresponds to a single object trait to be edited.
- In other cases, an **Item** represents a non-editable, visual element in the view, such as a separator line, or extra space between fields.
- The type of **Item** is determined by the value of its traits, which can be divided up into several categories…

Specifying an Item’s Content

- **name**: A string that specifies the name of the trait the **Item** will edit.
  - If empty, the **Item** defines a label only (using the *label* trait).
  - If = ‘ ‘ (a blank), the **Item** simply inserts extra space into the view.
  - If = ‘_’, the **Item** inserts a separator line into the view.
  - If = ‘nn’, the **Item** inserts ***nn*** pixels of space into the view.
- **object**: A string that specifies the name of the view *context* object the *name* trait belongs to. Unless you are editing multiple objects in a single **View**, this is typically left to its default value of “object”.

Controlling an Item’s Presentation and Editing

- **editor**: The editor (factory) used to edit the contents of the trait. If not specified, the editor will be obtained from the trait itself.
- **format_str**: A string containing standard Python formatting sequences (e.g. “%5d*”)* that can be used in conjunction with most text-based trait editors to format the trait value for editing.
- **format_func**: An alternative to *format_str* that specifies a *callable* that can be used with most text-based trait editors to format the trait value for editing.

Controlling an Item’s Appearance

- **label**: A string specifying the label to display next to the editor. The default is to use the trait name to automatically define the label (e.g. a trait name of *employee_name* becomes a label of *Employee name*).
- **style**: A string specifying the style of editor to use. The possible values are:
  - **simple**: A *property sheet* style editor, fits on a single line
  - **custom**: A custom editor that usually presents a more elaborate UI that may require a larger amount of screen real estate.
  - **text**: A single line text editor (i.e. the user must always type in a value)
  - **readonly**: A non-editable (i.e. display only) editor.

  The default is to use the style specified by the containing **Group** or **View**.

Controlling an Item’s Appearance (cont.)

- **width/height**: An integer value (default: -1) specifying the desired width/height of an item.
  - If the value is positive, *max(value, minimum_needed)* is used.
  - If the value is < -1, *abs(value*) is used, even if it is less than what the item says it needs.
  - A value of -1 means use the size requested by the item itself.
- **resizable**: A boolean (default **False**) specifying whether the item benefits from extra space (e.g. lists, tables, trees, multi-line text editors).
- **padding**: An integer value (default 0) specifying the amount of extra padding that should be added around an item.

Controlling an Item’s Visibility and Status

- **defined_when**: A Python expression evaluated when the UI for a **View** is created. If the value of the expression is true, the item is included in the UI; otherwise the item is omitted.
- **visible_when**: A Python expression evaluated when the UI for a **View** is created, and also whenever any trait belonging to any object in the UI’s *context* is changed. If the value of the expression is true, the editor for the item is visible; otherwise the editor is hidden.
- **enabled_when**: A Python expression evaluated when the UI for a **View** is created, and also whenever any trait belonging to any object in the UI’s *context* is changed. If the value of the expression is true, the editor for the item is enabled; otherwise the editor is disabled.

Note: These expressions are useful for relatively simple cases. For more complex cases, a **Handler** subclass should be used.

Providing User Assistance for an Item

- **tooltip**: A string specifying information that should be displayed as a *tooltip* whenever the mouse pointer is positioned over the item’s editor. The default is that no tooltip is displayed.
- **help**: A string specifying a complete help description of the item. This description is used when the **Help** button is clicked to automatically synthesize a complete help page for the UI.

The Group Object

- The **Group** object is used to organize **Item** or other **Group** objects visually and logically.
- Groups can be nested to any depth.

Specifying a Group’s Content

- The contents of a group are specified as any number of positional arguments to the **Group** constructor, or by assigning a list of values to the group’s **content** trait.
- Each item added to a group can be an:
  - **Item**
  - **Group**
  - **Include**
- The contents of the group are laid out in the same order they are added to the group.

Organizing a View’s Layout using Groups

- **orientation**: A string specifying the layout orientation used by the group. The possible values are:
  - **vertical**: The contents of the group are laid out in a single column. This is the default.
  - **horizontal**: The contents of the group are laid out in a single row.
- **layout**: A string specifying the layout method used by the group. The possible values are:
  - **normal**: The contents of the group are laid out normally, with no special handling. This is the default.
  - **split**: Similar to **normal**, but splitter bars are inserted between group items to allow the user to adjust the space devoted to each item. **Note**: the position of the splitter bars can be made a user preference item by assigning a value to the group’s **id** trait.
  - **tabbed**: Each item contained in the group is displayed on its own separate notebook page.

More complex layouts are accomplished by appropriate nesting of groups.

Controlling a Group’s Appearance

- **show_labels**: A boolean (default: **True**) specifying whether or not labels should be displayed next to each group item.
- **show_left**: A boolean (default: **True**) specifying whether labels, if shown, should be displayed to the left (True) or right (False) of the group’s items.
- **show_borders**: A boolean (default: **False**) specifying whether or not a border should be drawn around the contents of the group.
- **label**: A string specifying a label used to describe the entire group. If the value is the empty string (the default), no label is displayed. Otherwise. if **show_borders** is **True**, the label is displayed as part of the border drawn around the contents of the group. If **show_borders** is **False**, the label is displayed as a fancy text label if style is ‘custom’, or a simple text label if it is not.

Controlling a Group’s Appearance (cont.)

- **style**: A string specifying the default style to use for each item in the group. As with an **Item**, the possible values are: **simple**, **custom**, **text**, **readonly**. The group’s style value is only used for items contained in the group that do not explicitly specify their own style value.
- **padding**: An integer value (default: 0) in the range from 0 to 15, specifying the amount of extra padding to insert between each group item as well as around the outside of the group.
- **selected**: A boolean (default: **False**) specifying whether the group should be the selected notebook page when the containing **View** is displayed. Obviously, this only applies when the group represents a page within a notebook, and only one group at most within the notebook should have a **True** value.

Controlling a Group’s Visibility and Status

- **defined_when**: A Python expression evaluated when the UI for a **View** is created.
  - If the value of the expression is **True**, the group and its contents are included in the UI.
  - Otherwise the group and its contents are omitted.
- **visible_when**: A Python expression evaluated when the UI for a **View** is created, and also whenever any trait belonging to any object in the UI’s *context* is changed.
  - If the value of the expression is **True**, the group and its contents are visible.
  - Otherwise they are hidden.

Controlling a Group’s Visibility and Status (cont.)

- **enabled_when**: A Python expression evaluated when the UI for a **View** is created, and also whenever any trait belonging to any object in the UI’s *context* is changed.
  - If the value of the expression is **True**, the group and all editors for items contained in the group are enabled
  - Otherwise the group and all contained editors are disabled.
  - Note that this can be used to control a user’s progress through a *wizard* **View**, by enabling and disabling the groups defining each wizard page.

These expressions are useful for relatively simple cases. For more complex cases, a **Handler** subclass should be used.

Providing User Assistance for a Group

- **help**: A string (default ‘’) specifying a user-oriented description of the group’s contents.

This information is used to automatically create a help page for the containing **View** when the **Help** button is clicked. The help information for the group will be combined with the information provided by the **help** traits of the group’s content items.

Special Group Subtypes

- **HGroup**: A group of horizontally laid out items
- **VGroup**: A group of vertically laid out items
- **HSplit**: A group of horizontally split items
- **VSplit**: A group of vertically split items
- **Tabbed**: A group of tabbed notebook items

The Include Object

- **Include** objects are references to other view elements, namely **Items** and **Groups**.
- **Include** objects allow for factoring views into smaller pieces that are dynamically *included* when a user interface is constructed from a **View**.
- This allows for several interesting capabilities, including:
  - Parameterized views
  - “Visual” inheritance

How Include Objects are Handled

- When a user interface is being constructed from a **View**, any imbedded **Include** objects are logically replaced by the **Item** or **Group** object they refer to.
- For example:

```
my_view   = View( ‘a’, Include( ‘my_group’ ), ‘e’ )
my_group = Group( ‘b’, ‘c’, ‘d’ )
```

is equivalent to:

```
my_view = View( ‘a’, ‘b’, ‘c’, ‘d’, ‘e’ )
```

- The process is recursive, and will continue until no **Include** objects remain in the resulting **View**.

Creating Implicit Include Objects

- A class **View** containing **Group** or **Item** objects with non-default **id** traits is automatically refactored into an equivalent **View** using **Include** objects.
- For example:

```
my_view = View( Group( ‘a’, ‘b’, id = ‘my_group’ ),
                Group( ‘x’, Item( ‘y’, id = ‘my_item’ ) ) )
```

is equivalent to:

```
my_view = View( Include( ‘my_group’ ),
          Group( ‘x’, Include( ‘my_item’ ) ) )
my_group = Group( ‘a’, ‘b’ )
my_item = Item( ‘y’ )
```

Using Include Objects Effectively

- There are several possible uses for **Include** objects.
- One example is creating “parameterized” views.
- For example, the traits **TableFilter** class contains the following view:

```
traits_view = View( 'name{Filter name}', '_',
                    Include( 'filter_view' ),
                    title  = 'Edit filter', … )
filter_view = Group()
```

- This allows **TableFilter** subclasses to define a new version of the ***filter_view*** **Group** containing their custom content without having to redefine the main **TableFilter** view.

View Objects as Part of a Class Definition

- View elements, like **View**, **Group** and **Item** objects, can be created and used in any context, such as module level variables or dynamically created within methods or functions.
- However, there are some additional semantics that apply when they are created statically as part of a class definition…

Defining Views, Groups and Items within a Class

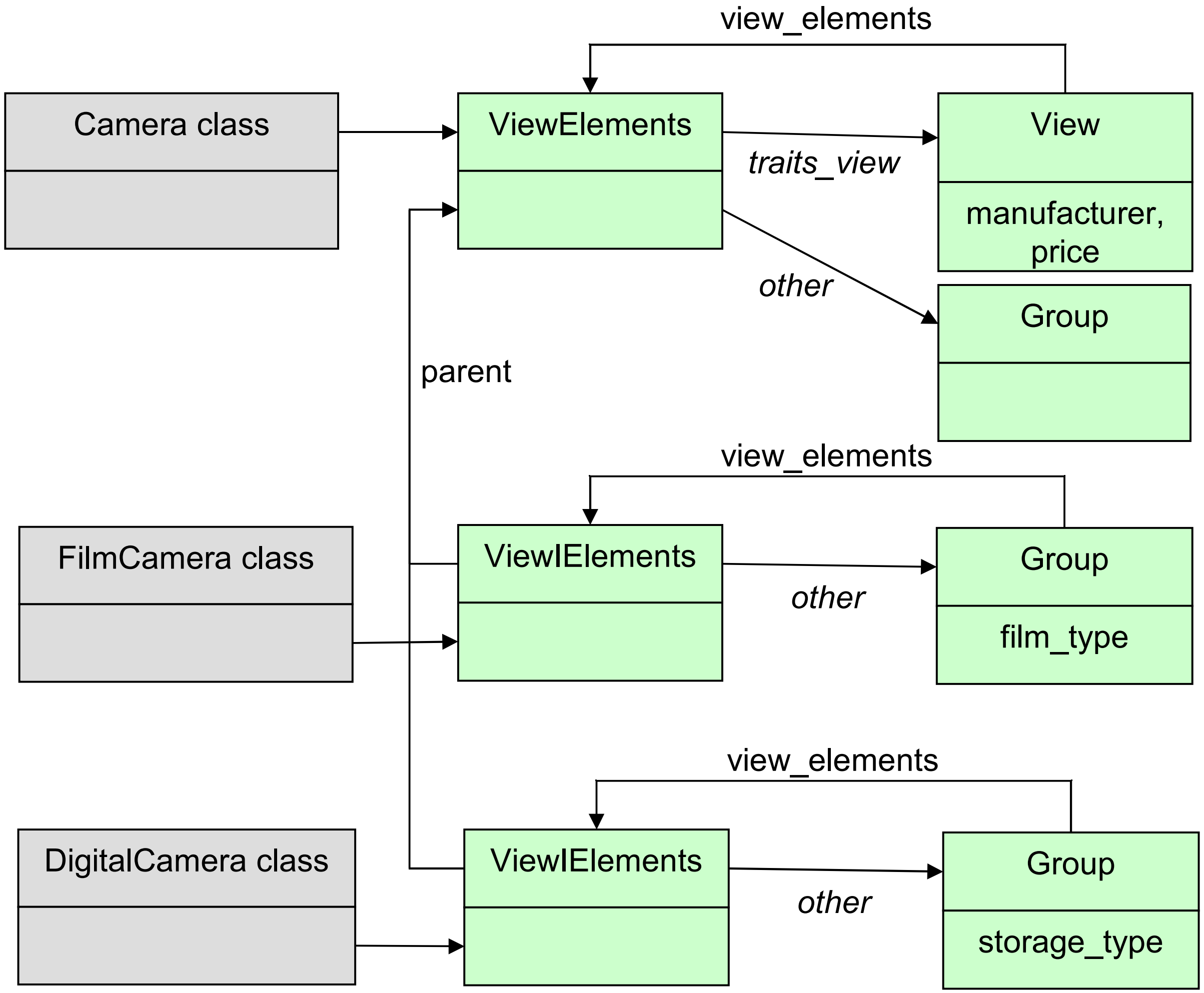
- Views elements created within a class definition have semantics similar to methods. In particular:
  - They are inherited by subclasses
  - They can be overridden by subclasses
- This leads to a feature referred to as “visual inheritance”…

“Visual” Inheritance

- Just like methods can be overridden in subclasses to customize behavior without rewriting an entire class, so to can view elements be overridden.
- For example:

```
class Camera ( HasTraits ):
    manufacturer = Str
    price             = Float
    traits_view     = View( ‘manufacturer, ‘price’, Include( ‘other’ ) )
    other              = Group()

class FilmCamera ( Camera ):
    film_type = Enum( ’35mm’, ’16mm’, ‘8mm’, ‘Polaroid’ )
    other       = Group( ‘film_type’ )

class DigitalCamera ( Camera ):
    storage_type = Enum( ‘CompactFlash’, ‘SD’, ‘xD’ )
    other             = Group( ‘storage_type’ )
```

“Visual” Inheritance



The tr**ait_view** Method

- The **trait_view** method allows you to get or set view element related information on an object.
- For example:
  - **obj.trait_view()** returns the default **View** associated with *obj*.
  - **obj.trait_view(‘my_view’)** returns the view element named ***my_view*** (or none if ***my_view*** is not defined).
  - **obj.trait_view(‘my_group’, Group(‘a’, ‘b’))** defines a **Group** with the name ***my_group***. This group can either be retrieved using **trait_view** or as the view element referred to by an **Include** object imbedded within a **View**.

The **trait_views** Method

- The **trait_views** method can be used to return the list of names of some or all view elements associated with a class.
- For example:
  - **obj.trait_views()** returns the names of all **View** objects associated with *obj*.
  - **obj_trait_views(Group)** returns the names of all **Group** objects associated with *obj*.

Events

- In the context of the Traits UI and user interface creation, an ***event*** is something that happens while a user is interacting with a user interface.
- More specifically, events include:
  - Button clicks.
  - Menu selections.
  - Window/Dialogs being opened or closed.
  - Changes made to a field or value by the user.
  - Changes made to a field or value by some other part of the program.

Event Handlers

- An ***event handler*** is a method or function that is called whenever a specific event occurs (e.g. the OK button being pressed).
- Proper event handling is the key to writing flexible and powerful user interfaces.
- In a traits UI, event handlers can either be written:
  - As methods on the model.
  - As methods on a special object called a **Handler**.

The Handler Class

- In the MVC programming model, a **Handler** class instance is a *controller*.
- If you are using the VET tool, it automatically builds a **Handler** for you.
- The main purpose of a **Handler** subclass is:
  - To handle events common to every traits UI view.
  - To handle events specific to a particular model and view.
  - To help keep model code separate from user interface specific details.

The Handler Class: Common Events

- The events (and methods) common to every **Handler** subclass are:
  - **init (info)**: Handle view initialization
  - **close (info, is_ok)**: Handle a user request to close a dialog or window.
  - **closed (info, is_ok)**: Handles a dialog or window being closed.
  - **setattr (info, object, name, value)**: Handles a request to change a model trait value.

The Handler Class: View Specific Events

- Event handlers specific to a particular **View** fall into the following categories:
  - Trait change handlers.
  - Menu and toolbar action handlers.

The Handler Class: Trait Change Handlers

- A trait change handler is called for each model trait when a view is initialized or when the trait changes value.
- A trait change handler is optional. If one is not defined, its corresponding event is ignored.
- A handler always has the following method signature:
  - *object*_*trait*_changed (info)
- where:
  - ***object***: The name of the context object containing the trait.
  - ***trait***: The name of the trait.
  - **info**: A **UIInfo** object containing information about the current state of the user interface.

The Handler Class: Menu/Toolbar Action Handlers

- Menu and toolbar action handlers are *not* optional.
- They are explicitly referenced by **Action** objects specified in a **View**.
- The signature for an action handler is always:
  - *method_name*(info)
- where:
  - ***method_name***: The method name specified in the corresponding **Action** object.
  - **info**: A **UIInfo** object containing information about the current state of the user interface.

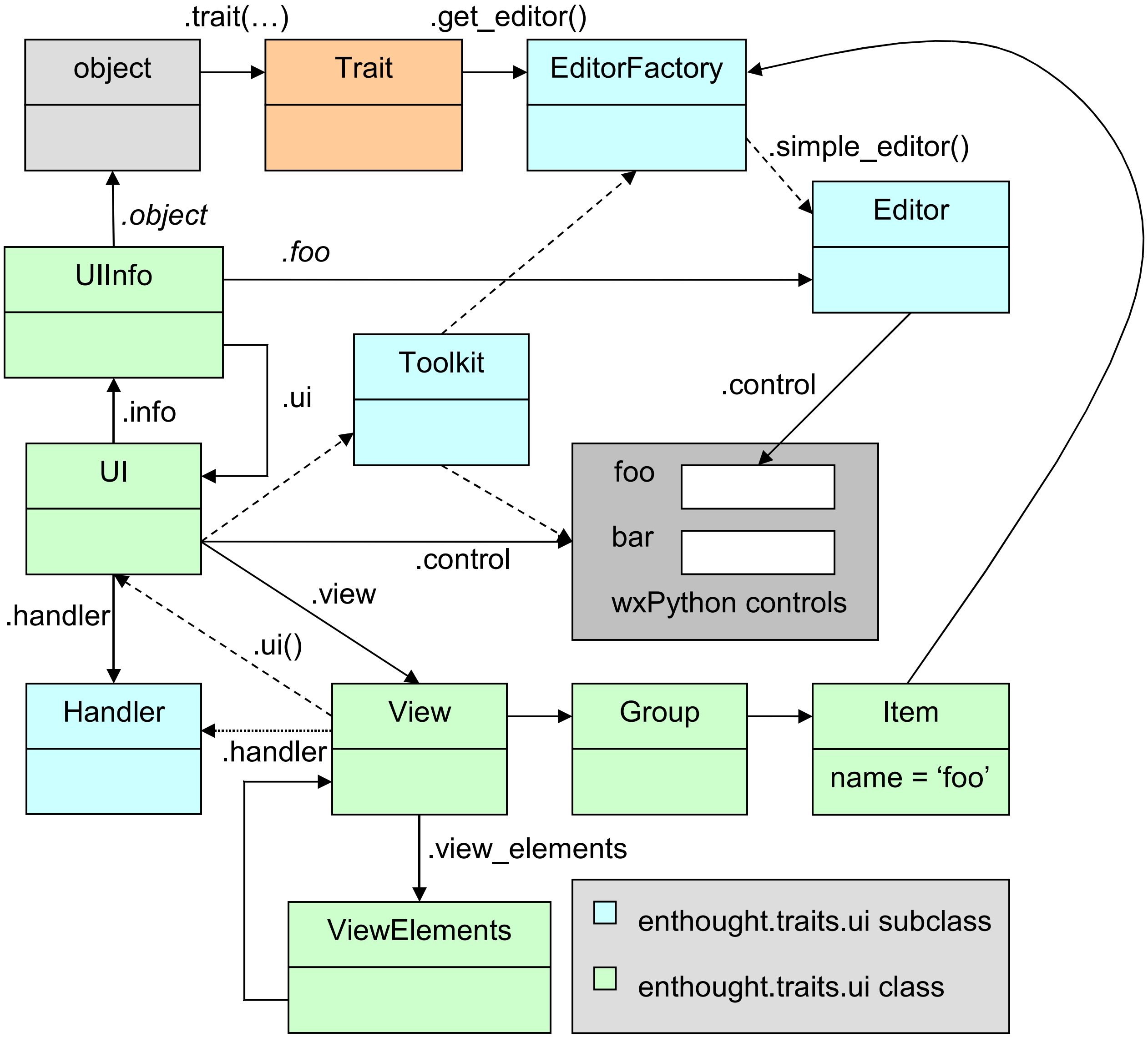
The UIInfo Object

- A **UIInfo** object is automatically created whenever a **View** is displayed.
- The **UIInfo** object is passed on every call to a **Handler** method (referred to as the ***info*** argument).
- The **UIInfo** object contains all the **View** specific information defined by the **View**.
- It is basically a *namespace* containing:
  - Each context object, referenced using its context name.
  - Each trait editor, referenced using its **Item *name*** or ***id***.
  - The traits UI created **UI** object, referenced using the name ‘**ui**’.
- It is your responsibility to ensure that context objects and editors don’t use duplicate names.

The UI Object

- A **UI** object is also created automatically each time a traits UI **View** is displayed.
- The **UI** object contains all the traits UI information common to every **View**.
- The **UI** object is returned as the result of a call to the **edit_traits**, **configure_traits** or **ui** method (on a **View** object).
- Although the **UI** object contains lots of information, the parts of most interest to a traits UI developer are:
  - **result**: A boolean value indicating the result of a modal dialog (i.e. **True** = user clicked OK; **False** = user clicked Cancel).
  - **dispose()**: A method that can be used to close the dialog or window under program control.

The Traits UI Object Model



What Editors Do

- Editors are the heart of the traits UI.
- Editors create a UI toolkit specific user interface for displaying and entering a specific type of data (e.g. floats, colors, fonts, file names, …)
- Every trait has a default editor associated with it.
- However, the default editor can be overridden either in the trait definition or in a view **Item** definition.

What Editors Do

- Editors and traits are loosely coupled: the editor only ensures that the type of data it understands is entered, and relies on the associated trait to actually validate the data entered.
- In some cases, the data allowed by the editor is a subset of the data allowed by the trait, so no user errors can occur.
- In other cases, the editor allows a superset of the data allowed by the trait, and catches exceptions thrown by the trait when invalid values are entered. The editor then provides feedback to the user to indicate that the entered value is not valid (e.g. the entry field turns red).

Editors and Editor Factories

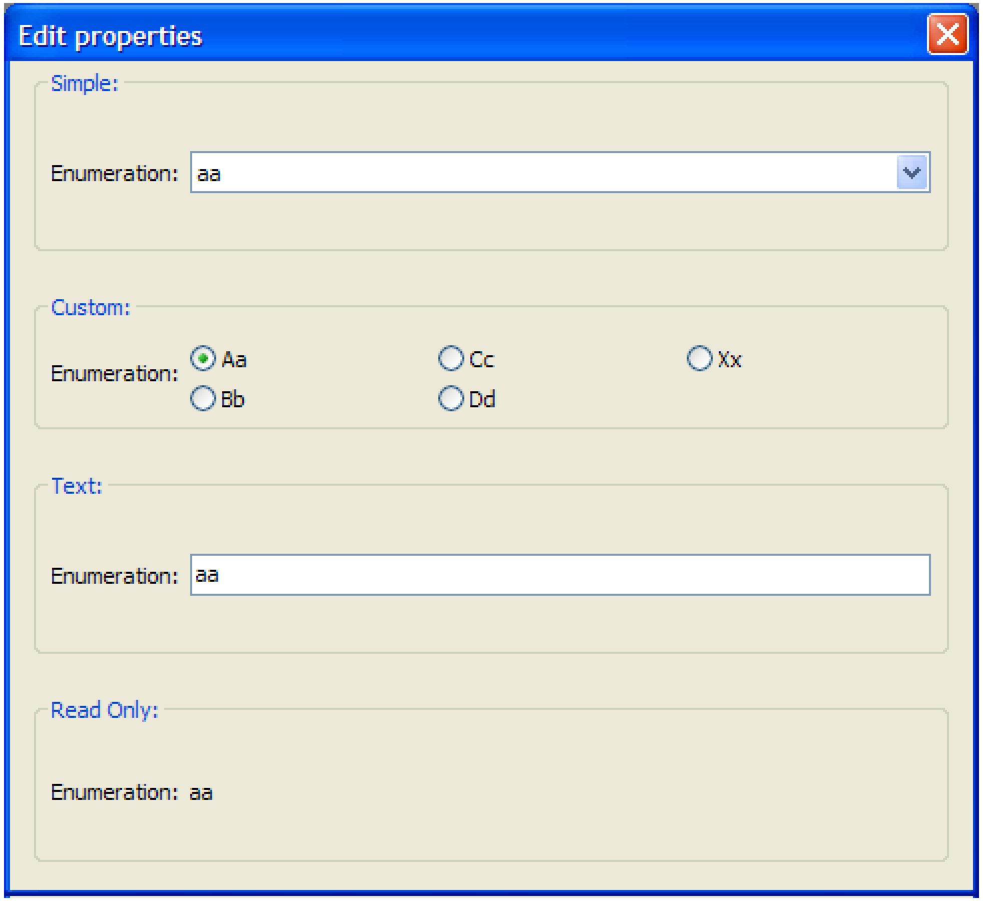
- What we call an ***editor*** is in fact an ***editor factory*** (but editor is a less intimidating term).
- An editor factory can be thought of as a template for creating the real editors when needed (i.e. when a View is displayed).
- Because they are templates, the same editor factory object can often be re-used in multiple views, or even multiple times within the same view.
- When a View is displayed, the editor factory for each Item in the View is called to create an editor for that Item.

The Basic Editor Styles

- Editor factories can create any one of four different editor styles, based on the value of the corresponding **Item’s** ‘style’ trait.
- The four editor styles are:
  - **simple**: Fits on a single line. Can be used to create Visual Basic style property sheets.
  - **custom**: Provides the richest user experience, and can use as much screen real estate as necessary.
  - **text**: Fits on a single line, and is always a text entry field.
  - **readonly**: Displays the current value of a trait, but does not allow the user to edit it.

The Basic Editor Styles

For example, these are the four styles supported by the **EnumEditor**:



Simple:

Enumeration: aa

Custom:

Enumeration: ● Aa ○ Bb ○ Cc ○ Dd ○ Xx

Text:

Enumeration: aa

Read Only:

Enumeration: aa

The Standard Trait Editors

- The Traits UI package comes with a large set of predefined editor factories, and an open architecture that allows for creating new ones as needed.
- The current set of predefined editor factories are:

Boolean	Button	CheckList	Code
Color	Compound	Custom	Directory
Drop	EnableRGBA	Enum	File
Font	KivaFont	ImageEnum	Instance
List	Plot	Range	RGBColor
RGBAColor	Set	Table	Tabular
HTML	IEHTML	LED	AnimatedGIF
Image	DND	Drop	KeyBinding
Plot	Shell	Value	Flash
Text	Tree	Tuple	

The Standard Trait Editors

- In this presentation we’ll focus on six of the predefined editor factories…
- Three that are simple, yet very useful:
  - **ButtonEditor**
  - **CustomEditor**
  - **EnumEditor**
- And three that are extremely useful, but require more setup to use properly:
  - **InstanceEditor**
  - **TableEditor**
  - **TreeEditor**

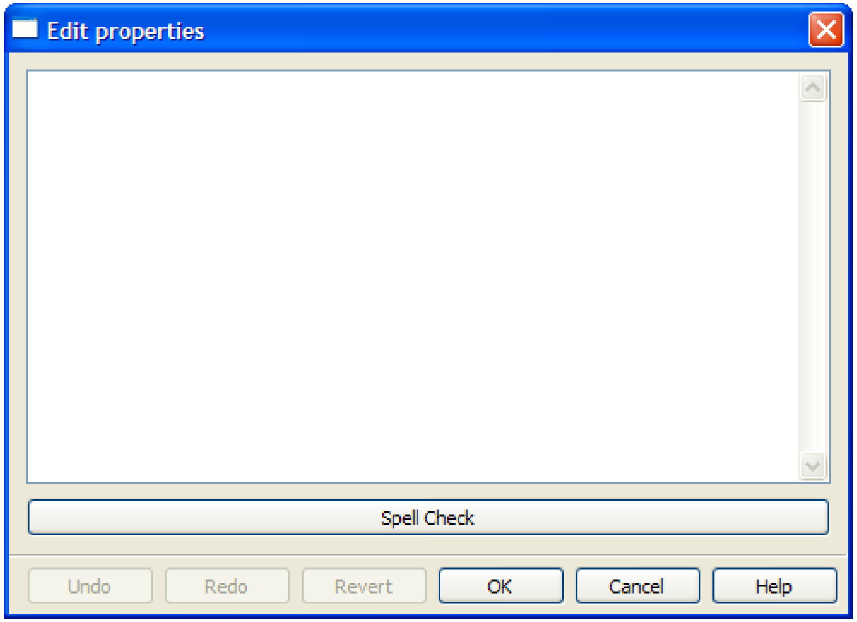
The ButtonEditor Editor Factory

- The ‘buttons’ trait of a **View** object allows you to define standard and custom buttons along the bottom edge of a window or dialog.
- However, there may be other cases where it would be useful to define buttons at other points in a view. This is where the **ButtonEditor** comes in handy.
- Traits defines a **Button** trait, which is an **Event** trait combined with a **ButtonEditor**. It is a parameterized type whose argument is the label you want to appear on the button in a view.

The ButtonEditor Editor Factory

For example:

```
class EMail ( HasTraits ):
    msg              = Str
    spell_check = Button( 'Spell Check' )

    view = View(Group(
                        Group (
                            Item('msg',
                                    style='custom',
                                    resizable=True),
                            Item('spell_check'),
                    show_labels=False)),
             height = .3 )
```



results in this display…

The ButtonEditor Editor Factory

- There are several ways to handle the button being clicked. Here’s one that treats ‘spell check’ as a model function:

```
class EMail ( HasTraits ):
    msg             = Str
    spell_check = Button( 'Spell Check' )

    view = View( <see previous slide> )

    def _spell_check_fired ( self ):
        # Perform spell checking...
```

The ButtonEditor Editor Factory

- Here’s another that treats it as a view/controller function:

```
class EMailHandler ( Handler ):
    spell_check = Button( 'Spell Check' )

    def handler_spell_check_changed ( self, info ):
        if info.initialized:
            # Perform spell checking...

    view = View( <see earlier slide> )

class EMail ( HasTraits ):
    msg = Str
```

The CustomEditor Editor Factory

- While the Traits UI can handle most user interface requirements, occasionally there are cases where it is useful to imbed a non-traits widget in the middle of a traits View.
- One solution is to write a new traits EditorFactory subclass that creates the needed widget.
- However, in many “one of” cases it may be faster and easier to simply use a CustomEditor to imbed the “foreign” control into a particular View.

The CustomEditor Editor Factory

- Using a **CustomEditor** requires specifying a callable function plus any additional arguments the function may require. The signature for the function must be:

function(parent, editor, args, …)

where:

- **parent**: The parent window for the custom widget
- **editor**: The **CustomEditor** instance being used to create the custom widget
- **args**: Any additional arguments needed by the function to create the custom widget

The function must return the custom widget it creates, created as a child of the **parent** window.

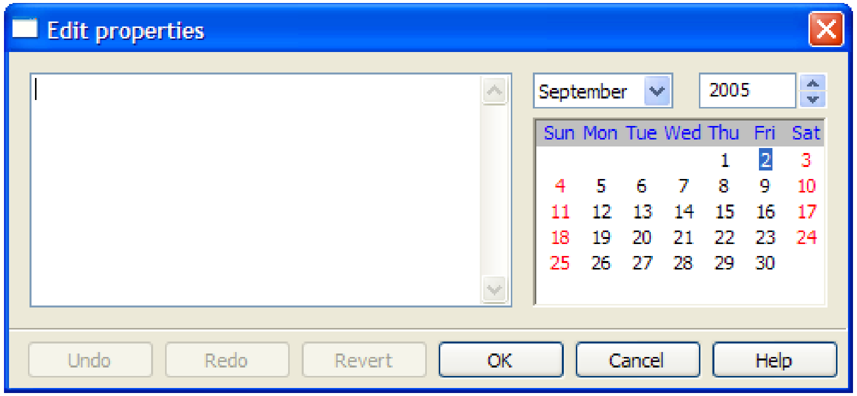
The CustomEditor Editor Factory

- The following is an example of using a **CustomEditor** to create a view:

```
def make_calendar ( parent, editor ):
    import wx
    import wx.calendar
    return wx.calendar.CalendarCtrl( parent, -1, wx.DateTime_Now() )

class Appointment ( HasTraits ):
    description = Str
    date        = Str

    view = View(Group (Group (Group (Item( name='description',
                                           style='custom',
                                           resizable=True),
                                     show_labels=False),
                              Group (Item( name='date',
                                           editor = CustomEditor( make_calendar )),
                                     show_labels=False),
                              orientation='horizontal')),
                height = 0.18)
```

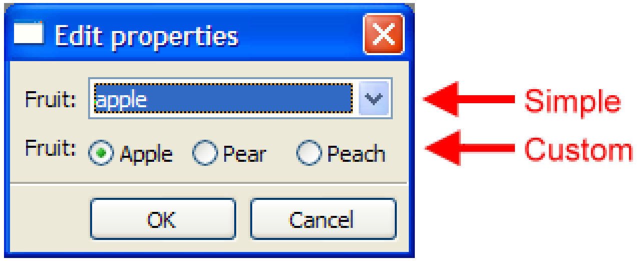
The CustomEditor Editor Factory

- Which results in the following display…
- Note that in practice, more code is required than this, since among other things, event handlers to handle input from the control and set the appropriate trait value also need to be set up.



The EnumEditor Editor Factory

- Let’s start with a trivial example of using an **EnumEditor**:
  - Fruit = Enum(‘apple’, ‘pear’, ‘peach’)
- Unsurprisingly, the **EnumEditor** is the default editor for the **Enum** trait, and will automatically yield the following results when used in a traits UI:



Simple

Custom

The EnumEditor Editor Factory

- Now, let’s make the example a little more “real world”…
- We’re doing an interactive menu, and ***fruit*** is a trait in an **Order** object that represents the diner’s choice from among the fruit currently on hand.
- Current stock is maintained in a separate **Stock** class instance that has a **fruits** trait that lists the fruit currently available.
- The diner should only be able to choose a fruit that is currently available.

```
class Order ( HasTraits ):
    fruit = Str    # A simple Enum won’t work anymore
    …

class Stock ( HasTraits ):
    fruits = List( Str )   # List of fruits on hand
    …
```

The EnumEditor Editor Factory

- We can still use the **EnumEditor** to create the UI, but we'll have to provide more information to help it tie things together.
- In this case, we'll focus on three of the **EnumEditor** traits that are of interest for this example:
  - **values**: The values to enumerate (can be a list, tuple, dict, or a **CTrait** or **TraitHandler** than is *mapped*).
  - **name**: Extended trait name of the trait containing the enumeration data.
- The **values** and **name** traits provide complementary means of accomplishing the same task: providing the set of enumeration values independently of the trait being edited.
- In this case, we'll use the **name** trait because we have access to a **Stock** object whose **fruits** trait contains the enumeration of available fruit.

The EnumEditor Editor Factory

- The resulting **Order** view would then look something like:

```
class Order ( HasTraits ):
    fruit = Str
    …
    view = View( …,
                Item( ‘fruit’,
                        editor = EnumEditor(name = ‘stock.fruits’ ) ),
                … )
```

- This view relies on two objects: the **Order** being edited, and a **Stock** object containing the available fruits, referred to as ‘stock’ in the above view.
- This requires providing a context containing both objects when the **Order** view is displayed:

```
order.edit_traits( context = { ‘object’: order, ‘stock’: current_stock,
                   view = ‘view’ )
```

The InstanceEditor Editor Factory

- The following type of trait declaration occurs frequently:
  - manager = Instance(Employee)
- Amazingly enough, the **InstanceEditor** is designed to edit these types of traits.
- There are multiple usage scenarios for editing this type of trait though:
  - The instance is fixed, but the user needs to edit the contents of the instance.
  - The user needs to select from a fixed or varying set of instances, but does not need to modify the contents of the instance once selected.
  - The user needs to select or create an instance, and then be able to edit the contents of the selected instance.
- The **InstanceEditor** is designed to handle all of these scenarios, along with several variations on how the editing should be performed.
- However, in order to do this, the **InstanceEditor** requires a more complex definition than do most other trait editors.

The InstanceEditor Editor Factory

- Let’s start with the easiest case:
  - The user only needs to edit the contents of the current instance object.
- In this case, there are two choices:
  1. Allow the user to edit the object contents in a separate pop-up dialog.
  2. Edit the contents of the instance object “in-line”, as if it were part of the main object being edited.

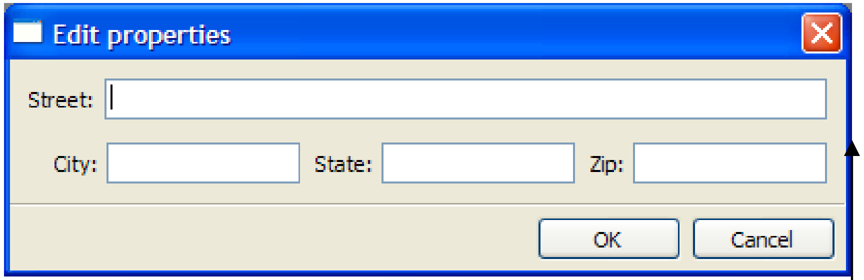
The InstanceEditor Editor Factory

- For case 1, use the “simple” editor style and specify some or all of the following traits:
  - **label**: The label on the button that displays the pop-up editor dialog. It defaults to the trait name.
  - **view**: The **View** object or name to display in the pop-up editor dialog. It defaults to the default **View** for the instance object.
  - **kind**: How the pop-up editor should be displayed (e.g. ‘*modal*’). It defaults to the value specified on the view itself.

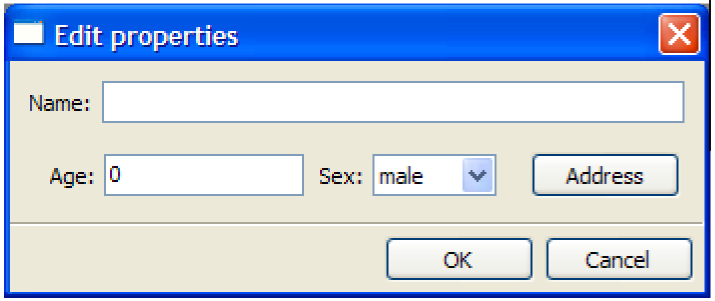
The InstanceEditor Editor Factory

For example:

```
class Address ( HasTraits ):
    street = Str
    city   = Str
    state  = Str
    zip    = Str

    view = View( [  [ 'street' ],
                    [ 'city', 'state', 'zip', orientation='horizontal'] ],
                 buttons = [ 'OK', 'Cancel' ] )
```



```
class Person ( HasTraits ):
    name    = Str
    age     = Int
    sex     = Enum( 'male', 'female' )
    address = Instance( Address, () )

    view = View(  [  [ 'name' ],
                     [  [ 'age', 'sex', orientation='horizontal' ],
                        [ 'address', show_labels=False],
                        orientation='horizontal' ] ],
                 buttons = [ 'OK', 'Cancel' ] )
```



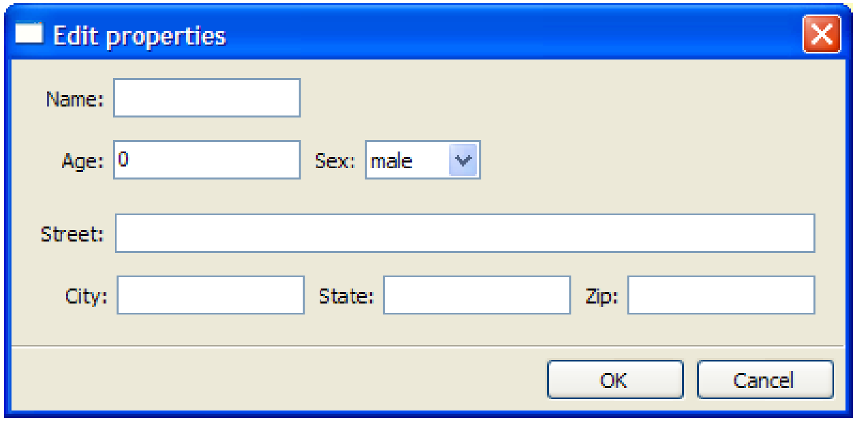
Age: 0

Sex: male

The InstanceEditor Editor Factory

For case 2, use the “custom” editor style and ignore the **label** trait:

```
class Address ( HasTraits ):
    street = Str
    city   = Str
    state  = Str
    zip    = Str

    view = View( [ [ 'street' ], [ '13', 'city', 'state', 'zip', '-' ] ],
                 buttons = [ 'OK', 'Cancel' ] )


class Person ( HasTraits ):
    name    = Str
    age     = Int
    sex     = Enum( 'male', 'female' )
    address = Instance( Address, () )

    view = View( [ [ '9', 'name', '-' ],
                   [ '17', 'age', 'sex', '-' ],
                   [ Item( 'address', style = 'custom' ), '-<>' ] ],

                 buttons = [ 'OK', 'Cancel' ] )
```



Edit properties

Name:

Age: 0

Sex: male

Street:

City:

State:

Zip:

The InstanceEditor Editor Factory

- Now let’s move on to the next case:
  - The user needs to select from a fixed or varying set of instances, but does not need to modify the contents of the instance once selected.
- Within this case there are several important sub-cases:
  - The user must select from a known set of existing instances. The set may change over time.
  - The user must select from several different types (i.e. classes) of objects, which are only created once the user selects them.
  - The user must select from an unknown set of existing instances (e.g. by drag and drop).

The InstanceEditor Editor Factory

- The **InstanceEditor** supports all of these sub-cases, and in fact allows any combination of them to be used together in the same editor.
- It does this by allowing you to specify one or more **InstanceChoiceItem** subclasses as part of the **InstanceEditor** definition.
- There are three predefined **InstanceChoiceItem** subclasses, each handling one of the three previously described sub-cases:
  - **InstanceChoice**: Describes a single instance object the user can select. Note: If an instance has a suitable **name** trait, the instance can be used instead of an **InstanceChoice** object.
  - **InstanceFactoryChoice**: Describes a '*factory*' (e.g. a class) which can create instance objects the user can select.
  - **InstanceDropChoice**: Describes a class of object the user can drag and drop on the editor to select it.

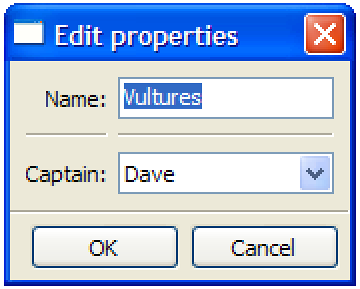
The InstanceEditor Editor Factory

- The list of **InstanceChoiceItems** can either be specified as part of the **InstanceEditor** itself, using the **values** trait, or as an external model, specified using the **name** trait.
- Or they can be used together to create a composite set.
- In any case, changes made to the set of **InstanceChoiceItems** are immediately reflected in the **InstanceEditor** user interface.

The InstanceEditor Editor Factory

For example:

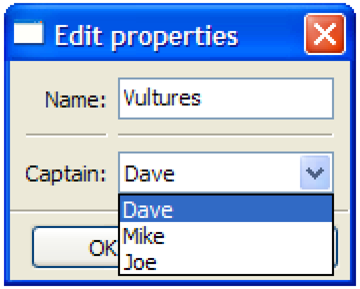
```
class Person ( HasTraits ):
    name  = Str
    age     = Int
    phone = Regex( value = '000-0000',
                         regex = '\d\d\d[-]\d\d\d\d' )

    view = View( 'name', 'age', 'phone' )

class Team ( HasTraits ):
    name    = Str
    captain = Instance( Person )
    roster   = List( Person )

    view = View( 'name', '_',
                    Item( 'captain',
                    editor = InstanceEditor( name = 'roster', editable = False ) ),
                    buttons = [ ‘OK’, ‘Cancel’ ] )
```



Edit properties

Name: Vultures

Captain: Dave



Edit properties

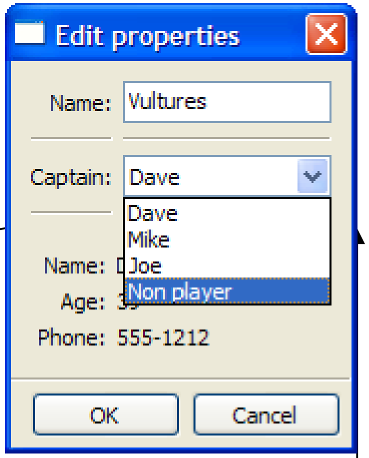
Name: Vultures

Captain: Dave

The InstanceEditor Editor Factory

And a slightly more complex example…

```
class Person ( HasStrictTraits ):
    name  = Str
    age     = Int
    phone = Regex( value = '000-0000', regex = '\d\d\d[-]\d\d\d\d' )

    traits_view = View( ['name', 'age', 'phone', style='readonly'] )
    edit_view   = View( ['name', 'age', 'phone'],
                             buttons = [ 'OK', 'Cancel' ]  )

class Team ( HasStrictTraits ):
    name    = Str
    captain = Instance( Person )
    roster   = List( Person )

   traits_view = View(
                 [ [ Item('name'),
                     Item('_'),
                     Item( 'captain',
                           editor = InstanceEditor(
                                        name = 'roster',
                                        editable = False,
                                        values   = [ InstanceFactoryChoice(
                                            klass = Person,
                                            name  = 'Non player',
                                            view  = 'edit_view' ) ] ) ),
                     Item( '_') ],
                 [ Item('captain', style='custom') ],
                 buttons = [ 'OK', 'Cancel' ] )
```



Name: Vultures

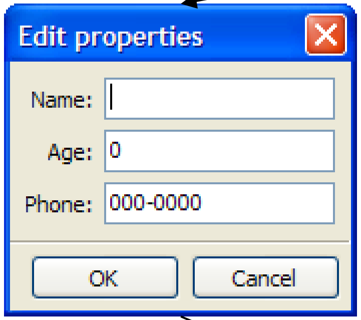
Captain: Dave

Dave
Mike
Joe
Non player

Name:

Age:

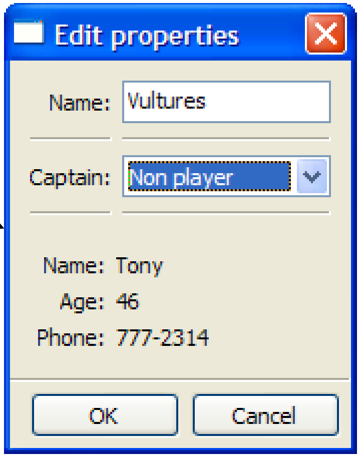
Phone: 555-1212



Name:

Age: 0

Phone: 000-0000



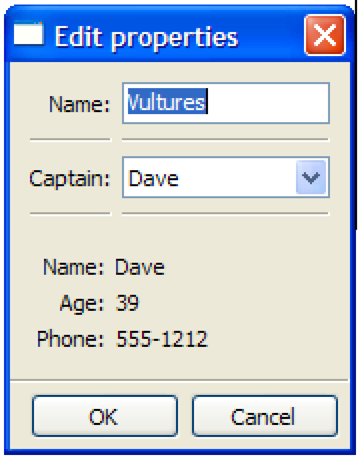
Name: Vultures

Captain: Non player

Name: Tony

Age: 46

Phone: 777-2314



Name: Vultures

Captain: Dave

Name: Dave

Age: 39

Phone: 555-1212

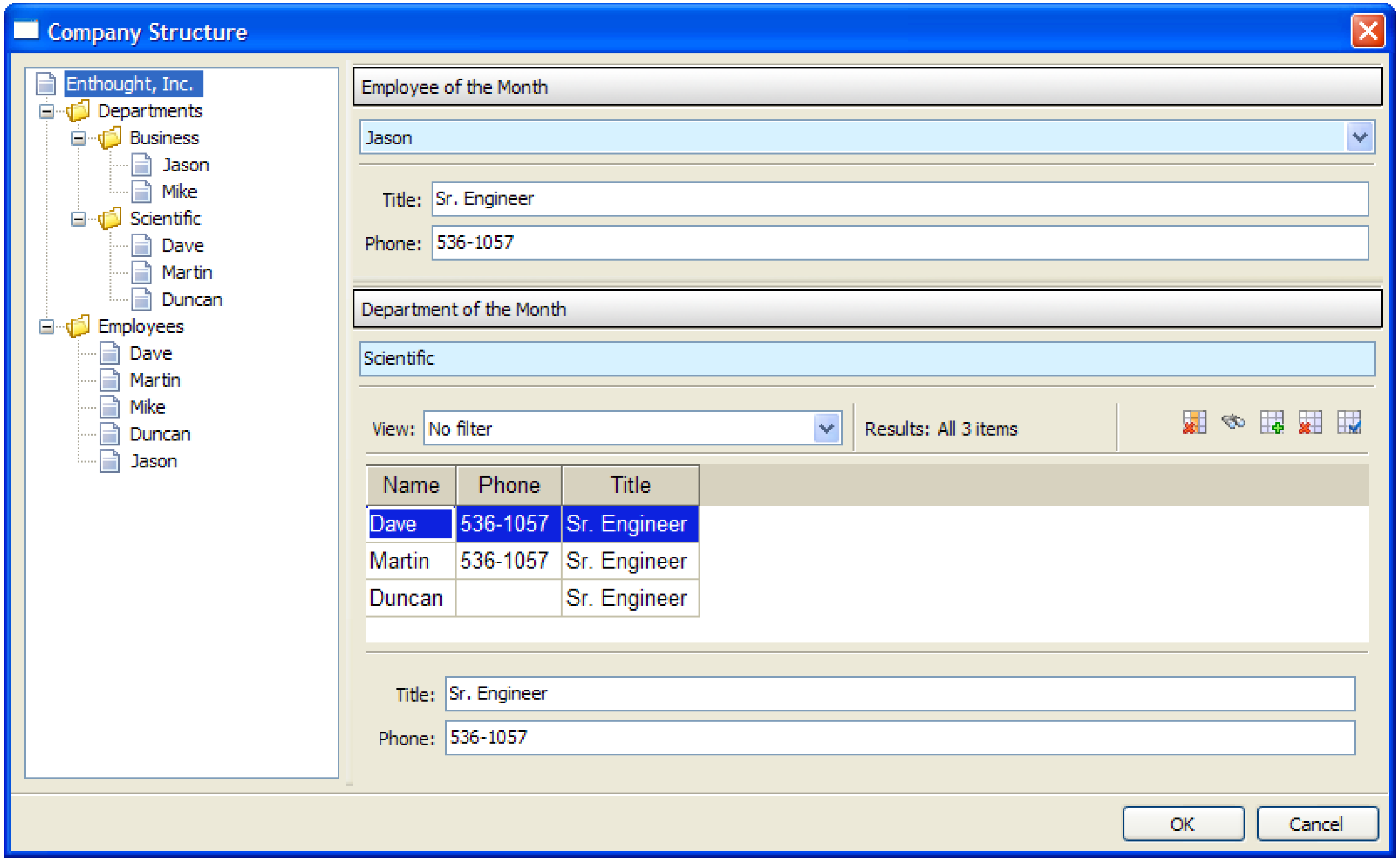
The InstanceEditor Editor Factory

And here’s an example showing “drag and drop” support:

```
view = View(
            Group(
                 Group( Item( 'company',
                                  editor = tree_editor,
                                  resizable = True )
                          show_labels=False),
            Group(
                 Group( Item( label ='Employee of the Month', style = 'custom'),
                           Item( 'eom',
                                  editor = InstanceEditor( values = [
                                                                     InstanceDropChoice( klass = Employee,
                                                                                                  selectable = True ) ] ),
                                  style='custom', resizable = True )
                           show_labels = False),

                Group( Item( label = 'Department of the Month}', style='custom'),
                          Item( 'dom',
                                 editor = InstanceEditor( values = [
                                                                    InstanceDropChoice( klass = Department ) ] ),
                                 style='custom',
                                 resizable = True )),
                          show_labels = False, layout = 'split' ),
                orientation = 'horizontal', show_labels = False, layout = 'split' ),
            title = 'Company Structure', handler   = TreeHandler(),
            buttons = [ ‘OK’, ‘Cancel’ ],     resizable = True,
            width     = .5, height   = .5 )
```

The InstanceEditor Editor Factory

Which looks like:



Employee of the Month

Jason

Title: Sr. Engineer

Phone: 536-1057

Department of the Month

Scientific

View: No filter

Results: All 3 items

Name	Phone	Title
Dave	536-1057	Sr. Engineer
Martin	536-1057	Sr. Engineer
Duncan		Sr. Engineer

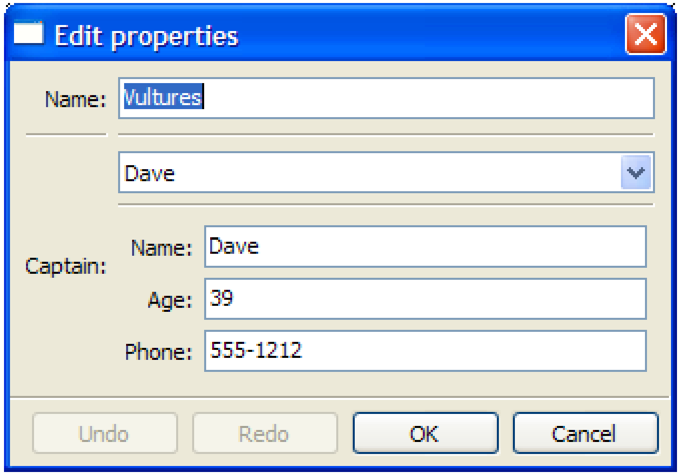
Title: Sr. Engineer

Phone: 536-1057

The InstanceEditor Editor Factory

Finally, it is possible to combine instance selection and editing in a single editor by simply combining the editing and selection traits:

```
class Person ( HasTraits ):
    name  = Str
    age     = Int
    phone = Regex( value = '000-0000',
                              regex = '\d\d\d[-]\d\d\d\d' )

   traits_view = View( 'name', 'age', 'phone' )

class Team ( HasTraits ):
    name    = Str
    captain = Instance( Person )
    roster   = List( Person )

    traits_view = View( Item('name'),
                              Item('_'),
                              Item( 'captain',
                                     editor = InstanceEditor( name = 'roster' ),
                                     style='custom'),
                              buttons = [ 'Undo', 'OK', 'Cancel' ] )
```



Name: Vultures

Dave

Captain:
Name: Dave
Age: 39
Phone: 555-1212

The TableEditor Editor Factory

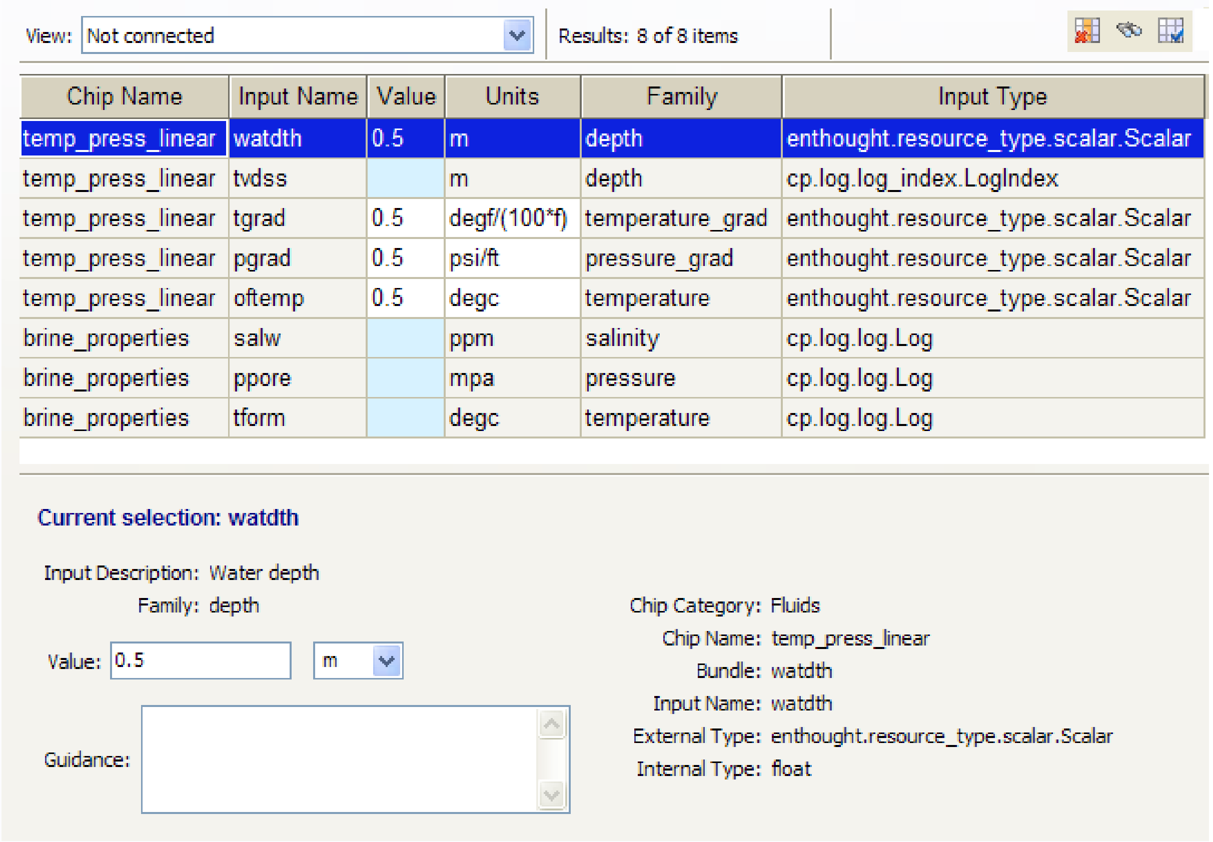
- Another common type of trait declaration is:
  - department = List(Employee)
- In the case of a list of objects with traits, the **TableEditor** can be used to display the list as a table, with one object per row, and one object trait per column.
- In fact, the **TableEditor** is the default editor for a **List** trait whose values are objects with traits.

The TableEditor Editor Factory

- Some of the features of the **TableEditor** are:
  - Supports ‘editable’ and ‘read-only’ modes.
  - Allows object editing either in-place within the table, or separately, in an external ‘inspector’ view.
  - In-place editing supports many of the common trait editors, including drag and drop.
  - Supports ascending/descending sorting on any column.
  - Sorting can either affect the underlying model or just the view.
  - Allows the user re-order/include/exclude any of the object columns, and persist the resulting set across application sessions.
  - Allows searching the table contents in a wide variety of ways.
  - Allows filtering the table contents using a wide variety of user customizable and persistable filters.
  - Table/column/cell level context menus
  - All changes made to the table are fully undoable/redoable.
  - Colors, fonts and grid lines are fully customizable.

The TableEditor Editor Factory

An example of a **TableEditor**:



View: Not connected

Results: 8 of 8 items

Chip Name	Input Name	Value	Units	Family	Input Type
temp_press_linear	watdth	0.5	m	depth	enthought.resource_type.scalar.Scalar
temp_press_linear	tvdss		m	depth	cp.log.log_index.LogIndex
temp_press_linear	tgrad	0.5	degf/(100*f)	temperature_grad	enthought.resource_type.scalar.Scalar
temp_press_linear	pgrad	0.5	psi/ft	pressure_grad	enthought.resource_type.scalar.Scalar
temp_press_linear	oftemp	0.5	degc	temperature	enthought.resource_type.scalar.Scalar
brine_properties	salw		ppm	salinity	cp.log.log.Log
brine_properties	ppore		mpa	pressure	cp.log.log.Log
brine_properties	tform		degc	temperature	cp.log.log.Log

Current selection: watdth

Input Description: Water depth
Family: depth

Value: 0.5 m

Guidance:

Chip Category: Fluids
Chip Name: temp_press_linear
Bundle: watdth
Input Name: watdth
External Type: enthought.resource_type.scalar.Scalar
Internal Type: float

The TableEditor Editor Factory

- All of these features and flexibility come at a price…in this case, the amount of work needed to *correctly* define a **TableEditor**.
- Out of the box, a **TableEditor** will display many lists of objects without any extra work…but the results will often be non-optimal.
- The traits that can be defined for a **TableEditor** include:
  - **Table Attributes**: Colors, fonts, sorting rules, …
  - **Table Columns**: What object traits can be displayed as columns and how.
  - **Table Filters**: What standard and custom filters can be applied to the table.
  - **Table Search**: What filter can be used to search the table.
  - **Table Factory**: An optional callable that can be used to add new object rows to the table.

The TableEditor Editor Factory: Table Attributes

auto_size	edit_view_handler	menu	selection_bg_color
cell_bg_color	edit_view_height	on_dclick	selection_color
cell_color	edit_view_width	on_select	show_column_labels
cell_font	editable	orientation	show_lines
cell_read_only_bg_color	filter	other_columns	sort_model
column_label_height	filters	row_factory	sortable
columns	label_bg_color	row_factory_args	auto_add
configurable	label_color	row_factory_kw	reverse
deletable	label_font	row_label_width	selected
reorderable	rows	columns_name	row_height
edit_view	line_color	search	

The TableEditor Editor Factory: Table Columns

- You define the content, appearance and behavior of a table by providing ordered sets of **TableColumn** objects.
- Each **TableColumn** object describes a single column/object trait.
- You can provide two sets of columns:
  - **columns**: The columns you see initially
  - **other_columns**: The remaining columns
- These are the default columns, the user's most recent preference setting overrides them.
- There are two basic types of **TableColumn**:
  - **ObjectColumn**: Used for objects with traits
  - **ListColumn**: For lists and tuples
- You can also define subclasses to get state dependent column behavior

The TableEditor Editor Factory: Table Columns

ObjectColumn traits:

- **name**: Name of the object trait to display/edit
- **label**: Column label to use for the column
- **type**: The type of data contained by the column
- **text_color**: Text color for this column
- **text_font**: Text font for this column
- **cell_color**: Cell background color for this column
- **read_only_cell_color**: Cell background color for non-editable columns
- **horizontal_alignment**: Horizontal alignment of text in the column
- **vertical_alignment**: Vertical alignment of text in the column
- **visible**: Is the table column visible (i.e. viewable)?
- **editable**: Is this column editable?
- **droppable**: Can external objects be dropped on the column?
- **editor**: Editor factory to use when editing the column “in-place”
- **menu**: Context menu to display when this column is right-clicked

The TableEditor Editor Factory: Table Columns

- For almost every **ObjectColumn** trait there is a corresponding “get” or “is” method.
  - For example, “editor” and “get_editor()”, “editable” and “is_editable()”.
- Defining a method overrides the corresponding trait.
- This allows subclasses to define values that are dependent upon the state of each table object or other values.

The TableEditor Editor Factory: Table Filters

- When applied to a table, a filter reduces the set of visible rows to only those objects which match the filter’s criteria.
- A **TableEditor** defines two filter related traits:
  - **filter**: The filter initially in effect (defaults to **None** = “No filter”)
  - **filters**: A list of **TableFilter** objects that the user can choose from using the “View” drop down list.

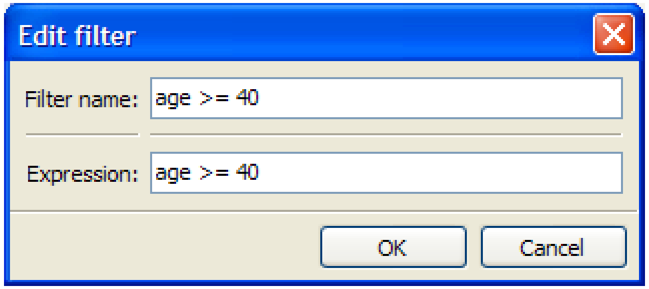
The TableEditor Editor Factory: Table Filters

- There are two basic types of filters:
  - **Normal filter**: An actual filter that can be applied and modified.
  - **Template filter**: A filter which cannot be applied or modified, but which is used to create new normal filter objects of the same type as the template and with the same initial filter values.
- A template filter is simply a normal filter with its ***template*** trait set to **True**.
- User created filters are automatically persisted across application sessions as part of the **TableEditor**'s user preference handling.
- You can create your own **TableFilter** subclasses, or use any of the standard subclasses:
  - **EvalTableFilter**
  - **RuleTableFilter**
  - **MenuTableFilter**

The TableEditor Editor Factory: Table Filters

The **EvalTableFilter**:

- Allows a user to enter a Python expression whose value determines whether or not an object meets the filter criteria.
- Its use is obviously best suited to users already familiar with Python.
- Trait references on the object being tested do not need to be explicitly qualified.



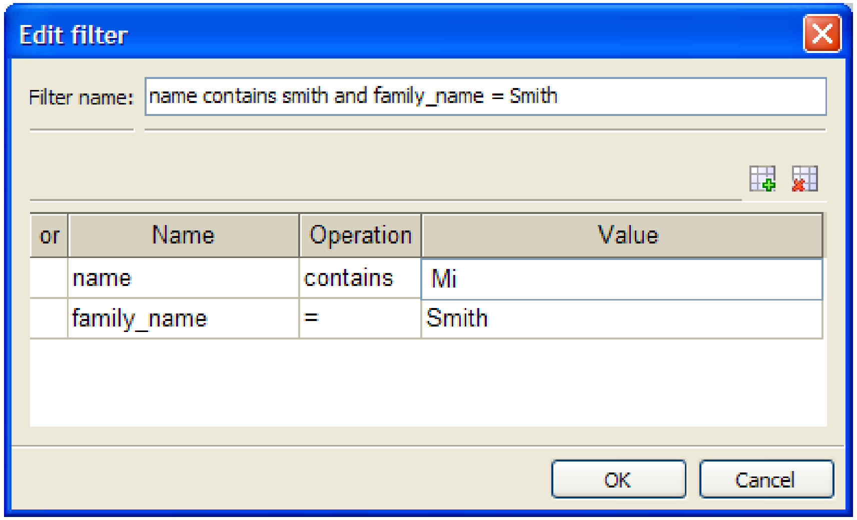
Filter name: age >= 40

Expression: age >= 40

The TableEditor Editor Factory: Table Filters

The **RuleTableFilter**:

- Allows users to define “rules” using drop down value entry for trait names and operations.
- Rules can be “and”ed or “or”ed together.
- Rules can be added, deleted and modified.
- Introspection based… requires no set-up by the developer.



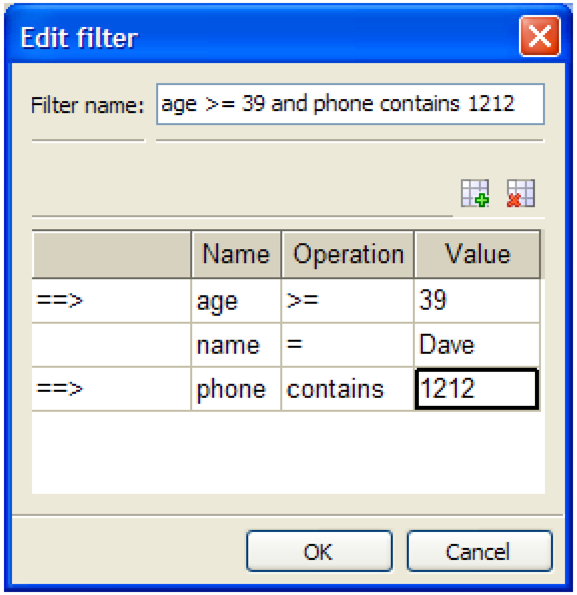
Filter name: name contains smith and family_name = Smith

or	Name	Operation	Value
	name	contains	Mi
	family_name	=	Smith

The TableEditor Editor Factory: Table Filters

The **MenuTableFilter**:

- Is similar to the **RuleTableFilter**
- The differences are:
  - A rule is automatically created for each object trait.
  - Rules cannot be added or deleted.
  - Rules are implicitly "and"ed together.
  - Rules can be turned on and off.

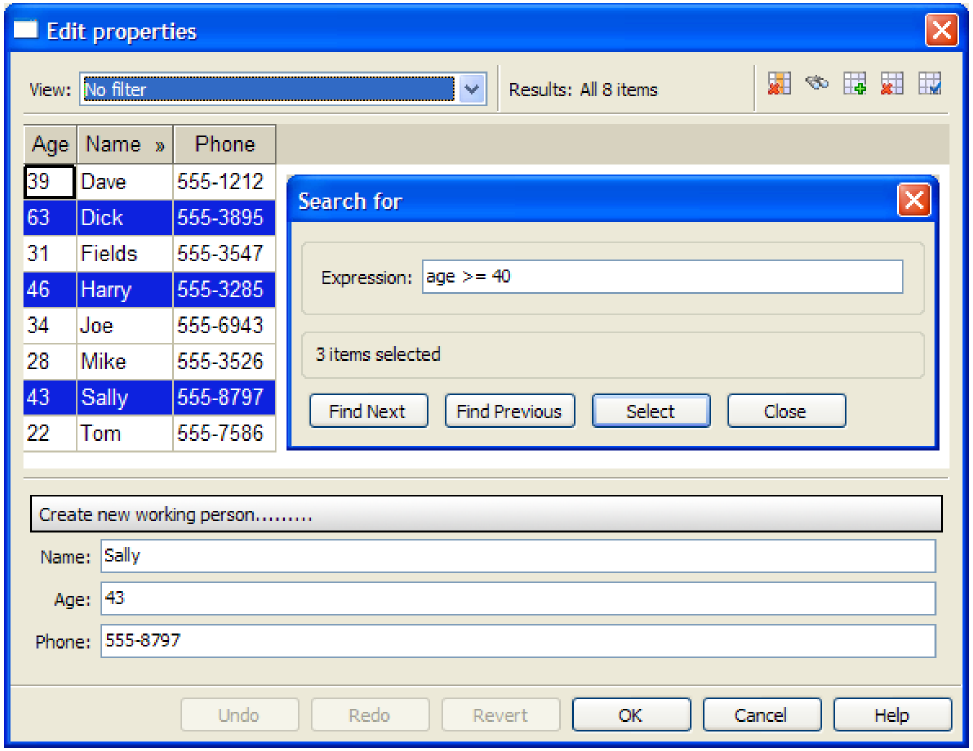


Filter name: age >= 39 and phone contains 1212

	Name	Operation	Value
==>	age	>=	39
	name	=	Dave
==>	phone	contains	1212

The TableEditor Editor Factory: Table Search

- Making a table searchable allows the user to search for (and optionally select) object rows which match a specified search criteria.
- A table is made searchable by setting the **TableEditor's** search trait to a **TableFilter** object.
- Doing so adds a "search" icon to the table's toolbar, which displays a pop-up search dialog.
- The search dialog allows the user to search for the next or previous match, or to select all matching rows.



View: No filter

Results: All 8 items

Age	Name »	Phone
39	Dave	555-1212
63	Dick	555-3895
31	Fields	555-3547
46	Harry	555-3285
34	Joe	555-6943
28	Mike	555-3526
43	Sally	555-8797
22	Tom	555-7586

Expression: age >= 40

3 items selected

Create new working person.........

Name: Sally

Age: 43

Phone: 555-8797

The TableEditor Editor Factory: Table Factory

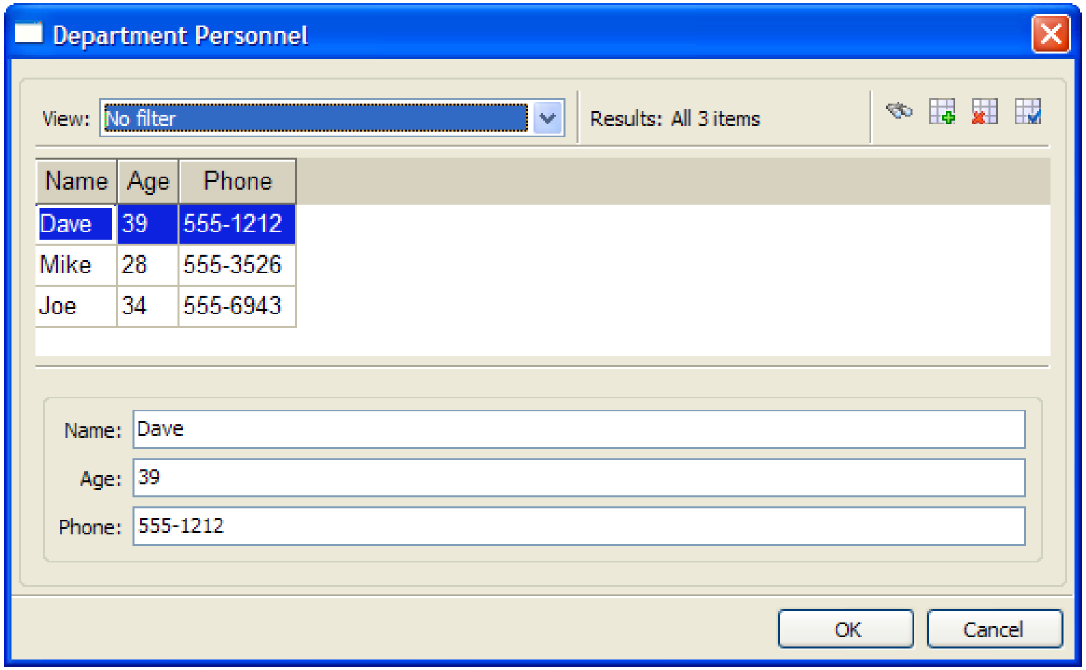
- If users can add new rows to a table, then a *factory* must be provided to create the new table objects.
- The **TableEditor** traits that specify the object factory are:
  - **row_factory**: A callable that creates and returns a new object instance when the user adds a new row to the table.
  - **row_factory_args**: An optional tuple that contains any positional arguments that need to be passed to the factory.
  - **row_factory_kw**: An optional dictionary that contains any keyword arguments that need to be passed to the factory.

The TableEditor Editor Factory: An Example

```
class Employee ( HasTraits ):
   name  = Str
   age     = Int
   phone = Regex( value = '000-0000', regex = '\d\d\d[-]\d\d\d\d'
   traits_view = View( 'name', 'age', 'phone',
                                  title = 'Create new employee', width = 0.18, buttons = [ ‘OK’,
‘Cancel’ ] )


table_editor = TableEditor(
   columns       = [ ObjectColumn( name = 'name' ),
                        ObjectColumn( name = 'age' ),
                        ObjectColumn( name = 'phone' ) ],
   deletable      = True,   sort_model  = True,   orientation = 'vertical',
   edit_view     = View( Group( 'name', 'age', 'phone', show_border=True ],
                                resizable = True ),
   filters           = [ EvalFilterTemplate, MenuFilterTemplate, RuleFilterTemplate ],
   search         = RuleTableFilter(),
   row_factory = Employee )


class Department ( HasStrictTraits ):
   employees = List( Employee )
   traits_view = View( Group( Item( 'employees',
                                              editor = table_editor,
                                              resizable = True ),
                                      show_border = True,
                                      show_labels = False),
                             title = 'Department Personnel', width = .4, height = .4,
                             resizable = True,  buttons   = [ ' OK', 'Cancel' ], kind = 'live' )
```

The TableEditor Editor Factory: An Example

The resulting view looks like:



Department Personnel

View: No filter

Results: All 3 items

Name	Age	Phone
Dave	39	555-1212
Mike	28	555-3526
Joe	34	555-6943

Name: Dave

Age: 39

Phone: 555-1212

The TreeEditor Editor Factory

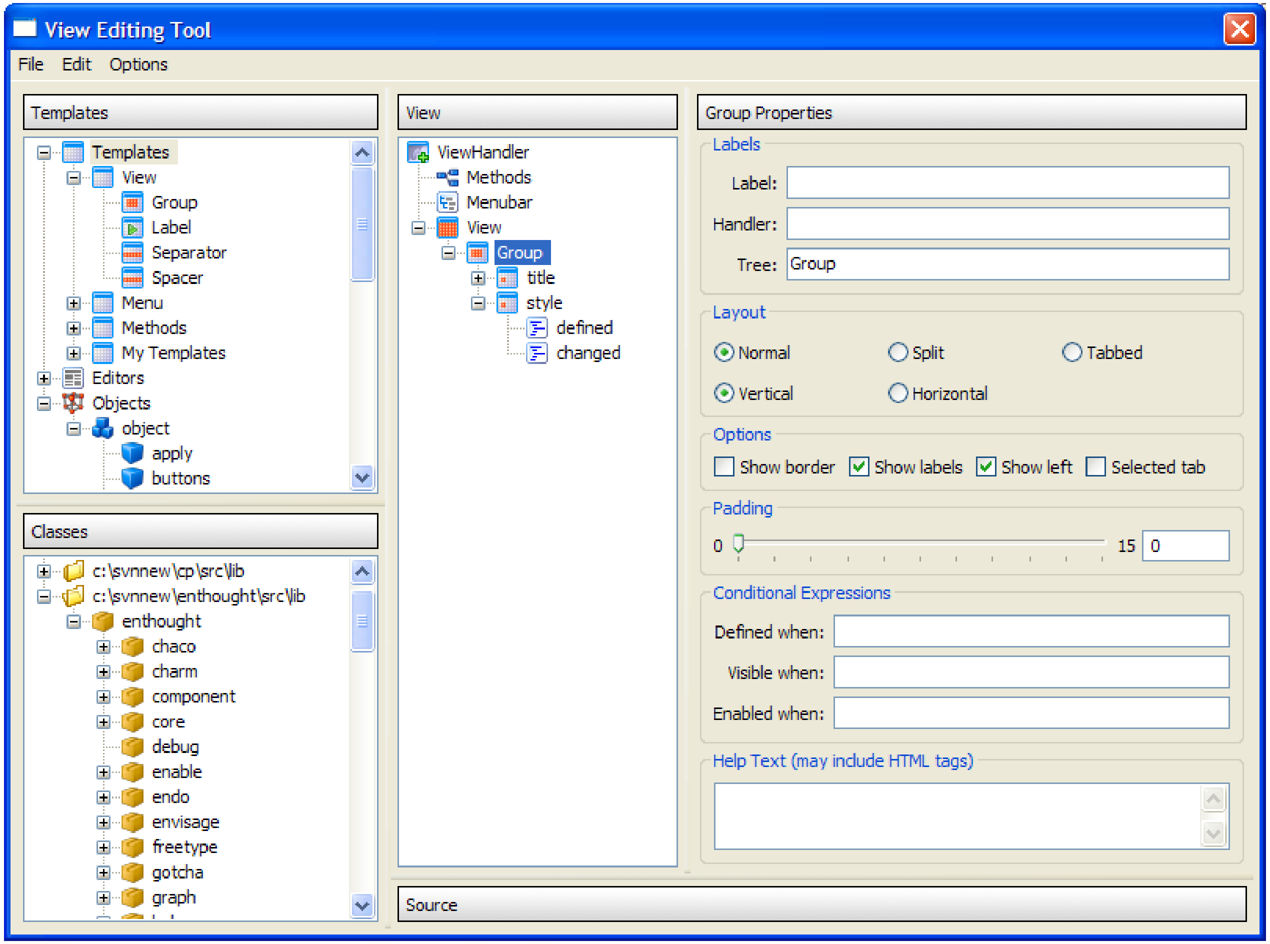
- Objects often are connected together in such a way that a hierarchical or tree view of them is a useful user interface feature.
- Common examples:
  - File system explorer: files are contained in directories, which are contained in other directories…
  - Organizational chart: Employees belong to departments, which in turn belong to the company itself.
- A **TreeEditor** allows interconnected objects with traits to be displayed as a tree.

The TreeEditor Editor Factory

- Using a **TreeEditor**, each connected object in the graph of objects to be displayed becomes a separate tree item.
- Some of the features supported by the **TreeEditor** include:
  - Full drag and drop support:
    - Objects can be dragged into the tree.
    - Objects can be dragged out of the tree.
    - Objects can be dragged within the tree.
    - Structural relationships between objects are enforced.
  - Objects can be:
    - Added
    - Deleted
    - Renamed
  - The contents of objects can be edited in a separate “inspector” view if desired.
  - Each object can have a standard or custom context menu.

The TreeEditor Editor Factory

Here are some examples of **TreeEditors** being used in the VET tool:



Labels

Label:

Handler:

Tree: Group

Layout

● Normal ○ Split ○ Tabbed

● Vertical ○ Horizontal

Options

☐ Show border ☑ Show labels ☑ Show left ☐ Selected tab

Padding

0 — 15: 0

Conditional Expressions

Defined when:

Visible when:

Enabled when:

Help Text (may include HTML tags)

The TreeEditor Editor Factory

- Like the **TableEditor**, in order to provide such a rich set of user interactions, the **TreeEditor** needs additional information to perform its task.
- The extra information is divided into two parts:
  - General information about the editor's behavior defined by traits on the **TreeEditor** itself.
  - Specific information about each of the object types that can appear in the tree, provided by a set of **TreeNode** objects associated with the **TreeEditor**.

The TreeEditor Editor Factory

The general **TreeEditor** traits are:

nodes	Supported TreeNode objects
multi_nodes	(TreeNode,...) -> MultiTreeNode map
editable	Are the individual nodes editable?
shared_editor	Is the editor shared across trees?
editor	Ref to a shared object editor
show_icons	Should tree nodes have icons?
hide_root	Should the tree root node be hidden?
icon_size	Size of the tree node icons
on_select	Called when a node is selected
on_dclick	Called when a node is double clicked
orientation	Layout orientation of tree and editor

The TreeEditor Editor Factory

- A **TreeNode** provides information about one or more types (i.e. classes) of object that can appear in the tree:
  - The object classes the **TreeNode** applies to.
  - Which object trait (if any) contains the *children* of the object.
  - Can the object's children be renamed, copied deleted or inserted?
  - What object class instances can be added to, copied to or moved to the children of the object.
  - The icons used to represent the object in the tree.
  - The context menu to display when the object is right-clicked on.
  - The view to display when the object is selected for editing.

The TreeEditor Editor Factory

The traits for a **TreeNode** are:

add	icon_item	move
auto_close	icon_group	name
auto_open	icon_open	node_for
children	icon_path	on_dclick
copy	insert	on_select
delete	label	rename
formatter	menu	view

The TreeEditor Editor Factory

- In addition, there are several different types of and ways to use **TreeNodes**:
  - If all the information about an object type is static, simply use a **TreeNode** and initialize its traits.
  - If some of the information is not known until run-time, create a subclass of **TreeNode** and override the necessary methods. Each trait has a corresponding method that can be used to override the trait value (e.g. “**get_children**()” overrides “**children**” and “**can_delete**()” overrides “**delete**”).
  - Sometimes you have data that is not explicitly or conveniently hierarchical, so you need to build a model that exposes the hierarchy. In this case, you can create your model classes as subclasses of **TreeNodeObject** and create corresponding **ObjectTreeNodes** for use with the model’s **TreeEditor**. An **ObjectTreeNode** simply delegates all its methods to the object it describes.

The TreeEditor Editor Factory

An example (Part 1):

```
class Employee ( HasTraits ):
    name  = Str( '<unknown>' )
    title     = Str
    phone = Regex( regex = r'\d\d\d-\d\d\d\d' )

    def default_title ( self ):
        self.title = 'Senior Engineer‘

class Department ( HasTraits ):
    name         = Str( '<unknown>' )
    employees = List( Employee )

class Company ( HasTraits ):
    name            = Str( '<unknown>' )
    departments = List( Department )
    employees   = List( Employee )

class Partner ( HasTraits ):
    name       = Str( '<unknown>' )
    company = Instance( Company )
```

The TreeEditor Editor Factory

An example (Part 2):

```
no_view = View()

tree_editor = TreeEditor(
    nodes = [
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children    = '',
                  label        = 'name',
                  view         = View( [ 'name' ] ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children    = 'departments',
                  label        = '=Departments',
                  view         = no_view,
                  add          = [ Department ] ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children    = 'employees',
                  label        = '=Employees',
                  view         = no_view,
                  add          = [ Employee ] ),
        TreeNode( node_for  = [ Department ],
                  auto_open = True,
                  children    = 'employees',
                  label        = 'name',
                  view         = View( [ 'name' ] ),
                  add          = [ Employee ] ),
        TreeNode( node_for  = [ Employee ],
                  auto_open = True,
                  label        = 'name',
                  view          = View( [ 'name', 'title', 'phone' ] ) ) ] )
```

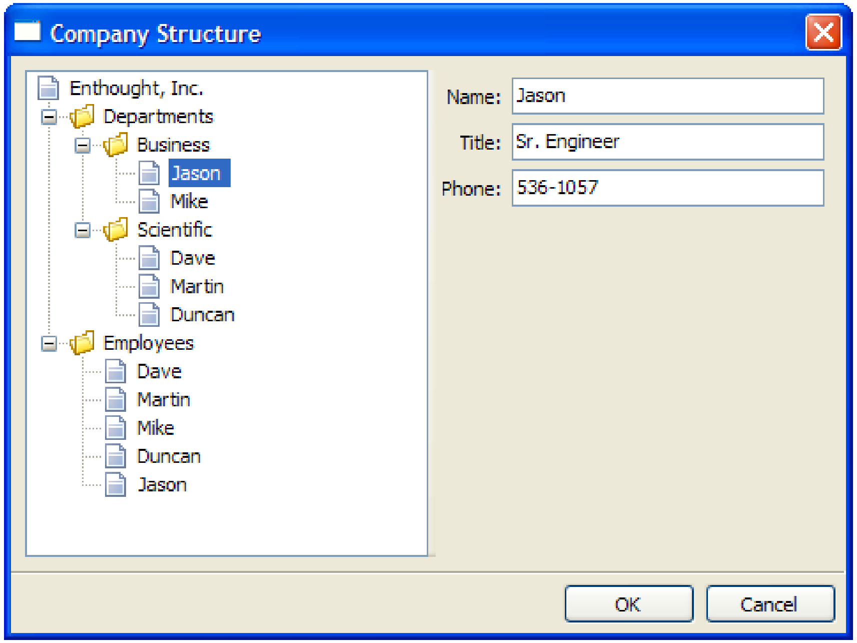
The TreeEditor Editor Factory

An example (Part 3):

```
view = View( Group( Item( name      = 'company',
                          editor    = tree_editor,
                          resizable = True ),
                    show_labels = False),
             title     = 'Company Structure',
             buttons   = [ ‘OK’, ‘Cancel’ ],
             resizable = True,
             width     = .3,
             height    = .3 )
```

The TreeEditor Editor Factory

The resulting view looks like:



Name: Jason

Title: Sr. Engineer

Phone: 536-1057

Saving and Restoring User GUI Preferences

- The Traits UI allows some user preference settings to be saved without writing any code.
- The preferences that can be saved automatically are defined by a **View** and the individual editors contained in a **View**.
- Some of the user preference settings that can be saved currently are:
  - Window/Dialog size and position
  - Splitter bar position
  - User defined table filters

Saving and Restoring User GUI Preferences

- User preferences are only saved when you request them to be saved.
- You request a preference to be saved simply by assigning a non-empty ‘id’ trait to the corresponding **View**, **Group** or **Item** object.
- In order for *any* preferences to be saved, a **View** must have a non-empty ‘id’.
- The user preference values for a **View** are saved in a global “*database*” under the view’s ‘id’, so the **View** ‘id’ should also be unique across applications and views.
- For example: View(…, id = ‘enthought.graph.vpl.graph_canvas’, …)

Saving and Restoring User GUI Preferences

- Group and Item ‘id’ values only need to be unique within the containing **View**.
- The currently supported preference items are:
  - **View** (Window/Dialog size and position)
  - **Group** (Splitter bar position when layout = ‘split’)
  - **Item** (Splitter bar position when editor = **TreeEditor**)
  - **Item** (User defined filters when editor = **TableEditor**)
- New editors can save their preferences by implementing the ‘save_prefs’ and ‘restore_prefs’ methods.