My first Yellow Application

‰ric Simenel

Since most of us are in the process of exploring the OpenStep (Yellow Box) framework and its development environment, it's likely that we are going to encounter the similar pitfalls. I thought it would be useful to describe my first experience at writing something more ambitious than a simple snippet, so that you could benefit from my trials. To make the most out of this article, you'll need to already have some knowledge of Yellow Box. This article is not a tutorial, it's more like a journal¼

First, let me give you a little background. I've been programming for the past 17 years (13 on Macintosh) using a lot of languages (in order of appearance: FORTRAN, 6502 asm, BASIC, COBOL, LISP, Pascal and Object Pascal, 68K asm, C and C++, PowerPC asm and Java), and a lot of environments and Operating Systems (micros, minis and mainframes).

As far as frameworks go, I've only used a very little bit of MacApp and more of ODF (the late OpenDoc Development Framework), but mainly I've used the Mac Toolbox when developing company and personal stuff. The main reason is that I've been developing with it for so long that it's just like an old friend; you know what you can ask it to do and what you'd better not.

Since I've been doing Object Oriented Programming for the last 6 years, I've always regretted a little bit that the Mac Toolbox doesn't have OO APIs. Of course, I could have used a framework but this adds an extra layer which you sometimes appreciate and sometimes don't.

Well, with Rhapsody, it's OOP from the word go, so no more excuses.

At first, the Objective-C syntax with all of its ª[ºs and ª]ºs looks really different, but I'd say that after a day or two, you don't even think about it anymore. And anyhow, the Developer Release or the Premier Release will come with the Objective-C modern syntax which will be more familiar to the C/C++ developer.

Although a lot of people admire Interface Builder (IB), and it's indeed a very nice tool, I, for one, think that the real power comes from the Yellow Box framework. When I developed my first Yellow application, I spent maybe 5-10% of my time in IB and the rest in Project Builder (PB). I must add that if IB didn't exist, we would need to spend an incredible amount of time writing source code to do what it does since this tool is much more than a simple graphic user interface designer tool, but more on that below¼

While writing my sample code, I learned quite a few things which I didn't find explicitly in the documentation, so I thought it might be useful to share them. I must add that I later found some (but not all) of this information in documents which I didn't possess at first, and since I believe this will be the case for most of you, and that, anyway, who reads the entire documentation before starting coding, it's still useful to have those points collected in a useful way, i.e. this article.

I guess readers of the late develop magazine will remember that I collect Comics Books, Comics in short, and my non-trivial sample code is the port of an existing Comics Database Management Macintosh application to the Yellow Box. There is an additional document ªThe way I collect Comicsº in the development folder, which explains why I designed the user interface the way it is. This article deals with the ªhowº I designed it.

The development folder containing all the source code, interface files, project files, etc. is on the Rhapsody Developer Release CD, and will also be on our web site, whose address is not known at this date but which I trust you shouldn't have too much difficulty to find. This project has been developed on OpenStep 4.2 prerelease for Intel, so some problems which I had and described below may already have been fixed in the Rhapsody Developer Release that you have in your hands.

This article contains extracts of the source code only when relevant, you might find it useful to read the entire source code along with this article.

What's in a nib (NeXT Interface Builder) file

Everything which has been created in the nib while using IB (i.e., every icon which you see in the ªInstancesº panel), will be instantiated when the nib opens.

That means that when your application launches, its main nib is opened, all objects which have an icon in the ªInstancesº panel will be instantiated, then their `init` method will be called, then the outlet connections you have established with IB will be filled with the appropriate value, and then their `awakeFromNib` method will be called.

As an example, since my Yellow application deals with one unique Comics database object, instead of allocating it in the source code somewhere, I just instantiated it in the nib file. The nib file is not just for graphic user interface elements.

This is also one of the reasons why you should use subprojects with their own nib files. You load these nib files with the `loadNibNamed:` method (in `NSBundle`) when needed which saves on memory usage and improves performance. And speaking of memory¼

Memory problems

Always **<u>when</u>**, and **<u>only</u>** **<u>when</u>**, you **<u>explicitly</u>** call the `alloc` class method, or the `retain`, `copy` or `mutableCopy` instance methods, you must `release` or `autorelease` the object thus allocated. All other objects you get through another method, you **<u>neither</u>** `release` **<u>nor</u>** `autorelease` .

That means that if you write

```
NSAClass *obj = [[NSAClass alloc] init];
```

then you must also do either `[obj release]` if the method where you allocated it is the only one using it, or `[obj autorelease]` if the method is returning this object to somebody else. But if you write, for example,

```
NSFileManager *nsfm = [NSFileManager defaultManager];

NSData *data = [nsfm contents AtPath:fileName];
```

then **don't** write `[nsfm release];` **nor** `[data release];` or you'll end up in trouble (I did).

Also, if you write

```
NSString *str = [[NSString alloc] initWithCString:myCString];
```

then you must write `[str release];` or `[str autorelease];` depending on the usage, but if you write

```
NSString *str = [NSString stringWithCString:myCString];
```

then don't release `str`, (in this case it has already been autoreleased by the class method `stringWithCString:`).

More about memory

All objects in the Yellow Box are refcounted.

The `alloc` class method allocates the object and sets its refcount to 1.

The `retain` instance method increments the refcount

The `release` instance method decrements the refcount and if it becomes 0, deallocates the object.

The `autorelease` instance method places the object in an autorelease pool. When the current event ends, or in other words, when the control returns to the Main Event Loop, all objects in this pool will be sent the `release` message once, and if their refcount becomes 0, they are deallocated.

The `copy` instance method will do the same thing as the `retain` method if the object is immutable. If the object is mutable, `copy` will create a new object with the same content and a brand new refcount of 1, just as `mutableCopy` does.

You should **never** call the `dealloc` method yourself.

Whoever creates an object **or increments** its refcount, by any of the provided means, **is responsible** for decrementing its refcount by calling `release` or `autorelease` . That's the reason why `autorelease` is provided. For example, when you create and return an object to your caller, the caller has no idea how you created the object; it could be an allocation, but it could also be a static or on a stack or whatever. Therefore the caller must **<u>not</u>** `release` this object, **<u>you</u>** have to. But your problem is that you may no longer have access to this object later when the moment comes. What you do, in this case, is to `autorelease` the object before you return it. If the caller wishes to keep it, then the caller must do a `retain` on it, but if the caller doesn't care, then the object will be automatically released at the end of the current event since nobody retained it.

Memory protection

Although when you mismanage memory in the Yellow Box, it's not as bad (directly) as on the Mac OS, you may still have some surprises.

These problems come from releasing an object too many times (see above), going outside the range of an `NSArray` object, and certainly other ways which haven't bitten me yet. In all those cases of memory mismanagement, if you launched your application from the launcher/debugger window in PB, you'll get explicit error or exception messages. You then fix the problem in your source code, build and test again. Afterwards, in some cases, although everything looks fine, you'll experience minor (but irritating) annoyances: PB won't build anymore (the compiler keeps sending obscure messages), PB won't index your project(s) anymore, IB won't save your modifications, or PB or IB will quit suddenly and then refuse to launch again.

Depending on the exact symptom, I found the following solutions: quit PB and IB, relaunch them, everything's fine (this works 50% of the time), if not, log out, log in again (takes just a couple of seconds), relaunch them, everything's fine (this works 45% of the time), if not, power off, then cold start again (this works 5% of the time). In any case, I never lost any data.

On a brighter side, the more I developed with this framework, the less I wrote these silly memory mismanagement mistakes, meaning that I have less and less trouble with the tools themselves. And anyway, those bugs are currently being tracked and will be fixed.

Updating window content

The updating mechanism is very different from what you are used to on Mac OS. By default your window is buffered (as in offscreen), it can also be retained and non-retained, but I must say I was rather surprised by the standard graphic objects reaction to those choices, try it and you'll see what I mean. It looks like some of these objects assume they are living in a buffered window and have their own bit cache management relying on that assumption, and this goes awry whenever the window is not buffered.

All the standard graphic objects call you (as a developer) to be able to redraw, such as `drawRect:` (for an `NSView`) or they will ask for some data as in `willDisplayCell:atRow:column:` (for `NSBrowser`) and then draw.

So far, so good; it's not really different from the Mac OS frameworks we know. The difference comes from the fact that you will not be called at all most of the time. If your window was in the background and comes to the foreground, chances are that the update will occur from the offscreen buffer and your redraw method won't be called at all (so if you put in some debugging display like I did at first, don't be surprised if you don't see it).

Some slightly annoying surprises

Although the APIs of `NSBrowser` and `NSTableView` are very similar, their mechanism is quite different. To use `NSTableView`, you must (among other things) implement the `numberOfRowsInTableView:` method and, as its name suggests, you must return the number of rows of the table. To use `NSBrowser`, you must (among other things) implement the `numberOfRowsInColumn:` method. The difference is that `numberOfRowsInTableView:` is called a **zillion** times by the `NSTableView` object, whereas `numberOfRowsInColumn:` is called only **once** by the `NSBrowser` object whenever you click on a cell in a column which will make the browser fill a new column to its right.

So your strategies as a developer must be quite different. In a browser, it is the right thing to do the computation of the numbers of rows of the column you're being asked for in `numberOfRowsInColumn:`, whereas, in a table, you'd better do the intensive computation **only** at `awakeFromNib` (or `init` time, depending on your situation) time, and then after **only** if the number of rows changes, save off this number of rows in a field of your controller so that you can just return this value when asked in `numberOfRowsInTableView:`.

Setting up IB and PB

The default rules for indenting text in the PB text editor are not necessarily what you'd wish they are. Don't hesitate to modify them in the ªPreferencesº window till the indentation is as you like. You'll need to set up the ªIndentationº panel criteria **and** the ªKey bindingsº panel criteria.

If you happen to close a Palette, using the ªClose Paletteº menu item of the ªPalettesº menu in IB, this action won't close the Palettes window, but remove the current selected palette icon from this window. To reinstall it, use the ªPreferencesº window of IB.

gdb

Since it is rather unlikely that we write perfect source code the first time, it is useful to follow what's happening in the debugger, or at least in the console. That's where gdb (gnu debugger) makes its entrance.

What happens to me the most, and following the works of my colleagues, I found that I am not the only one, is that I design my interface with IB, write my source code with PB, build and test and nothing happens. The stuff I just wrote just does not do what it is supposed to. In nearly all these cases, what's wrong is that I had forgotten to establish crucial connections with IB (such as a delegate or a datasource, or even an action method). Most likely you will experience the same problem.

So the first step is a double one: verify your connections again with IB **and** add some tracing code using `NSLog` (and don't forget to launch your application through the launcher/debugger window of PB, instead of launching it from the Workspace; if you do, however, you can still see your `NSLog` messages in the main console window which you can display from the ªToolsº menu of the Workspace).

The prototype of `NSLog` is

```
(void)NSLog(format, ¼);
```

and you use it as you would use `printf`, except that `NSLog` has an extra format `"%@"`, which enables the printing of the description of an object (very useful).

If you end up being halted in gdb because of an exception or an error, the following useful commands (also accessible from the graphic user interface of the launcher/debugger window) will help you determine the cause of the problem. There are many more commands of course, so read the gdb documentation or use its help command to find out more.

Table 1

Useful commands in gdb

Command	**Description**
`po` <object>	prints the description of an object
`print *` <object>	prints the content of an object
`print *` <structure>	prints the content of a structure

`print` expression	prints the evaluated expression
`bt`	prints the stack crawl
`future-break` <method>	sets a breakpoint for code not loaded yet
`kill`	terminates the process being debugged

Note: You can also evaluate and call methods from known objects such as

`print * [comicsBase titles]` or

```
print * (NSScroller *)[self verticalScroller]
```

Last note

Before we enter the detail of each class in my first Yellow application, I must say that I was pleasantly surprised by the speed of my development.

In less than 3 weeks (and that's my **first** Yellow application, meaning that I spent a lot of time in the documentation trying to understand how everything works), I reached a level (converting data from the Mac OS database, archiving and unarchiving data the Yellow Box way, displaying the database content in 4 different ways, and modifying the data in a rather complex user interface) which had taken me 2 or 3 months to reach in C++ with the Mac Toolbox on Mac OS (remember, this is a port from a Mac OS application which I've been using for years).

So, whenever you hear some guy speaking about Rapid Application Development (RAD) regarding the Yellow Box framework, that's the real untarnished truth, not just some sales talk.

One last thing before we go, the Yellow Box framework, like MacApp, ODF and others, is a Model-View-Controller kind of a framework. Very classic these days, very powerful, and very helpful for object reuse. When I talk later about a View (i.e. user interface), I mean it in the Model-View-Controller context, so don't confuse it with an `NSView` which is a specific class of the Yellow Box.

Onward with the details

ComicsObj

This is the Model part of my application. It only deals with data, not user interface.

The CComics class contains mainly an `NSMutableArray` of CTitles objects which contains mainly an `NSMutableArray` of CIssues objects. The important methods are `initWithCoder:` and `encodeWithCoder:` which are implemented in all 3 classes to allow the archiving and unarchiving of the database (look into the source code to see their implementation, nothing really difficult), `sortArray:withBrand:withSeries:withKind:withState:withSort:` (in CComics) which not only sorts but also extracts the appropriate titles depending on the criteria parameters, and which will be called from nearly all the controller objects of the user interface, and various accessor methods for each class.

Since the comics database will be accessed from nearly all the controller objects, it was expedient to set up a global object (of class CComics) named comicsBase declared in ComicsObj.h. It could also have been an outlet, paired with an accessor, in the main controller (see the above paragraph about nib files), and then accessed from everywhere with `[[NSApp delegate] comicsBase]`. This is mainly a **style** choice from the developer.

Since the comics database already existed on Mac OS and I was reluctant to reenter all the data (see the ªThe way I collect Comicsº document for further explanation), the `init` method of CComics can either convert the Mac OS database or load the Yellow Box database thus:

```
#if realLoad
   self = [NSUnarchiver unarchiveObjectWithFile:fileName];
   [self retain];
   comicsBase = self;
#else
   comicsBase = self;
   [self setTitles:[NSMutableArray array]];
   [self _convertFromMac];
   [self save:nil];
#endif
return self;
```

You'll note that in the first case, `comicsBase` must be assigned only after the unarchiving (`self` is changed), whereas in the other, it must be assigned before the call to `_convertFromMac` which uses this global object.

Since this conversion only has to occur once, unless I destroy the application or have data corruption, I do a special build setting `realLoad` to 0 to convert the database. Afterwards, `realLoad` is set back to 1.

VerifyController (the main controller)

The Controller for the ªVerifyº window which is a View.

This is a very basic use of an `NSTableView`. Just don't forget, in IB, to connect both the `dataSource` and `delegate` outlets of the `NSTableView` object to this controller and implement the 2 minimum requested methods `numberOfRowsInTableView:` and `objectValueForTableColumn:row:` and your table will be fine.

To be able to return appropriate content in `objectValueForTableColumn:row:`, you should also not forget to give an identifier to each column in the IB inspector window; this identifier may or may not be the same as the column title (your choice; in most cases, it's a good idea to give them the same name, it makes source code writing easier later).

Since this controller is the main controller, it also provides the appropriate action methods (to be connected with the appropriate menu items in IB) to deal with the objects from the subprojects (browser, calendar and title longevity). Furthermore, as an `NSApp delegate` (again don't forget to establish the connection in IB), it also implements the `applicationShouldTerminate:` action method which, in this case, will lead to the saving of the comics database if it has been modified.

Each time the user chooses a new choice in the popup buttons, the `selChanged` method is called, which returns a new array of the appropriate CTitle objects, saves the new number of rows (see the above section about the number of rows in an `NSTableView`), and calls the private method `_update` which, by calling the `noteNumberOfRowsChanged` and `reloadData`, will force a redisplay of the table.

Listing 1

Extract from VerifyController.m

```
- (void)selChanged

{

    [comicsBase sortArray:array withBrand:brand withSeries:series
```

```
                 withKind:kind  withState:state   withSort:sort];

    nbRows = [array count];

    [nbSelTitles setIntValue:nbRows];

    [self _update];

}

- (int)numberOfRowsInTableView:(NSTableView *)tableView

{

    return nbRows;

}

- (id)tableView:(NSTableView *)tv objectValueForTableColumn:(NSTableColumn *)tableColumn
```

```
                                                       row:(int)row
{
    CTitle *thisTitle = [array objectAtIndex:row];
    NSString *identifier = [tableColumn identifier];
    if ([identifier isEqualToString:@"Abb"])
        return [thisTitle abb];
    else if ([identifier isEqualToString:@"Title"])
        return [thisTitle title];
    else return [thisTitle listIssues];
}
```

```
- (void)_update

{

    [verifyView noteNumberOfRowsChanged];

    [verifyView reloadData];

}
```

BrowserController

The Controller for the ªBrowserº window which is a View.

This is a very basic use of an `NSBrowser`. Just don't forget, in IB,   to connect the `delegate` outlet of the `NSBrowser` object to this controller (the File's owner icon should be of custom class BrowserController since this is a subproject) and implement the 2 minimum requested methods `numberOfRowsInColumn:` and `willDisplayCell:atRow:column:`, and your browser will be fine.

Also remember to call `[[browser window] makeKeyAndOrderFront:nil];` in the `awakeFromNib` method or else you won't see any window and lose some time understanding why.

Since the browser is defined in a subproject of the application project, we have to load its nib file, when the user wants to see this window. The menu item he will use is connected to the

```
- (void)newBrowser:(id)sender {[[BrowserController alloc] init]; }
```

action method in the VerifyController which is the main controller. The `init` method of

BrowserController will do the nib loading (among other things):

Listing 2

Extract from BrowserController.m

```
- (id)init
{
    if (self = [super init])
      {
        // the browser lies in a subproject, so let's get its nib
        if (![NSBundle loadNibNamed:@"Browser" owner:self])
          {
            NSLog(@"Unable to load Browser.nib");
            [self release];
            return nil;
          }

        array = [[NSMutableArray alloc] initWithCapacity:1500];

      }

    return self;

}

- (int)browser:(NSBrowser *)sender numberOfRowsInColumn:(int)column

{

    short brand, series, state, kind;



    // the first 4 colummns have all only 3 cells
```

```
if (column < 4) return 3;



// if not, let's see what the user has currently selected in the first colums

brand = [sender selectedRowInColumn:0];

series = [sender selectedRowInColumn:1];

state = [sender selectedRowInColumn:2];

kind = [sender selectedRowInColumn:3];



// so that we can retrieve the appropriate titles

// (we don't care for sorting in this browser)
```

```
    [comicsBase sortArray:array withBrand:brand withSeries:series
                 withKind:kind  withState:state   withSort:0];
    return [array count];
}
- (void)browser:(NSBrowser *)sender willDisplayCell:(id)cell
                                              atRow:(int)row
                                             column:(int)column
{
    switch(column)
      {
```

```
case 0: switch(row)
  {
    case 0: [cell setStringValue:@"All Brands"]; break;
    case 1: [cell setStringValue:@"Marvel"]; break;
    case 2: [cell setStringValue:@"DC & Others"]; break;
  }break;
case 1: switch(row)
  {
    case 0: [cell setStringValue:@"All Series"]; break;
    case 1: [cell setStringValue:@"Long"]; break;
    case 2: [cell setStringValue:@"Mini"]; break;
  }break;
case 2: switch(row)
  {
    case 0: [cell setStringValue:@"All States"]; break;
    case 1: [cell setStringValue:@"Dead"]; break;
    case 2: [cell setStringValue:@"Live"]; break;
  }break;
case 3: switch(row)
  {
        case 0: [cell setStringValue:@"All Kinds"]; break;

        case 1: [cell setStringValue:@"Main"]; break;

        case 2: [cell setStringValue:@"Dual"]; break;

      }break;

    case 4:

        [cell setStringValue:[[array objectAtIndex:row] title]];

        break;

  }



// the 5th column is the last
```

```
    [cell setLeaf:(column == 4)];



    // this means this cell is ready for display

    [cell setLoaded: YES];

}
```

TitleLongevityController and TitleLongevityView

The Controller for the ªTitle Longevityº window which is a View, and the custom view inheriting from `NSView` which implements the graph.

This is a very basic use of an `NSView` (`NSCustomView` in IB). The TitleLongevityView is just a graph, and the controller allows the user to change the criteria of the graph (through the popup menu buttons).

TitleLongevityView has to implement just 2 methods (`initWithFrame:` and `drawRect:`) to work fine. Each time a criteria changes, the `display` (of TitleLongevityView, inheriting from `NSView`) method is called, which will, at some point, calls the `drawRect:` method.

Never call `drawRect:` yourself

Before `drawRect:` can be called, the graphic context has to be set right (the Mac OS Toolbox equivalent would be a SetPort). That's what the Window Server does, and what you should never try to attempt yourself. If you want `drawRect:` to be called, call `display` instead, and the Window Server will do the right thing for you.

The only thing you have to pay attention to is the fact that, the origin is in the lower left corner of the view (this is Display Postscript), instead of the upper left as you might be used to in QuickDraw. To draw, you can mix C-ified Postscript calls such as `PSmoveto`, `PSlineto`, `PSstroke` and framework calls such as `drawAtPoint:withAttributes:` (in `NSString`).

Listing 3

Extract from TitleLongevityController.m

```
- (void)brandSelect:(id)sender

{

    [titleLongevity setBrand:[sender indexOfSelectedItem]];

    [titleLongevity display];

    [nbSelTitles setIntValue:[titleLongevity nbSelTitles]];

}
```

```
- (void)seriesSelect:(id)sender
{
    [titleLongevity setSeries:[sender indexOfSelectedItem]];
    [titleLongevity display];
    [nbSelTitles setIntValue:[titleLongevity nbSelTitles]];
}
```

Listing 3 bis

Extract from TitleLongevityView.m

```
- (void)drawRect:(NSRect)rect

{

    NSString *theString;

    short i, j, startEditMonth, lastEditMonth, tlarr[600];



    // get the right array of titles

    [comicsBase sortArray:array withBrand:brand withSeries:series

                 withKind:1     withState:0       withSort:0];

    nbSelTitles = [array count];
```

```
startEditMonth = [comicsBase startEditMonth];

lastEditMonth = [comicsBase lastEditMonth];



// clear and then fill the local array to be graphed

for(i=0; i<(lastEditMonth-startEditMonth); i++) tlarr[i] = 0;

for(i=0; i<nbSelTitles; i++)

  {

    CTitle *thisTitle = [array objectAtIndex:i];

    NSMutableArray *theseIssues = [thisTitle issues];

    // if edited then add 1 if there is an issue for this particular edit month
    if (!editOrLong) for(j=0; j < [thisTitle nbIssues]; j++)

        tlarr[[[theseIssues objectAtIndex:j] editMonth] - startEditMonth] += 1;

    // else add 1 for all months between the first published issue to the latest

    else for(j = [[theseIssues objectAtIndex:0] editMonth];

             j <= [[theseIssues lastObject] editMonth];

             j++)

        tlarr[j-startEditMonth] += 1;

  }



// draw the axes
```

```
PSsetrgbcolor(0, 0, 0);

PSmoveto(orx, ory-3); PSlineto(orx, 700); PSstroke();

PSmoveto(orx-3, ory); PSlineto(700, ory); PSstroke();

// put the labels on vertical axis

for(i=10; i<=130; i+=10)

  {

    PSmoveto(orx-3, ory+i*5); PSlineto(700, ory+i*5); PSstroke();

    theString = [NSString stringWithCString:gnums[i]];

    [theString drawAtPoint:NSMakePoint(orx-20, ory-12+i*5) withAttributes:dictionary];

  }
```

```
// put the labels on horizontal axis

for(i=0; i<(lastEditMonth-startEditMonth+13); i++)

  if (((i + startEditMonth-1) % 12) == 0)

  {

    j = (i + startEditMonth-1) / 12;

    PSmoveto(orx+i, ory); PSlineto(orx+i, ory-3-((j % 2)?7:0)); PSstroke();

    theString = [NSString stringWithCString:gnums[j]];

    [theString drawAtPoint:NSMakePoint(orx+i-9, ory-21-((j % 2)?7:0))

            withAttributes:dictionary];

  }
```

```
    // draw the graph in blue
    PSsetrgbcolor(0, 0, 32767);
    for(i=0; i<(lastEditMonth-startEditMonth); i++)
      {
        PSmoveto(orx+i, ory);
        PSlineto(orx+i, ory+tlarr[i]*5);
        PSstroke();
      }
}
```

CalendarController and CalendarView

The Controller for the ªCalendarº window which is a View, and the custom view inheriting from `NSView` which implements the tabulated display.

This is another very basic use of an `NSView`. The CalendarView is just some text displayed like a table, and the controller allows, again, the user to change the criteria of the display. There is no big difference from the TitleLongevityView except for one trick. At first, I filled the `NSString` object to be displayed with the entire text data (with carriage returns), and had only a single `drawAtPoint:withAttributes:` call. The result was that the lines were reversed from bottom to top (which makes sense when you remember that the origin is in the lower left corner and that the y axis is directed towards the top). Although I could have reversed the text in the `NSString` object, it made more sense (for easy code reading purposes) to display the text one line at a time, correctly anchored where it should be.

Listing 4

Extract from CalendarView.m

```
- (void)drawRect:(NSRect)rect

{
    char strtit[] = "      OCT NOV DEC  97 FEB MAR APR MAY JUN          OCT NOV DEC  97 FEB MAR
APR MAY JUN          OCT NOV DEC  97 FEB MAR APR MAY JUN\n", strlin[200], strstart[7];
    short i, j, k, n, lastmonth, isharr[189][9];


    // get the right array of titles
    [comicsBase sortArray:array withBrand:brand withSeries:series
                 withKind:kind  withState:2       withSort:sort];
    nbSelTitles = [array count];
    lastmonth = [comicsBase lastEditMonth];
```

```
// fill the correct month names or year value in strtit based upon the model above

for(i=1; i<=3; i++) for(j=1; j<=9; j++)

    strncpy(&strtit[6 + (i-1)*45 + (j-1)*4],

            (k = (lastmonth-1-9+j)%12) ? gmonths[k] : gnums[(lastmonth-1-9+j)/12], 3);

// and draw it

[[NSString stringWithCString:strtit] drawAtPoint:NSMakePoint(0,rect.size.height-20)

                                  withAttributes:dictionary];




// clear and fill the local array of issue numbers to be displayed

for(i=0; i<189; i++) for(j=0; j<9; j++) isharr[i][j] = -1;
for(i=0; i<nbSelTitles; i++)
  {
    ... some code irrelevant to the purpose of this article,
    ... see the source code for details
  }
// and display it
for(i=0; i<63; i++)
  {
    strcpy(strlin, "");
    for(j=i; j < nbSelTitles; j += 63)
      {
        strcpy(strstart, [[[array objectAtIndex:j] abb] cString]);
        while (strlen(strstart) < 6) strcat(strstart, " ");
        strcat(strlin, strstart);
        for(k=0; k<9; k++)
            strcat(strcat(strlin,((n = isharr[j][k]) < 0)?"   ":(char *)gnums[n])," ");
        strcat(strlin, "   ");
      }
    [[NSString stringWithCString:strlin]
               drawAtPoint:NSMakePoint(0,rect.size.height-11*(i+3))
```

```
                withAttributes:dictionary];

        }

}
```

InputController

After getting familiar with the framework by doing some basic stuff (what you read above), I eventually reached the level where I wanted to do more ambitious and serious stuff. This Controller (for the ªInputº window View) enables the user to modify the Comics Database in a lot of ways.

First, I started with an `NSTableView`, then implemented the `willDisplayCell:forTableColumn:row:` method which allowed me to control the graphical aspect (such as fonts and colors, including background colors) of a cell, before `objectValueForTableColumn:row:` is called.

Then I implemented the `setObjectValue:forTableColumn:row:` method which allows the user to type in text data in the first 2 columns (the only ones which have been set up as editable in IB).

Listing 5

Extract from InputController.m

```
- (int)numberOfRowsInTableView:(NSTableView *)tableView

{

   return nbRows;

}

- (void)tableView:(NSTableView *)tv willDisplayCell:(id)cell

                                     forTableColumn:(NSTableColumn *)tableColumn

                                                row:(int)row

{
```

```
NSString *identifier = [tableColumn identifier];

[cell setFont:([identifier cString][0] == 'M')?fontNonProp:fontProp];

if ([identifier isEqualToString:@"CurIsh"]) [cell setFont:fontNonProp];

if ([identifier isEqualToString:@"Empty"])

    if (row == [tv selectedRow])

        [cell setBackgroundColor:[NSColor redColor]];

    else

        [cell setBackgroundColor:[NSColor blackColor]];

else if ((row == [tv editedRow]) &&

         ([tv columnWithIdentifier:identifier] == [tv editedColumn]))
        [cell setBackgroundColor:[NSColor whiteColor]];
    else
        [cell setBackgroundColor:arrCol[row % 6]];
    [cell setDrawsBackground:YES];
}
- (id)tableView:(NSTableView *)tv objectValueForTableColumn:(NSTableColumn *)tableColumn
                                                        row:(int)row
{
    CIssue *thisIssue;
    CTitle *thisTitle = [array objectAtIndex:row];
```

```
NSString *identifier = [tableColumn identifier], *result;

thisIssue = [[thisTitle issues] objectAtIndex:[thisTitle findIssue:curIshArray[row]]];

if ([identifier isEqualToString:@"Abb"]) result = [thisTitle abb];

else if ([identifier isEqualToString:@"Title"]) result = [thisTitle title];

else if ([identifier isEqualToString:@"Brand"]) result = [thisTitle brand];

else if ([identifier isEqualToString:@"Series"]) result = [thisTitle series];

else if ([identifier isEqualToString:@"State"]) result = [thisTitle tstate];

else if ([identifier isEqualToString:@"Kind"]) result = [thisTitle kind];

else if ([identifier isEqualToString:@"CurIsh"])

    result = [NSString stringWithCString:gnums[curIshArray[row]]];
```

```
else if ([identifier isEqualToString:@"Grade"]) result = [thisIssue grade];

else if ([identifier isEqualToString:@"Type"]) result = [thisIssue ishtype];

else if ([identifier isEqualToString:@"Content"]) result = [thisIssue content];

else // column identifiers "M1" to "M6"

 {

   short k, j = -1, i = theEditMonth - 6 + [identifier cString][1] - '0';

   for(k = [thisTitle nbIssues]-1; (k >= 0) && (j == -1); k--)

       if (  ([thisIssue = [[thisTitle issues] objectAtIndex:k] editMonth] == i) &&

            !([thisIssue issueFlags] & mskMiss)  )

           j = k;
```

```
        if (j == -1) result = @"";
        else result = [NSString stringWithCString:gnums[[thisIssue issueNumber]]];
      }
    return result;
}
```

At this point, I had a scrolling table with rows of different background colors, and cells with text set up in different fonts, and we could edit textually the first 2 columns.

For the other columns, it made more sense to have popup buttons to allow the user to edit the criteria by choosing rather than by typing. So I implemented a `doClick:` action method which I connected with IB (this name, `doClick`, is purely mine, we can call this method any name we want, the important thing is that it must be connected as a target/action method in IB for the `NSTableView`). In this method, I set up (see the next section) the popup button depending on the row and column of the clicked cell, with both the InputController and PopupInTable classes containing outlets (`puit` and `ic` respectively) pointing at each other (connected with IB as usual). Since this popup button is ªkindaº floating above the `NSTableView`, extra care has to be applied to have a good user experience.

For instance, it disappears when you click elsewhere: `doClick:` calls `releasePub` before it sets up a new one, `_update` (called when the display criteria changed) also calls `releasePub`.

But that's not enough¼ If the popup button is displayed and the user scrolls the table, then the image of the popup button scrolls along, but it really still stays where it was, meaning that we get a ªphantomº undesirable popup button. To have the popup button disappearing when the user scrolls is a little bit more complex. First, in `awakeFromNib`, I parsed the superview hierarchy to get the `NSScrollView`. The documentation states that there is one, but nothing is said about other superviews, and as a matter of fact, between the `NSTableView` and the `NSScrollView`, there is an `NSClipView;` since it is undocumented, I can't rely on the fact that the NSScrollView is 2 levels up in the hierarchy since it could change in the future, hence the little loop:

```
for ( aView = inputView;
      ![aView isKindOfClass:[NSScrollView class]];
      aView = [aView superview] );
```

When I had the `NSScrollView`, I saved the current action method and target outlet of its verticalScroller, and I replaced them with the `userHasScrolled:` action method and `self` (respectively).

Then in `userHasScrolled:`, just call `releasePub` and `sendAction:` to the previously saved action and target of the verticalScroller (or else, our `NSTableView` wouldn't scroll anymore¼).

Listing 6

Extract from InputController.h

```
@interface InputController : NSObject

{

    ...

    NSScroller *myVerticalScroller;

    id saveVerticalScrollerTarget;
    SEL saveVerticalScrollerAction;
    ...
}
```

Listing 6 bis

Extract from InputController.m

```
- (void)awakeFromNib
{
    ...
```

```
NSView *aView;

// finding the NSScrollView superview...
for ( aView = inputView;
      ![aView isKindOfClass:[NSScrollView class]];
     aView = [aView superview]);
myVerticalScroller = [(NSScrollView *)aView verticalScroller];
// saving the current target and action of the vertical scroller
// to be able to call them later in userHasScrolled
saveVerticalScrollerTarget = [myVerticalScroller target];
```

```
    saveVerticalScrollerAction = [myVerticalScroller action];

    // set the new target and action. Much more elegant than patching,

    // much more efficient than subclassing.

    [myVerticalScroller setTarget:self];

    [myVerticalScroller setAction:@selector(userHasScrolled:)];

    ...

}

- (void)userHasScrolled:(id)sender

{

    // if the user has scrolled then release the pop up button
    [puit releasePub];

    // and call back the original target and action of the vertical scroller

    // so that we actually scroll... (see below)

    [myVerticalScroller sendAction:saveVerticalScrollerAction

                            to:saveVerticalScrollerTarget];

}
```

The number of lines of code to achieve this interception mechanism is much less than its explanation, and the target/action mechanism is the real power of the Yellow Box framework. With this mechanism, we can add new behaviors without having to subclass (and in this particular case, the subclassing would not have been trivial nor easy).

PopupInTable

This is the Controller for the ªfloatingº popup menu button.

Disclaimer

This specific usage of a floating popup menu button as I'm using it, is strictly my own user interface. As it goes, some of my colleagues don't like it since they feel it doesn't respect all the proper user interface guidelines. Do not take this sample code as a model for your own user interface in your projects.

The `setUpPopup` method creates the appropriate menu items according to the row and column of the clicked cell, memorizing in a separate array the command and attribute for each menu item (easier than trying to extract the information later from the title of the selected menu item).

Since I wanted some empty separation menu items in the popup, I discovered that you can't add the same title twice (only one will remain), so that's why I use `addItemWithTitle:@""` and then `addItemWithTitle:@" "` and then `addItemWithTitle:@"  "`, etc.

The `pubSelect:` action method modifies the comics database according to the choice selected by the user. The only twist is that, when the user selects the ªOther...º menu item, the cell is made editable and the `editColumn:row:withEvent:select:` method (in `NSTableView`) is called; then the user has to type in the text data which is then analyzed in the `setObjectValue:forTableColumn:row:` method (of InputController).

To prevent endless testing of an empty `issues` array of the CTitle object (in nearly all of the other controllers or CComics and CTitle objects), we forbid the user from deleting the last issue of any title, suggesting deletion of the title itself instead. For the same reason, when the user creates a new title, it comes with a first issue automatically.

Since the source code for both `setUpPopup` and `pubSelect:` is way too long for me to insert in these pages, I invite you, instead, to take a look at it directly.

For speed optimization, since a `reloadData` method call to the `NSTableView` object would be costly (there are a lot of rows and columns all with different fonts and background colors in our `NSTableView` object), and since the deselection of a currently edited row forces the redisplay of that row calling `objectValueForTableColumn:row:` for each cell of this row, I call, where appropriate, the sequence `deselectRow:`/`selectRow:byExtendingSelection:` to always have a valid display of the edited row.

CONCLUSION

You'll find out, using the Yellow Box framework, that you don't have to subclass as much as you would using a C++ framework. The dynamic binding, coming from Objective-C, enables us to rely more on the concepts of target/action, delegation and notification. This dynamic binding has its advantages and its drawbacks: you can inspect objects at runtime to determine their abilities, thus use objects that you never knew about as long as they respond to the appropriate messages, reuse objects much more easily, etc. On the downside, you can't have as much strong type checking at compilation time than you would get with C++, which may lead to interesting experiences at debugging time.

As far as notifications go, the Yellow Box mechanism is very simple to use. Whenever the content of the database changes, the InputController object will send a notification this way:

```
[[NSNotificationCenter defaultCenter]
                          postNotificationName: ComicsDidChange
                                         object: self];
```

Any other Controller (Verify, Calendar, Browser, TitleLongevity), wishing to be informed has just to register for this notification, in its `init` method, for example:

```
[[NSNotificationCenter defaultCenter]
                        addObserver: self
                           selector: @selector(comicsChanged:)
                               name: ComicsDidChange
                             object: nil];
```

then its `comicsChanged:` method will be called to do whatever appropriate to change the display of the window, whenever the InputController sends the notification. The only thing to remember is to unregister this notification in the Controller's `dealloc` method or else the Notification manager will try to send a notification to a released object.

What could be more simple?

Thanks

to our technical reviewers Michelle Wyner, Deborah Grits, Andy Bachorsky, Andy Belk, Alex Dosher, Tony Frey and Randy Nelson.

‰ric Simenel

is really happy he transferred in Cupertino's DTS from Paris' DTS. Aside from the fact that he got a real good welcome from his current colleagues, he's getting much more sun here than there, and, due to his constant location here, he has easier access to Comic Books Conventions where he completed many runs¼ The current mark is at 22,000 and counting.