Build an Outliner Using NSOutlineView with NSTreeController

Introduction
This tutorial was intended as a starter for using NSTreeController with an outline view. However, in the process of writing this tutorial, I had a sudden realisation, and came to this conclusion: NSTreeController is completely useless for anything but the most simple of tasks. Hopefully Apple will enhance NSTreeController in the future, but for now, you will find that you will need to write much *less* code if you just use the data source methods, especially if you want to add drag and drop and state-saving. I think this fact is enough in itself to suggest that there is something seriously wrong with NSTreeController. On the other hand, I have never tried using NSTreeController with Core Data (because Core Data is not suitable for Scrivener), so maybe it is more useful in that context. With all of this in mind, I never quite finished this tutorial. I had intended to go back and add images and to edit it for typos and silly mistakes. I haven't done so, so I offer it as-is for those who do want to experiment with NSTreeController regardless. If I ever get time, I might write a tutorial for NSOutlineView using the data source methods…

Objective and Scope
Bindings have been available for some time for NSTableView by using it with an NSArrayController, but until recently the only way to populate an NSOutlineView was to rig up a datasource and write all the "glue" code yourself. With Mac OS X Tiger, though, Apple introduced NSTreeController. Ostensibly, NSTreeController is to NSOutlineView what NSArrayController is to NSTableView – an easy way to get your data into it – though in practice, NSTreeController has a number limitations not present in NSArrayController. It is also not very well documented as yet. Nonetheless, once you've worked around its shortcomings (more on those later), NSTreeController is a beautiful thing – it removes the necessity for a lot of that glue code. I had a lot of frustrations with NSTreeController at first, but now wouldn't be without it. This tutorial provides an introduction to getting up and running with NSTreeController in the hope of helping others avoid the frustrations I had.

First let me state categorically: I am no Cocoa guru, and make no pretences to be any such thing.

Anyway. In this tutorial we're going to build a simple outlining program that will let us create different notes and order them hierarchically. The interface will consist of an outline view on the left which will show us our notes organised into groups, a table view on the right that will show us the contents of any selected item in the outline view, and a text view beneath that which will show us the contents of the selected note.

We are not going to use Core Data for this tutorial. Also note that this tutorial assumes you are familiar with the basics of Cocoa bindings, know your way around Xcode, Interface Builder, and are generally comfortable with Objective-C and Cocoa.

Getting Started
In Xcode, create a new Cocoa document-based project and call it CoolOutliner.

Now let's get rid of all those unsightly references to MyDocument and rename it to something more meaningful. Because our program is called CoolOutliner, we will call our document CODocument:
1)
Rename MyDocument.h, MyDocument.m and MyDocument.nib to CODocument.h, CODocument.m and CODocument.nib respectively.
2)
In MyDocument.h and MyDocument.m, replace all references to "MyDocument" with "CODocument" (just do a Replace All in both).
3)
Finally, under Targets, double-click on where it says "CoolOutliner" to bring up the Info panel. Under

the Properties tab, under “Document Types”, change where it says “MyDocument” under “Class” to “CODocument” and close the Info panel.

At this point, it’s a good idea to build and run the application just to make sure that everything is still working correctly. If not, check that you have followed all of the preceding steps properly.

Defining the CODocument Class

Now let’s set up the interface file of the CODocument class. It’s fairly straightforward – we are going to need outlets to our outline view, table view, text view and tree controller, and we will need an array to store the contents of our outline view. We will also require a couple of actions, one for adding groups to the outline view, and one for adding notes. Our CODocument.h file should therefore look like this:

```
#import <Cocoa/Cocoa.h>

@interface CODocument : NSDocument
{
    IBOutlet NSOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;

    NSMutableArray *contents;
}
- (void)setContents:(NSArray *)newContents;
- (NSMutableArray *)contents;
- (IBAction)addGroup:(id)sender;
- (IBAction)addNote:(id)sender;
@end
```

Building the Interface

Double-click on CODocument.nib to open Interface Builder and create the interface as follows:

1)

Get rid of the “Your document contents here” text field.

2)

Drag an NSTableView to the top-right corner of the window and set its “autosizing” under Size so that it resizes horizontally and vertically (the two inner springs):

[PICTURE]

3)

Under Attributes, choose “Use Alternating Row Background”, uncheck “Horizontal Scroller” and check “Multiple Selection” and “Vertical Lines”. Make sure there are two columns, and set “Focus Ring” to “None”.

4)

Drag the first column so that it is a little wider, and name the column headers “Title” and “Description”.

5)

Drag an NSTextView underneath the NSTableView, again setting its size so that it resizes horizontally and vertically (leave room underneath it, as we will be placing some buttons at the bottom of the window):

[PICTURE]

6)

Under Attributes, check “Graphics Allowed” and “Undo Allowed” and uncheck “Editable” (our text view will only be editable when there is something to edit).
7)
Shift-click to select both the table view and the text view, and then go to Layout > Make subviews of > Split View. (Note that I find that this sometimes throws out the size a little bit, so you might need to make some minor size adjustments.) With the split view selected, set its size so that it resizes horizontally and vertically.
8)
Drag an outline view out to the left of the split view, again ensuring its size is et to resize both ways:

[PICTURE]

9)
Under Attributes, check “Multiple Selection” and uncheck “Horizonal Scroller” and “Column Headers”. Changes “Columns” to “1” and “Focus Ring” to “None”.
10)
Shift-click to select both the outline view and the split view, then go to Layout > Make subviews of > Split View. (Again, you may need to adjust the size of the newly created split view after this.) Set the size so that the new split view resizes in both directions.
11)
Drag out three buttons to the bottom-left of the window and rename them “Add Group”, “Add Note” and “Delete”. Set their size so that the area above and to the right of them can expand and shrink:

[PICTURE]

Now’s a good time to go to File > Test Interface to make sure that everything is looking good (well, okay, we’re going to win no awards for interface design here) and that everything sizes properly. Satisfied? Okay, now it’s time to wire everything together.

Before we can do that, we need to let Interface Builder know about the outlets and actions we created in COController:

1)
Make sure Xcode is visible in the background of Interface Builder, and drag COController.h from Xcode and drop it into the Instances tab of the nib’s document panel in Interface Builder.
2)
That will have automatically switched the panel to the “Classes” tab. Switch back to the “Instances” tab, select “File’s Owner”, and then, under “Custom Classes” in the inspector, set File’s Owner’s class to “CODocument”.

Now let’s set our outlets:
1)
Ctrl-drag from File’s Owner (which represents CODocument, of course) to the outline view, and set the outlineView outlet.
2)
Do the same for the text view and table view, setting them to the textView and tableView outlets respectively.

There’s one missing though – the treeController outlet. So let’s sort that out, even though we won’t be setting up our tree controller just yet:

1)
Drag out an NSTreeController from the Controllers palette and drop it into the Instances tab of the

nib's document panel. Rename it "OutlineController":

[PICTURE]

2)
Ctrl-drag from File's Owner to the newly-created tree controller, and set the treeController outlet.

Now let's connect our actions:

1)
Ctrl-drag from the "Add Group" button to "File's Owner" and connect its action to addGroup:.
2)
Ctrl-drag from the "Add Note" button to "File's Owner" and connect its action to addNote:.
3)
Ctrl-drag from the "Delete" button to "OutlineController" and connect its action to remove:.

That's our interface built. It won't do much yet, though – we need to think about the data with which we are going to populate it next. Save the .nib file and return to Xcode for the next part.

The Model Object
Next, we need to design a model object that will work nicely with the outline view. All we have to design is the model that can be used for each "node" (or branch) of the outline view. This node model will have to main requirements:

1)
It must know whether it is a "group" node (one that has a disclosure triangle and can contain sub-nodes) or a "leaf" node (which will be used for our documents – leaf nodes do not have disclosure triangles and cannot hold sub-nodes).
2)
It must be able to hold an array of all sub-nodes if it is a "group" node.

The latter is fairly straightforward – our model can just contain an NSMutableArray named "children".

As for the former – seeing as NSTreeController differentiates between leaf and group nodes by looking for a leaf key path, it makes sense for our model to contain an isLeaf BOOL variable. With this in mind, our node model class header would look like this at its most basic:

```
@interface Node : NSObject
{

NSMutableArray *children;

BOOL isLeaf;
}

- (void)setChildren:(NSArray *)newChildren;
- (NSMutableArray *)children;

- (void)setLeaf:(BOOL)flag;
- (BOOL)isLeaf;
```

In fact, I have already written a multi-purpose base node class called KBBaseNode that provides this functionality along with support for archiving, copying and several convenience methods that make supporting drag and drop much easier. Although KBBaseNode works as-is by providing a "properties" mutable dictionary variable that could be used to contain any extra instance variables you may need, it

is designed to be subclassed. We will use KBBaseNode as the basis for our model class in this tutorial.

The files for KBBaseNode can be downloaded here.

Subclassing KBBaseNode
Before we can subclass KBBaseNode, make sure you add KBBaseNode.h and KBBaseNode.m to your project in the usual way.

Our nodes need to hold the following information:

- The title of the note.
- A short description of the note.
- The text of the note.

We will use NSStrings for the title and description, and we will use an NSTextStorage for the text (you will see why later).

As mentioned earlier, we *could* quite easily store all of this information in the "properties" dictionary of KBBaseNode, in which case we wouldn't need to subclass at all. However, if we needed a lot of extra variables, our dictionary could soon grow unwieldy, so we will subclass.

Create a new Objective-C class in Xcode and name it "CONode". Change the header so that it looks like this:

```
#import <Cocoa/Cocoa.h>
#import "KBBaseNode.h"

@interface CONode : KBBaseNode
{
    NSString *title;
    NSString *description;
    NSTextStorage *text;
}

- (void)setTitle:(NSString *)newTitle;
- (NSString *)title;
- (void)setDescription:(NSString *)newDescription;
- (NSString *)description;
- (void)setText:(id)newText;
- (NSTextStorage *)text;

@end
```

Now we need to write the actual code. Because the super class does most of the work for us (including ensuring that archiving and copying works even with subclasses), there's not much to do. We have to override the initialisation and deallocation methods, write our accessors, and override a method that will tell the superclass that it needs to archive and unarchive our new instance variables.

KBBaseNode has two –init methods: -init initialises all of the instance variables and sets isLeaf to NO (thus creating a group node), and –initLeaf calls –init but then sets isLeaf to YES (thus creating a leaf node). –initLeaf also sets the children array so that it just contains a reference to self. We will see why it does this rather than leaving the children array empty a little later. All we need to know for now is that, because -initLeaf calls -init to set its instance variables, the only initialiser we need to override

is –init.

In CONode.h, then, our –init and –dealloc methods will look like this:

```
- (id)init
{
    if (self = [super init])
    {
        title = [[NSString alloc] initWithString:@"Untitled"];
        description = [[NSString alloc] initWithString:@"- No description -"];
        text = [[NSTextStorage alloc] init];
    }
    return self;
}

- (void)dealloc
{
    [title release];
    [description release];
    [text release];
    [super dealloc];
}
```

Next we will write our accessors, which will look like this:

```
- (void)setTitle:(NSString *)newTitle
{
    [newTitle retain];
    [title release];
    title = newTitle;
}

- (NSString *)title
{
    return title;
}

- (void)setDescription:(NSString *)newDescription
{
    [newDescription retain];
    [description release];
    description = newDescription;
}

- (NSString *)description
{
    return description;
}

- (void)setText:(id)newText
{
    if ([newText isKindOfClass:[NSAttributedString class]])
        [text replaceCharactersInRange:NSMakeRange(0,[text length])
withAttributedString:newText];
    else
        [text replaceCharactersInRange:NSMakeRange(0,[text length]) withString:newText];
}

- (NSTextStorage *)text
{
```

```
    return text;
}
```

Note that setText: can accept an NSTextStorage, an NSAttributedString, an NSMutableAttributedString, or a simple NSString – any will work. (This is a nifty tip I picked up from an article over at www.projectomega.org.)

Finally, we need to override KBBaseNode's –mutableKeys method, like so:

```
- (NSArray *)mutableKeys
{
    return [[super mutableKeys] arrayByAddingObjectsFromArray:[NSArray arrayWithObjects:
        @"title",
        @"description",
        @"text",
        nil]];
}
```

Because KBBaseModel's –initWithCoder:, -encodeWithCoder: and –copyWithZone: methods use the -mutableKeys method to determine which instance variables need archiving and copying, by overriding just this method we do not have to bother overriding any of the others just mentioned. Those methods will now know that about our three new instance variables that respond to the key-value coding keys "title", "description" and "text". This is really time-saving, and I take zero credit for it – it's a great tip I found over at Todd Ransoms's site:

http://returnself.com/blog/archives/category/apple/cocoa/

(Note that if we had added an instance variable that was not an object – such as an int or a BOOL – we would also have had to override –setNilValueForKey:.)

And that's our model code done. There's one more thing we need to do before we bind everything together, though…

Writing the Document Class Code

Before moving on, we need to write the code for CODocument – we have so far only written the header. First, let's write the init/dealloc methods, which are straightforward. Amdend the –init method in CODocument.m and add the –dealloc method as follows:

```
- (id)init
{
    if (self = [super init])
    {
        contents = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)dealloc
{
    [contents release];
    [super dealloc];
}
```

Next, add the accessor methods:

```
- (void)setContents:(NSArray *)newContents
{
```

```
    if (contents != newContents)
    {
        [contents autorelease];
        contents = [[NSMutableArray alloc] initWithArray:newContents];
    }
}

- (NSMutableArray *)contents
{
    return contents;
}
```

Remember that we defined two action methods – addGroup: and addNote: - in our CODocument class a little earlier? Well, now that we've created our model, it's time to write the code for them.

First, why are we bothering to create these methods at all? NSTreeController already has methods for add: and addChild:, after all. The trouble is that NSTreeController has no method for addLeaf:. So if you rely on the NSTreeController methods, you will only ever be able to create group nodes, not leaf nodes (because our node model's –init method automatically initialises with isLeaf set to NO). Moreover, you may want to change certain attributes of newly created nodes before they are added to the outline view, such as adding a unique project ID (something we will do later when adding drag and drop support) or appending a number to the title (eg. "untitled1", "untitled2", etc). The NSTreeController methods can't help us with all of this, so this is why we create our own actions in our document. Right, enough blathering – time to write the code for these two actions. Before we can do so, we need to tell CODocument about our CONode class. Add the following line to the top of CODocument.m, right below #import "CODocument.h":

```
#import "CONode.h"
```

Finally, add the actions:

```
- (IBAction)addGroup:(id)sender
{
    // NSTreeController inserts objects using NSIndexPath, so we need to calculate this
first
    NSIndexPath *indexPath = nil;

    // If there is no selection, we will add a new group to the end of our contents array
    if (![treeController selectionIndexPath])
        indexPath = [NSIndexPath indexPathWithIndex:[contents count]];

    else
    {
        // We can't add nodes to leaf nodes
        if ([[[treeController selectedObjects] objectAtIndex:0] isLeaf])
        {
            NSBeep();
            return;
        }

        // Get the index path of the currently selected node, and then add the number its
children to the path -
        // this will give us an index path which will allow us to add a node to the end of
the currently selected
        // node's children array
        indexPath = [treeController selectionIndexPath];
        indexPath = [indexPath indexPathByAddingIndex:[[[treeController selectedObjects]
objectAtIndex:0] children] count]];
```

```
    }

    // Create and add a new group node
    CONode *node = [[CONode alloc] init];
    // This is where you would add any code to customise the new node
    [treeController insertObject:node atArrangedObjectIndexPath:indexPath];
    [node release];
}

- (IBAction)addNote:(id)sender
{
    // We do not allow nodes to be added to the root - they can only be added to groups.
(If you want to allow
    // notes to be added to the root, just get ride of these five lines.)
    if (![treeController selectionIndexPath])
    {
        NSBeep();
        return;
    }

    // We can't add nodes to leaf nodes
    if ([[[treeController selectedObjects] objectAtIndex:0] isLeaf])
    {
        NSBeep();
        return;
    }

    NSIndexPath *indexPath = [treeController selectionIndexPath];
    indexPath = [indexPath indexPathByAddingIndex:[[[[treeController selectedObjects]
objectAtIndex:0] children] count]];

    CONode *node = [[CONode alloc] initLeaf];
    // This is where you would add any code to customise the new node
    [treeController insertObject:node atArrangedObjectIndexPath:indexPath];
    [node release];
}
```

The most important thing to note about these methods is how they use NSIndexPath to insert the nodes into our tree controller. From the NSIndexPathDocs:

"NSIndexPath specifies the path to a specific node in a tree of nested array collections. Each index in an index path represents the index into an array of children from one node in the tree to another, deeper, node."

Although these methods both add new nodes to the end of currently selected group node (and thus just beep and do nothing if a leaf node is selected), a quick read of the NSIndexPath and NSTreeController docs will reveal how easy it would be to change this behaviour if you want, or to add insert methods. (For insert methods, for instance, you would just insert the object at the currently selected index path.)

Right, time to bind our CONode model to our interface – which will bring us to the end of the first part of this tutorial.

Binding to the Interface

Back in Interface Builder, select OutlineController in the nib document panel and make sure "Attributes" is selected in the inspector. To tell our NSTreeController to use our CONode class, do the following:

1)

Set “Object Class Name” to “CONode”.
2)
Set “Children key path” to “children”.
3)
Set “Leaf key path” to “isLeaf”.

[PICTURE]

What Have We Just Done?
We have just told the NSTreeController that it will be dealing with objects of class CONode. We have also told it that it can find the children for these nodes by looking in an array entitled “children”. And finally, we told it that it can tell which objects should show a disclosure triangle and which ones should not by looking for the “isLeaf” variable. (Note that if an item has isLeaf set to YES, the tree controller will not show it as having any children even if it has objects inside its children array – which is why we are safe to put leaf objects inside their own children array.)

Now we need to tell the NSTreeController where all those objects should be stored. This is what we created the contents array for in CODocument, so we need to tell it all about that. With OutlineController still selected, choose “Bindings” in the inspector:

1)
Under “contentArray”, choose “File’s Owner (CODocument)” from the dropdown menu.
2)
Next to “Model Key Path”, type “contents” and hit return. This should automatically check the “bind” checkbox.

[PICTURE]

That’s all we need to to to get our NSTreeController up and running. Next we need to tell our outline view and table view to use this information:

1)
Double-click on the outline view until you can select its column (there should be only one column). Make sure “Bindings” is still selected in the inspector.
2)
Click on the disclosure triangle next to “contents”. “Bind to” should already be set to “OutlineController (NSTreeController)” and “Controller Key” should already be set to “arrangedObjects” (if not, make sure they are).
3)
Next to “Model Key Path”, type “title” and hit enter so that “Bind” gets checked.

[PICTURE]

We have just told the outline view column to display the objects in the tree controller using their titles.

Our table view works slightly differently – our table view shows items that are selected in the outline view. To set this up, we’re also going to need an NSArrayController. Let’s set this up now:

1)
Drag an NSArrayController out from the Controllers palette and drop it into the nib document panel. Rename it TableController.

[PICTURE]

2)
With TableController still selected, select “Attributes” in the inspector.
3)
Set the “Object Class Name” to “CONode”, and uncheck “Avoids Empty Selection”.
4)
Select “Bindings” in the inspector.
5)
Under “value”, “Bind to” should already be set to “OutlineController (NSTreeController)” and “Controller Key” should already be set to “selection” (if not, do that now).
6)
Next to “Model Key Path”, type “children” and hit enter so that “Bind” gets checked.

What Have We Just Done?
We have just created an array controller which will dynamically change depending on what is selected in the outline view. We told it to use the “children” array of whatever is selected in the outline view, and that this array will contain objects of type CONode. We also told it not to avoid empty selection – this means that newly created nodes displayed in the table view won’t get selected by default (this will ensure that the selection never gets confused about determining which text to show later). This means that we can now rig up our table to show only the children of selected items:

1)
Double-click on the table view until you can select the “Title” column. Again, make sure “Bindings” is still selected in the inspector.
2)
Under “value”, set “Bind to” to “TableController (NSArrayController)” if it is not already.
3)
Set “Controller Key” to “arrangedObjects”.
4)
Set “Model Key Path” to “title”, hitting return to ensure that “Bind” gets checked.
5)
Do exactly the same for the “Description” column, but setting “Model Key Path” to “description”.

That has told our table columns to display information about the nodes that can be found in our array controller – which just contains the children of the node currently selected in the outline view.

Build and Test
Okay, at last we get to test what we have done so far. Build and Run the application. Try adding groups and notes, editing them, deleting them, whatever. You will see that everything works beautifully, without any fuss. Note that if a group node is selected, the table view displays its contents. If a leaf node (a note) is selected, though, the table view just shows the information for that note. This is exactly how Xcode works, too. Now it should be clear why leaf nodes store a reference to themselves inside their children array.

[PICTURE]

Something’s still missing though – the selection in the outline view or table view has no effect on the text displayed in the text view, and we can’t yet edit the text view at all. Let’s fix this.

Setting Up the Text View
We could have used the text view’s bindings to bind it to the selected object in the outline view or the table view. The reason we’re not going to do this is because the text displayed in the text view can be determined by the selection in *either* the outline view or table view – depending on which one was

clicked on last. Moreover, if a group node is clicked, this shouldn't affect the text that is displayed because we don't want to associate any text with group nodes (even though all nodes have a text variable, we are only going to use it for notes, not groups).

Before we do anything else, let's do something we should have done back at the beginning – let's the "Format" menu to our menu so that we can change fonts and display the ruler in our text view if we so desire:

1)
Double-click "MainMenu.nib" to open it in Interface Builder.
2)
Make sure the menu is visible – if you can't see it, double-click on "MainMenu" in the nib document panel.
3)
From the Menus palette, drag the "Format" menu between "Edit" and "Window".
4)
Save and close "MainMenu.nib".

To get our text view working, we are going to use the delegate methods of our table view and outline view:

-outlineViewSelectionDidChange:
-tableViewSelectionDidChange:

These methods will tell us whenever the selection changes. We can then get the currently selected object from the tree controller (if the outline view caused the change) or the array controller (if the table view caused the change) and display the text accordingly. Because we will need to access the array controller in our document for this, we need to add an outlet to the array controller in CODocument.h, as follows:

```
@interface CODocument : NSDocument
{
    IBOutlet NSOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
}
```

Open up "CODocument.nib" in Interface Builder if it isn't already open, drag CODocument.h from Xcode into the nib document panel in Interface Builder, and then ctrl-drag from File's Owner to TableController and connect the new arrayController outlet.

Now we need to set CODocument as the delegate of the outline and table views:

1)
Ctrl-drag from the outline view to File's Owner and make sure "Outlets" is selected in the inspector. Connect the "delegate" outlet.
2)
Do the same for the table view.

Save the nib file and then return to Xcode. To implement the delegate methods in which we are interested, add the following to CODocument.m:

```
- (void)outlineViewSelectionDidChange:(NSNotification *)notification
{
    // Make sure we are responding to the correct outline view
    if ([notification object] != outlineView)
        return;

    // If the selection changed to nothing, do nothing
    if ([[treeController selectedObjects] count] == 0)
        return;

    CONode *selectedNode = [[treeController selectedObjects] objectAtIndex:0];

    // Don't display the text of group nodes
    if (![selectedNode isLeaf])
        return;

    [textView setSelectedRange:NSMakeRange(0,0)];

    // Replace the text in the text view if it has changed
    if ([textView textStorage] != [selectedNode text])
        [[textView layoutManager] replaceTextStorage:[selectedNode text]];

    // Make sure the text view is in an editable state
    if (![textView isEditable])
        [textView setEditable:YES];
}

- (void)tableViewSelectionDidChange:(NSNotification *)notification
{
    // Make sure we are responding to the correct table view
    if ([notification object] != tableView)
        return;

    // If the selection changed to nothing, do nothing
    if ([[arrayController selectedObjects] count] == 0)
        return;

    CONode *selectedNode = [[arrayController selectedObjects] objectAtIndex:0];

    // Don't display the text of group nodes
    if (![selectedNode isLeaf])
        return;

    [textView setSelectedRange:NSMakeRange(0,0)];

    // Replace the text in the text view if it has changed
    if ([textView textStorage] != [selectedNode text])
        [[textView layoutManager] replaceTextStorage:[selectedNode text]];

    // Make sure the text view is in an editable state
    if (![textView isEditable])
        [textView setEditable:YES];
}
```

What Have We Done?

This is fairly straightforward. Whenever the selection changes in the outline or table view, we first make sure that it hasn't changed to an empty selection. If it has, we do nothing (we just leave whatever text was previously selected, if any was, in the text view). Next, we check whether the selected object (we take the selected object to be the first one in the selection if there is multiple selection) is a group

node – if it is, we do nothing, because group nodes are just folders that don't contain any text. (You may want to change this behaviour – just remove the isLeaf check if you want folders to be associated with text too.) If we've got this far, we know we can change the text. We reset the selection to avoid out of bounds errors, swap the text storage, and ensure that our text view is editable.

Now it's time to build and run the application again. Have a play around – it's now fully functional. Try writing some text, changing the selection, flicking between notes. Cool.

Now that we have our basic functionality sorted out, it's time to turn it into a more rounded application. To do this, we're going to add save/load capabilities and drag and drop support.

Adding Save/Load Capabilities

Prior to Tiger, saving and loading in document-based applications was usually done by overriding -dataRepresentationOfType: and –loadDataRepresentation:ofType:. If we're only supporting Tiger and above, though (and seeing as we are using the Tiger-only NSTreeController, we are), we should use –dataOfType:error: and –readFromData:ofType:error:. Those are the methods we shall override here:

1)
Delete -dataRepresentationOfType: and –loadDataRepresentation:ofType: from CODocument.m.
2)
Implement –dataOfType:error: and –readFromData:ofType:error: as follows:

```
- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError
{
    NSDictionary *d = [NSDictionary dictionaryWithObjectsAndKeys:
        contents,@"contents",
        nil];
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:d];
    return data;
}

- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError **)outError
{
    NSDictionary *d = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    [self setContents:[d objectForKey:@"contents"]];
    return YES;
}
```

Note that we use an NSDictionary to save our file rather than archive the contents array directly. This is only because I know that we are going to need to save some other variables in a little while, which we can easily add to the dictionary.

Also note that we have only done the bare minimum here – if this was a shipping application, we should really check for errors and fill outError as appropriate, and check for typeName if we want to allow different types.

Next, we need to set the extension of any files we save:

1)
Double click on CoolOutliner under Targets to open the Info panel.
2)
Under "Document Types" in the "Properties" tab, set "Name" to "Cool Outliner Document".
3)
Set "Extension" to "cold" (for "cool out-liner document" – or choose whatever you like).
4)

Close the Info panel.

Build and run the application again. You should now be able to save and load your work to and from disk.

Saving and Restoring Expanded State

You may notice that one thing isn't getting saved, though: the expanded state of the outline items. Whenever you load a saved file, all of the nodes are collapsed, no matter what state they were in when you saved your work. We want to save the state so that whenever we open the file, the outline view items are in exactly the same expanded or collapsed state that they were in when we saved it.

NSOutlineView already provides datasource methods to support this:

-outlineView:persistentObjectForItem:
-outlineView:itemForPersistentObject:

Great! We'll just implement these. But wait a minute… Remember back at the beginning, when I said that NSTreeController has some annoying shortcomings? Well, this is one of them. In order to use these methods, you expect to be passed in an item. In –outlineView:persistentObjectForItem:, you are expected to decide on something that will identify that item at a later date – perhaps a unique identifier, for instance. Then in –outlineView:itemForPersistentObject: you are passed back that unique identifier and expected to be able to find the item it refers to. This is fairly straightforward when you're not using bindings and NSTreeController. Unfortunately, though, when you use NSTreeController, the item you get passed isn't – as you would expect in our case – an object of class CONode, which is after all what is stored in the outline view. Oh no, that would be far too simple. Instead, you get passed a proxy object of private class _NSArrayControllerTreeNode. That's right – a completely undocumented and opaque class. So what can we do?

Well, fortunately, Scott Stevenson of CocoaDevCentral has pointed out that _NSArrayControllerTreeNode does respond to an undocumented method, -observedObject. This will return the object you would have expected to have been passed in the first place – in our case, a CONode object. It will throw a compiler warning, because it's an undocumented, private method, but we can suppress that by declaring the method in a category on NSObject. Using –observedObject, then, we could get the object passed in by –outlineView:persistentObjectForItem: and return a unique, persistent object for it. Unfortunately, though, this doesn't help us when it comes –outlineView:itemForPersistentObject:. This method expects us to return an item of the same type it passed us in the former method – in other words, it expects an _NSArrayControllerTreeNode object. And there is no way we can find that object, because we have no access to this class.

This is all just a long-winded way of saying that there is no way – or at least no way I have found – to use NSOutlineView's traditional expanded state-saving methods when working with NSTreeController.

So what are we to do? Well, we have no choice but to write our own code to save the state of the outline view. Happily enough, this isn't too complicated. To make our code as reusable as possible, we are going to create simple subclass of NSOutlineView that will make it very easy for us to save its state. We'll call it ESOutlineView, for "Expanded State Outline View", to remind ourselves that all it does extra is help save the expanded state:

1)
Create a new Objective-C class in Xcode and name it ESOutlineView.
2)
Edit the ESOutlineView.h file so that it looks like this:

```
#import <Cocoa/Cocoa.h>


@interface ESOutlineView : NSOutlineView
{
}
- (id)observedObjectForItem:(id)item;
- (NSArray *)expandedState;
- (BOOL)restoreExpandedStateWithArray:(NSArray *)stateArray;
@end

@interface NSObject (ESOutlineViewDataSource)
- (id)outlineView:(NSOutlineView *)outlineView
uniqueValueForObservedObject:(id)observedObject;
@end
```

All we've done is declared three methods in our new NSOutlineView subclass. –obsevedObjectForItem: will provide us with a nice way to use the undocumented –observedObject method we discussed earlier. The other two methods are self-explanatory – they will deal with saving and restoring the expanded state of items in the view. We have also declared a new datasource method, -outlineView:uniqueValueForObservedObject:. This has pretty much the same purpose as –outlineView:persistentObjectForItem: - our outline view will use the returned value to save and restore the expanded state of the object.

Now let's write the code for the implementation. Edit ESOutlineView.m so that it looks like this:

```
#import "ESOutlineView.h"


// Avoid compiler errors by declaring the required _NSArrayControllerTreeNode private
method here
@interface NSObject (PrivateTreeNodeMethods)
- (id)observedObject;
@end


@implementation ESOutlineView

- (id)observedObjectForItem:(id)item
{
    if ([item respondsToSelector:@selector(observedObject)])
        return [item observedObject];
    return item;
}

- (NSArray *)expandedState
{
    // This depends on the delegate implementing uniqueValueForObservedObject
    if (![[self dataSource]
respondsToSelector:@selector(outlineView:uniqueValueForObservedObject:)])
        return nil;

    NSMutableArray *state = [NSMutableArray array];
    int i;
    for (i=0; i<[self numberOfRows]; i++)
    {
        if ([self isItemExpanded:[self itemAtRow:i]])
            [state addObject:[[self dataSource] outlineView:self
                                uniqueValueForObservedObject:[self
```

```
observedObjectForItem:[self itemAtRow:i]]]];
    }
    return state;
}

- (BOOL)restoreExpandedStateWithArray:(NSArray *)state
{
    if (![[self dataSource]
respondsToSelector:@selector(outlineView:uniqueValueForObservedObject:)])
        return NO;

    int i;
    int numberOfRows = [self numberOfRows];
    for (i=0; i<[state count]; i++)
    {
        int d;
        for (d=0; d<numberOfRows; d++)
        {
            if ([[[self dataSource] outlineView:self
                    uniqueValueForObservedObject:[self observedObjectForItem:[self
itemAtRow:d]]]
                isEqual:[state objectAtIndex:i]])
            {
                [self expandItem:[self itemAtRow:d]];
                numberOfRows = [self numberOfRows];
            }
        }
    }
    return YES;
}

@end
```

What have we done?
There is nothing too complicated here.

The first thing we do is declare –observedObject in a category of NSObject. This tells the compiler about the method so that when we use it on those private _NSArrayControllerTreeNode objects it doesn't spew a warning.

Next we implement –observedObjectForItem:. If the item passed in responds to –observedObject, we return that – otherwise we just return the original item. That way, the method is compatible with items that are not of class _NSArrayControllerTreeNode, too.

Then we implement –expandedState, which we can use to save the state of our outline view. First it checks to ensure that the outline view's datasource implements –outlineView:uniqueValueForObservedObject: - if it doesn't, it returns nil as it can't do anything. If everything is present and correct, though, it runs through all the rows in the outline view. Whenever it comes to an item that is expanded, it asks the datasource for a unique identifier for this item and saves that identifier in an array. It then returns the completed array. That way, we can save the state array in with our document data.

Finally, we implement –restoreExpandedStateWithArray:. Again, we first check to make sure that the datasource implements the necessary method and do nothing if not. Then we cycle through all of the objects inside the array passed in (which will be the array we saved using –expandedState) and then look through all of the items in the outline view to see if any match what is in the array. If they do, we expand them and update the number of rows (which obviously changes whenever an item is

expanded).

Right, time to put all that into action. The first thing we need to do is change the NSOutlineView we have been using up until now into an ESOutlineView. First change it in CODocument.h, ensuring that you remember to import ESOutlineView.h too:

```
#import <Cocoa/Cocoa.h>
#import "ESOutlineView.h"

@interface CODocument : NSDocument
{
    IBOutlet ESOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
    NSMutableArray *selectedNodes; // Nodes that are shown in the table view
}
```

Next, we need to change it in Interface Builder:

1)
Open up CODocument.nib in Interface Builder again.
2)
Drag ESOutineView.h from Xcode and drop it into the nib document panel.
3)
Select the outline view.
4)
From “Custom Class” in the inspect, change the class of the outline view from NSOutlineView to ESOutlineView.
5)
Ctrl-drag from the outline view to File’s Owner and set File’s Owner to be the datasource of the outline view.
6)
Save and return to Xcode.

There are certain methods that a datasource *must* implement. So, because we’ve rigged up our CODocument object to be the datasource of the outline view, we need to implement these compulsory datasource methods, otherwise we’ll get errors. Strictly speaking, we don’t need any of the compulsory datasource methods because our NSTreeController does all the stuff that these datasource methods normally do. But because they *must* be implemented, we’ll create dummy methods that do nothing. At the bottom of COController.m, implement them thus:

```
- (int)outlineView:(NSOutlineView *)ov numberOfChildrenOfItem:(id)item
{
    return 0;
}

- (BOOL)outlineView:(NSOutlineView *)ov isItemExpandable:(id)item
{
    return NO;
}

- (id)outlineView:(NSOutlineView *)ov child:(int)index ofItem:(id)item
{
```

```
    return nil;
}

- (id)outlineView:(NSOutlineView *)ov objectValueForTableColumn:(NSTableColumn
*)tableColumn byItem:(id)item
{
    return nil;
}
```

Now all we have to do is implement –outlineView:uniqueValueForObservedObject: and use our new outline view methods to save and restore the outline view's state. There are a number of ways you might assign a unique value to an item, but an easy way is just to assign each note a unique project number whenever it is created. This is what we will do. For this, we'll need to create a new instance variable in CODocument entitled uniqueID. We'll also create a –uniqueID method that will ensure we get a different number everytime we access it. Let's declare these in CODocument.h first:

```
@interface CODocument : NSDocument
{
    IBOutlet ESOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
    NSMutableArray *selectedNodes; // Nodes that are shown in the table view

    int uniqueID;
}
- (void)setContents:(NSArray *)newContents;
- (NSMutableArray *)contents;
- (int)uniqueID;
- (IBAction)addGroup:(id)sender;
- (IBAction)addNote:(id)sender;
@end
```

Now let's initialise the new variable in our –init method in CODocument.m:

```
- (id)init
{
    if (self = [super init])
    {
        contents = [[NSMutableArray alloc] init];
        selectedNodes = [[NSMutableArray alloc] init];
        uniqueID = 0;
    }
    return self;
}
```

And let's write the code to the method we will use to get a unique ID:

```
- (int)uniqueID
{
    return uniqueID++;
}
```

Every time we request an ID, we add one so that every time we ask it will be different.

Now we need to make sure that whenever we create a new node, it is given a unique ID. We're not

going to bother creating a new instance variable in CONode for this, although we could. Instead, we will use CONode's properties dictionary that it inherits from KBBaseNode to store the ID. We need to make a minor alteration to our –addGroup: and –addNote: action methods to do this. Edit the last three lines in –addGroup: as follows:

```
CONode *node = [[CONode alloc] init];
[node setProperties:[NSDictionary dictionaryWithObject:[NSNumber numberWithInt:[self
uniqueID]]
                                                forKey:@"ID"]];
[treeController insertObject:node atArrangedObjectIndexPath:indexPath];
[node release];
```

Do the same for –addNote:

```
CONode *node = [[CONode alloc] initLeaf];
    [node setProperties:[NSDictionary dictionaryWithObject:[NSNumber numberWithInt:[self
uniqueID]]
                                                    forKey:@"ID"]];
    [treeController insertObject:node atArrangedObjectIndexPath:indexPath];

[node release];
```

Now we are ready to implement our state-saving datasource method. Add the following method to the end of CODocument.m:

```
- (id)outlineView:(NSOutlineView *)ov uniqueValueForObservedObject:(id)object
{
    return [[object properties] objectForKey:@"ID"];
}
```

That provides the outline view with the information it needs to save the state. Now all we have to do is modify our saving and loading methods. First let's change our save method:

```
- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError
{
    NSDictionary *d = [NSDictionary dictionaryWithObjectsAndKeys:
        contents,@"contents",
        [outlineView expandedState],@"outlineState",
        [NSNumber numberWithInt:uniqueID],@"uniqueID",
        nil];
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:d];
    return data;
}
```

Note that we not also save the current uniqueID – that way, when we reload our project we can carry on creating unique IDs from where we left off.

Loading is a little more tricky. Becasuse –readFromData:ofType:error: is called before the interface gets loaded, we can't just restore the state of the outline view in that method. Instead, we will need to save the state information and wait until the interface has loaded in –windowControllerDidLoadNib: to restore the state. We therefore need an extra instance variable for this, which we need to define now in CODocument.h:

```
@interface CODocument : NSDocument
{
    IBOutlet ESOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
```

```
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
    NSMutableArray *selectedNodes; // Nodes that are shown in the table view

    int uniqueID;
    NSArray *outlineExpandedState;
}
```

Now we can edit our load method in CODocument.m:

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError **)outError
{
    NSDictionary *d = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    [self setContents:[d objectForKey:@"contents"]];
    outlineExpandedState = [[d objectForKey:@"outlineState"] retain];
    uniqueID = [[d objectForKey:@"uniqueID"] intValue];
    return YES;
}
```

Finally, we just need to edit –windowControllerDidLoadNib: to use the information we just saved to restore the outline view's state:

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    [super windowControllerDidLoadNib:aController];
    // Add any code here that needs to be executed once the windowController has loaded the
document's window.

    if (outlineExpandedState)
    {
        [outlineView restoreExpandedStateWithArray:outlineExpandedState];
        [outlineExpandedState release];// No longer need the info
    }
}
```

At this point, you're probably thinking that was a whole lot of work just to save the state of the outline view. I agree. Please file a bug report/enhancement request with Apple begging them to get their state-saving methods to work with NSTreeController.

At any rate, now you can rebuild and run the program, and you should find that whenever you save and reload a project, the outline retains its state. (Note that because we changed the save and load information, you'll receive errors if you try save any old projects that you load. Obviously, if this was a shipping application, you would make sure that when you modified the save and load methods, they would convert old file formats. For now, you'll have to create a completely new project and save it to get results.)

Clearly there is still room for improvement here if this was a shipping application: there are other things we would want to save (like the currently selected note and the window size and position), and we may want to change it so that the outline state gets saved more often, not just when the document is saved. But for the purposes of this tutorial, we have now managed to save the state of our outline view even with NSTreeController.

The main thing remaining now is to add drag and drop support. But before that, I'm going to head off on a tangent and fix something that has been annoying me.

Refining the Table View

Now that I have played with CoolOutliner more, something bugs me. I want the table view to work just like the one in Xcode, but right now it doesn't. I want it to show only notes, not groups. At the moment it will show groups that are children of another group. I want it to show all the notes selected if there is a multiple selection – at the moment, though, it shows nothing if we select more than one thing. If a group and a note is selected, I want the table view to show the note and all of the children of the group. Hell, I want the table view to show all of the children of the children of any groups that are selected. That's how Xcode works, and that's how I want CoolOutliner to work. Fortunately, it's not too difficult – we are just going to have to change the bindings of the table view's array controller. First, we need to create a new array in CODocument which we will use to hold all of the selected items – because we want custom behaviour, we will have to manage this array ourselves. We will also need accessors for it. Add the new array and its accessors to CODocument.h:

```
@interface CODocument : NSDocument
{
    IBOutlet NSOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
    NSMutableArray *selectedNodes; // Nodes that are shown in the table view
}
- (void)setContents:(NSArray *)newContents;
- (NSMutableArray *)contents;
- (void)setSelectedNodes:(NSArray *)newSelection;
- (NSMutableArray *)selectedNodes;
```

Now we need to alter the –init and –dealloc methods in CODocument.m to manage our new array:

```
- (id)init
{
    if (self = [super init])
    {
        contents = [[NSMutableArray alloc] init];
        selectedNodes = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)dealloc
{
    [contents release];
    [selectedNodes release];
    [super dealloc];
}
```

And we need to add the accessors somewhere in CODocument.m, too:

```
- (void)setSelectedNodes:(NSArray *)newSelection
{
    if (selectedNodes != newSelection)
    {
        [selectedNodes autorelease];
        selectedNodes = [[NSMutableArray alloc] initWithArray:newSelection];
    }
```

```
}

- (NSMutableArray *)selectedNodes
{
    return selectedNodes;
}
```

Next, we need to change the bindings of our array controller to use this new array:

1)
Open up CODocument.nib in Interface Builder and select TableController.
2)
Select “Bindings” in the inspector.
3)
Change the contentArray bindings so that “Bind to” is set to “File’s Owner (CODocument)”, “Controller Key” is blank, and “Model Key Path” is set to “selectedNodes”.

That’s all you need to do, so you can now save and close interface builder. The array controller and the table view that is bound to it are now ready to use our new selectedNodes array. Now all we need to do is ensure that this array contains all of the selected nodes and gets updated whenever the selection in the outline view changes. The best place to do this, of course, is back in –outlineViewSelectionDidChange:. Fortunately for us, KBBaseNode – and therefore our CONode subclass – has a method called -allChildLeafs which returns an array containing all of the children and grandchildren and great-grandchildren (and so on) of the node that have isLeaf set to YES. We will use this method to calculate what should go into our selectedNodes array. Add the following lines in bold to the –outlineViewSelectionDidChange: method:

```
    // Make sure we are responding to the correct outline view
    if ([notification object] != outlineView)
        return;

    // Deal with multiple selection
    NSMutableArray *newSelection = [NSMutableArray array];
    if ([[treeController selectedObjects] count] > 1)
    {
        NSEnumerator *enumerator = [[treeController selectedObjects] objectEnumerator];
        CONode *node;

        while (node = [enumerator nextObject])
        {
            if ([node isLeaf])
            {
                if (![newSelection containsObject:node])
                    [newSelection addObject:node];
            }
            else
            {
                NSMutableArray *leafNodes = [[node allChildLeafs] mutableCopy];
                [leafNodes removeObjectsInArray:newSelection];
                [newSelection addObjectsFromArray:leafNodes];
                [leafNodes release];
            }
        }
    }
    else if ([[treeController selectedObjects] count] == 1)
    {
        [newSelection addObjectsFromArray:[[[treeController selectedObjects]
objectAtIndex:0] allChildLeafs]];
```

```
    }
    [self setSelectedNodes:newSelection];

    // If the selection changed to nothing, do nothing
    if ([[treeController selectedObjects] count] == 0)


return;
```

Note that by using -setSelectedNodes: to completely replace the selection array every time we want to update it, we ensure that the array controller keeps up to date. This is because the array controller uses key-value observing to check for all changes to the selectedNodes key. If we just modified the array directly using -addObject: or -removeObject:, our array controller would never notice because as far as it is concerned the selectedNodes array would not have been changed.

Now fire up the program and try it out. You will see that the table view now works just like the one in Xcode – it only ever displays leaf nodes, and it displays the contents of subfolders along with the regular contents. Nice.

Adding Drag'n'Drop Support

And so we come to the final part of this tutorial – adding drag and drop support to our outline view. In fact, this isn't going to be too difficult, because KBBaseNode provides us with some methods that are very handy for this. We are going to need to implement three datasource methods. To make our outline view a dragging source, we need to implement:

- outlineView:writeItems:toPasteboard:

To make our outline view a dragging destination, we need to implement:

- outlineView:validateDrop:proposedItem:proposedChildIndex:
- outlineView:acceptDrop:item:childIndex:

The first thing we need to do is define a pasteboard type for the data that we will be moving around. Because we are only going to be using this within CODocument, we can keep it private to that class. If we were going to be making the pasteboard public, we would define it in the header file using extern NSString… Here, though, we will just use a simple define in the implementation file. Add the following somewhere at the top of CODocument.m between the imports and @implementation:

```
#define CONodesPboardType
@"CONodesPboardType"
```

We'll also need to tell the outline view that it can accept this pasteboard type. We'll do this as soon as the outline view is unarchived, in –windowControllerDidLoadNib:

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    [super windowControllerDidLoadNib:aController];
    // Add any code here that needs to be executed once the windowController has loaded the
document's window.

    [outlineView registerForDraggedTypes:[NSArray arrayWithObject:CONodesPboardType]];

    if (outlineExpandedState)
    {
```

```
        [outlineView restoreExpandedStateWithArray:outlineExpandedState];
        [outlineExpandedState release];// No longer need the info
    }
}
```

We're also going to need a way to keep track of which nodes are being dragged around at any one time, so that once they are dropped somewhere, we know which nodes we need to delete or move. For that purpose, we will create a new instance variable to hold this information. Add an NSArray instance variable called "draggedNodes" to CODocument.h:

```
@interface CODocument : NSDocument
{
    IBOutlet ESOutlineView *outlineView;
    IBOutlet NSTableView *tableView;
    IBOutlet NSTextView *textView;
    IBOutlet NSTreeController *treeController;
    IBOutlet NSArrayController *arrayController;

    NSMutableArray *contents;
    NSMutableArray *selectedNodes; // Nodes that are shown in the table view

    int uniqueID;
    NSArray *outlineExpandedState;

    NSArray *draggedNodes;
}
```

Now we are ready to make our outline view a dragging source. Add the following method to the bottom of CODocument.m:

```
- (BOOL)outlineView:(NSOutlineView *)ov writeItems:(NSArray*)items
toPasteboard:(NSPasteboard*)pboard
{
    // Save the list of items (don't need to retain as it's just used temporarily while the
drag occurs)
    // (Note that we have to convert this to the observed object because we are using
NSTreeController)
    draggedNodes = [items valueForKey:@"observedObject"];

    // Declare the types we are about to put on the pasteboard
    [pboard declareTypes:[NSArray arrayWithObject:CONodesPboardType] owner:self];

    // Archive the nodes for moving (we must set the data as we can drag to another
document if we want)
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:draggedNodes];
    [pboard setData:data forType:CONodesPboardType];

    return YES;
}
```

So far, so straightforward. First we save the nodes that are being dragged in our draggedNodes array. Notice that because we are using NSTreeController, yet again the items array that is passed in will, in fact, contain objects of type _NSArrayControllerTreeNode. By using valueForKey: with @"observedObject", we can retrieve the array of node items that we want (again, thanks to Scott Stevenson for this tip). Next we declare our pasteboard type, and then we archive the dragged nodes and place them on the pasteboard, so that they are available for any application that can read our pasteboard type.

You can build and run the application if you want. You will see that you can now drag the nodes around – but there is nowhere you can drop them. So we now need to make our outline view into a drag destination. Before we accept any drops, though, we first need to check whether they are valid. There are a couple of things we have to consider here: 1) Clearly, we can only drop into group items. 2) We can not drop an item into one of its descendents. This second one is very important. Imagine you have a folder called "outerFolder" that contains a folder called "innerFolder". You cannot drag "outerFolder" and drop it inside "innerFolder" because it makes no sense – they would both disappear, as they would suddenly become caught in a mobius-like loop: outerFolder is inside innerFolder which is inside outerFolder which is… Well, you get the idea. For this very purpose, KBBaseNode provides a method entitled –isDescendentOfOrOneOfNodes:, which will check to see if the receiver is a descendent of , or one of, the nodes passed in. We will use this to implement our drop validation method:

```
- (unsigned int)outlineView:(NSOutlineView*)ov
                 validateDrop:(id <NSDraggingInfo>)info
                 proposedItem:(id)item
           proposedChildIndex:(int)index
{
     NSPasteboard *pboard = [info draggingPasteboard];

     // Convert item to something useful (because we are using NSTreeController)
     item = [outlineView observedObjectForItem:item];

     // Check drag types
     if ([pboard availableTypeFromArray:[NSArray arrayWithObject:CONodesPboardType]])
     {
          // Ensure the proposed drop index is valid
          if (index == -1)
               return NSDragOperationNone;

          // If we are dragging into the root (contents) - in which case
          // the item will be nil - we are fine
          if (!item)
               return NSDragOperationGeneric;

          // We can only drag into folders
          if (![item isLeaf])
          {
               // If we were dragged from a different outline view, we don't need to do any
more checks - just accept
               if ([info draggingSource] != outlineView)
                    return NSDragOperationGeneric;

               // Otherwise, we have to make sure the drop is valid

               // Don't allow a folder to be dragged inside itself or any of its descendants
               // (We check draggedNodes because we have to check the items that are
currently there)
               if ([item isDescendentOfOrOneOfNodes:draggedNodes])
                    return NSDragOperationNone;

               // If we've got this far, we're good to go
               return NSDragOperationGeneric;
          }
     }

     return NSDragOperationNone;
}
```

Read through the comments and check you understand all of this. It's fairly straightforward. We use our new outline view method, -observedObjectForItem: to convert the target item into something meaningful (from an _NSArrayControllerTreeNode proxy object to a CONode object in this case). Then we make sure that the pasteboard contains the type of data in which we are interested. Next we ensure that the proposed drop index is valid – I don't know if this is really necessary, but it doesn't hurt. Then we check if the target item (the item into which we want to drop) is nil, which indicates that we are dropping on the root – in our case, the CODocument's contents array; if so, we are okay to drop and need no further checks. If that wasn't the case, then we make sure that the drop target is a group and not a leaf. After that, we check to see if we are the dragging source. If not, then the information is coming from a different outline view and we don't have to worry about whether we're trying to drop a node on top of one its descendents or anything else, so we can return that we can drop. Otherwise, we check to ensure that the target item isn't a descendent of any of the nodes that we are trying to drop (note that we use our draggedNodes array for this because we know it must exist because by this point we must be both the source and the destination).

Right, now we are ready to accept the drop. This is the most difficult part, and needs some thought. These are the steps we will need to take to make a successful drop:

1)
First we need to check to see whether we are dropping on the root (which will be our contents array) or on a particular group item. This makes a difference, because the item passed in by -outlineView:accectDrop:item:childIndex will be nil if we are dropping onto the root. So we need to make sure that we set up a targetArray that will point to the correct place.
2)
We will need to check the pasteboard type, to ensure it is one that we can read (CONodesPboardType).
3)
If so, we need to get the data from the pasteboard and unarchive it into an array of nodes that we can use.
4)
Once that's done, we will need to insert the items into our outline view. Handily, the method we use (-outlineView:accectDrop:item:childIndex) passes us the item that should hold the dropped items and the index at which they need to be inserted. We can therefore go through the dropped nodes one by one and insert them. But if we think about this some more, we will realise that every time you insert an object at the same index, it will go *on top* of the last one inserted. So we will have to insert the objects in reverse order, inserting the last one first.
5)
But we can't just insert everything willy-nilly. Imagine that there is a group that contains two notes, and that the group is expanded. The user selects the group and the two notes and drags all three items. What will happen when the items are dropped? If we insert everything, we will end up with two copies of the notes: the group will get inserted, which already contains the two notes as its children, and then the notes will get inserted again. So whenever we go to insert an object, we need to make sure that it isn't already contained inside the children array of one of the groups being dropped. If it is, we don't need to copy it in – it will get added anyway when we add the group that contains it. KBBaseModel has a method entitled –isDescendentOfNodes: for checking precisely this eventuality. (Note that the difference between –isDescendentOfNodes: and –isDescendentOrOneOfNodes: is that –isDescendentOfNodes: doesn't check to see if the receiver is one of the nodes passed into the method, only if it is a descendent of any of the nodes, whereas –isDescendentOrOneOfNodes: checks both.)
6)
Next, we need to check the dragging source. If the node that was just added came from a different outline view (ie. a different project) then we need to assign a new project ID to it (otherwise we could end up with different nodes that have the same ID).
7)

Once we've added all the new nodes, we need to check the dragging source again. If our outline view was the source as well as the destination, we now need to clean up by removing the nodes from their original places. To do this, we can go through our draggedNodes array and first try removing them from the main contents array. In case that didn't work, we can call KBBaseModel's –removeObjectFromChildren: method on all the nodes contained in the contents array. This method will go through all of the descendents of these nodes looking for the object, and if it finds it, it removes it. Note that although at this point we have two copies of the nodes that were dropped, there is no risk of the wrong ones getting deleted. This is because the dropped nodes were archived and then unarchived, so they will no longer be recognised as the same nodes that were placed into the draggedItems array – very convenient.
8)
Once that's done, all that remains is to make sure that the group node onto which we dropped our dragged nodes is expanded so that we can see the results of our drop, and to change the selection so that the dropped nodes are selected.

This is how we do all that in code:

```
- (BOOL)outlineView:(NSOutlineView*)ov acceptDrop:(id <NSDraggingInfo>)info
item:(id)targetItem childIndex:(int)index
{
    int i, n;
    NSPasteboard *pboard = [info draggingPasteboard];         // Get the pasteboard
    CONode *targetNode = [outlineView observedObjectForItem:targetItem];
    NSMutableArray *targetArray = (targetItem) ? [targetNode children] : contents;

    // Check the draggind type
    if ([pboard availableTypeFromArray:[NSArray arrayWithObject:CONodesPboardType]])
    {
        // Read the data
        NSData *data = [pboard dataForType:CONodesPboardType];
        NSArray *newNodes = [NSKeyedUnarchiver unarchiveObjectWithData:data];

        // Add the new items (we do this backwards, otherwise they will end up in reverse
order)
        for (i = ([newNodes count]-1); i >=0; i--)
        {
            // We only want to copy in each item in the array once - if a folder
            // is open and the folder and its contents were selected and dragged,
            // we only want to drag the folder, of course.
            if (![[newNodes objectAtIndex:i] isDescendantOfNodes:newNodes])
            {
                [targetArray insertObject:[newNodes objectAtIndex:i] atIndex:index];

                // For some reason, when using an NSTreeController, it is vital to
refresh the data,
                // otherwise we get strange effects.
                if (targetItem)
                    [outlineView reloadItem:targetItem reloadChildren:[outlineView
isItemExpanded:targetItem]];
                else
                    [outlineView reloadData];

                // Set a unique ID that fits with this document if dragged from another
document
                if ([info draggingSource] != outlineView)
                    [[[newNodes objectAtIndex:i] properties] setValue:[NSNumber
numberWithInt:[self uniqueID]] forKey:@"ID"];
            }
        }
```

```
        // Now delete the originals if dragged from self
        if ([info draggingSource] == outlineView)
        {
            for (i = 0; i < [draggedNodes count]; i++)
            {
                // First, try deleting them from the root folder
                [contents removeObject:[draggedNodes objectAtIndex:i]];

                // In case this didn't work, check all the subfolders
                for (n = 0; n < [contents count]; n++)
                {
                    if(![[contents objectAtIndex:n] isLeaf])
                        [[contents objectAtIndex:n]
removeObjectFromChildren:[draggedNodes objectAtIndex:i]];
                }
            }

            // Reload the outline view
            [outlineView reloadData];
        }

        // Make sure target item is expanded
        if (targetItem)
            [ov expandItem:targetItem];

        // Now go through the outline view and select any items that we just added (note
that
        // we extend the selection only after selecting the first one, so that this
replaces
        // any current selection).
        BOOL extendSelection = NO;
        for (i=[outlineView rowForItem:targetItem]; i<[outlineView numberOfRows]; i++)
        {
            if ([newNodes containsObject:[outlineView observedObjectForItem:[outlineView
itemAtRow:i]]])
            {
                [outlineView selectRow:i byExtendingSelection:extendSelection];
                extendSelection = YES;
            }
        }
        return YES;
    }
    return NO;
}
```

Read through the comments and make sure you understand what is going on here. There are a couple of things to notice:

1)
We are inserting the new nodes and removing the old ones by manipulating the arrays directly – that is, without using any of NSTreeController's insert or remove methods. The reason for this is that, having tested out a lot of different approaches, this seems to be the only one that works reliably. Using the tree controller's insert methods in this situation seems to have problems when inserting objects at the root, so that they don't always get dropped in the right place. It would also be impossible to remove the objects using the tree controller's methods, because there is no way of searching the descendents of an object without doing it directly.
2)
Because we are manipulating the arrays directly, we have to reload the outline view data every time

we insert an object and once we have finished removing objects. Actually, I'm not quite sure why you should need to reload the outline view every time you insert an object. If you weren't using an NSTreeController, it would be enough to refresh the data only once you had finished the whole process. But without this step, I found that you got some very strange results – things wouldn't drop, or they would get copied, or they would disappear entirely. My guess is that unless it's kept up to date, the tree controller automatically removes objects it think shouldn't be there.

Build and run the application. You should now be able to drag and drop items in the outline view to wherever you want.

Making the Table View a Drag Source

One more thing remains. We can drag around the outline view at the moment, but we ought to add some sort of drag support to the table view too for consistency. We're again going to use Xcode as our behaviour model here. It makes no sense for the table view to be a drop destination because it doesn't show structured information; it can show nodes that are contained in various different places in the outline view, so we would no idea of where to put anything that was dropped on it. However, it does make sense for the table view to be a drag *source*. If we drag from the table view to the outline view, that could be a different way of moving the object. This is exactly how Xcode works, of course – you can drag from the table view onto the outline view, but not vice versa.

Thus, we only need to implement the drag source method for our table view:

-tableView:writeRowsWithIndexes:toPasteboard:

Before we can do this, we need to set CODocument as our table view's datasource:

1)
Open CODocument.nib in Interface Builder.
2)
Ctrl-drag from the table view to File's Owner.
3)
Set File's Owner as the datasource in the inspector under "Outlets".
4)
Save and return to Xcode.

Unlike NSOutlineView, when we set a datasource for NSTableView we don't have to create any dummy datasource methods to avoid receiving errors. All we need to do is implement our dragging method in CODocument.m:

```
- (BOOL)tableView:(NSTableView *)tv writeRowsWithIndexes:(NSIndexSet *)rowIndexes
toPasteboard:(NSPasteboard*)pboard
{
    // Save the list of drag items (don't need to retain as it's just used temporarily
while the drag occurs)
    draggedNodes = [selectedNodes objectsAtIndexes:rowIndexes];

    // Declare the types we are about to put on the pasteboard
    [pboard declareTypes:[NSArray arrayWithObject:CONodesPboardType] owner:self];

    // Archive the nodes for moving (we must set the data as we can drag to another
document if we want)
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:draggedNodes];
    [pboard setData:data forType:CONodesPboardType];

    return YES;
```

```
}
```

We'll also need to edit the outline view -acceptDrop:… method a little so that it recognises the table view as a drag source too. In –outineView:acceptDrop:item:childIndex:, change the following lines:

```
// Set a unique ID that fits with this document if dragged from another document
if ([info draggingSource] != outlineView)
	[[[newNodes objectAtIndex:i] properties] setValue:[NSNumber numberWithInt:[self
uniqueID]] forKey:@"ID"];
```

to:

```
// Set a unique ID that fits with this document if dragged from another document
if ( ([info draggingSource] != outlineView) && ([info draggingSource] != tableView) )
	[[[newNodes objectAtIndex:i] properties] setValue:[NSNumber numberWithInt:[self
uniqueID]] forKey:@"ID"];
```

And then change these lines:

```
// Now delete the originals if dragged from self
if ([info draggingSource] == outlineView)
```

to:

```
// Now delete the originals if dragged from self
if ( ([info draggingSource] == outlineView) || ([info draggingSource] == tableView) )
```

Build and run. You can now drag from the table view to the outline view.

Conclusion
That's our application finished. We now have a fully-functional outliner that lets us group and organise notes, drag them around, and save our work.

Possible Refinements
It would be very easy to add a search feature that would filter the table view, using a subclass of NSArrayController (there are examples at http://www.returnself.com and http://homepage.mac.com/mmalc/CocoaExamples/controllers.html).

For the purposes of this tutorial, we have kept things very simple and placed all of our controller code into CODocument. For a larger, more complicated application, you would want to split this controller code up. One way might be to subclass NSArrayController and NSTreeController and place all of the table view datasource and delegate methods into the array controller subclass and all of the outline view datasource and delegate methods, along with –addGroup: and –addNote:, into the tree controller subclass.

Keith Blount
http://www.literatureandlatte.com