StartingUpÑWhatHappensinNSApplicationMain();¬Starting Up Ð What Happens in NSApplicationMain()

Every OpenStep application project created through Project Builder has the same **main()** function (in the file **ApplicationName_main.m**). When users double-click an application or document icon in the File Viewer, **main()** (the entry point) is called first; **main()**, in turn, calls **NSApplicationMain()**Ðand that's all it does.

The **NSApplicationMain()** function does what's necessary to get an OpenStep application up and runningÐresponding to events, coordinating the activity of its objects, and so on. The function starts the network of objects in the application sending messages to each other. Specifically, **NSApplicationMain()**:

1 Gets the application's attributes, which are stored in the application wrapper as a property list. From this property list, it gets the names of the main nib file and the principal class (for applications, this is NSApplication or a custom subclass of NSApplication).

2 Gets the Class object for NSApplication and invokes its **sharedApplication** class method, creating an instance of NSApplication, which is stored in the global variable, NSApp. Creating the NSApplication object connects the application to the window system and the Display PostScript server, and initializes its PostScript environment.

3 Loads the main nib file, specifying NSApp as the owner. Loading unarchives and re-creates application objects and restores the connections between objects.

4 Runs the application by starting the main event loop. Each time through the loop, the application object gets the next available event from the Window Server and dispatches it to the most appropriate object in the

application. The loop continues until the application object receives a **stop:** or **terminate:** message, after which the application is released and the program exits.

You can add your own code to **main()** to customize application start-up or termination behavior.

55588_TableRule.eps ¬

OnlyWhenNeeded:DynamicallyLoadingResourcesandCode;¬Only When Needed: Dynamically Loading Resources and Code

As any developer knows well, performance is a key consideration in program design. One factor is the timing of resource allocation. If an application loads all code and resources that it *might* use when it starts up, it will probably be a sluggish, bloated applicationÐand one that takes awhile to launch.

You can strategically store the resources of an application (including user-interface objects) in several nib files. You can also put code that might be used among one or more *loadable bundles*. When the application needs a resource or piece of code, it loads the nib file or loadable bundle that contains it. This technique of deferred allocation benefits an application greatly. By conserving memory, it improves program efficiency. It also speeds up the time it takes to launch the application.

Auxiliary Nib Files

When more sophisticated applications start up, they load only a minimum of resources in the main nib fileÐthe main menu and perhaps a window. They display other windows (and load other nib files) only when users request it or when conditions warrant it.

Nib files other than an application's main nib file are sometimes called auxiliary nib files. There are two general types of *auxiliary nib files*: special-use and document.

Special-use nib files contain objects (and other resources) that *might* be used in the normal operation of the application. Examples of special-use nib files are those containing inspector panels and Info panels. Document nib files contain objects that represent some repeatable entity, such as a word-processor document. A document nib file is a template for documents: it contains the UI objects and other resources needed to make a document.

The Owner of an Auxiliary Nib File

The object that loads a nib file is usually the object that owns it. A nib file's owner must be external to the file. Objects unarchived from the nib file communicate with other objects in the application only through the owner.

In Interface Builder, the File's Owner icon represents this external object. The File's Owner is typically the application controller for special-use nib files, and the document controller for document nib files. The File's Owner object is not really appearing twice; it's created in one file and referenced in the other.

The File's Owner object dynamically loads a nib file and makes itself the owner of that file by sending **loadNibNamed:owner:** to NSBundle, specifying **self** as the second argument.

NSBundle and Bundles

A bundle is a location in the file system that stores code and the resources that go with that code, including images, sounds, and archived objects. A bundle is also identified with an instance of NSBundle, which makes the contents of the bundle available to other objects that request it.

The generic notion of bundles is pervasive throughout OpenStep. Applications are bundles, as are frameworks and palettes. Every application has at least one bundleÐits main bundleÐwhich is the ª.appº directory (or *application wrapper*) where its executable file is located. This file is loaded into memory when the application is launched.

Loadable Bundles

You can organize an application into any number of other bundles in addition to the main bundle and the bundles of linked-in frameworks. Although these loadable bundles usually reside inside the application wrapper, they can be anywhere in the file system. Project Builder allows you to build Loadable Bundle projects.

Loadable bundles differ from nib files in that they don't require you to use Interface Builder to build them. Instead of containing mostly archived objects, they usually contain mostly code. Loadable bundles are especially useful for incorporating extra behavior into an application upon demand. An economic-forecast application, for example, might load a bundle containing the code defining an economic model, but only when users request that model. You could also use loadable bundles to integrate ªplug and playº components into an existing framework.

TD_LoadableBundles.eps ¬

Loadable bundles usually have an extension of ª.bundleº (although that's a convention, not a requirement). Each loadable bundle must have a principal class that mediates between bundle objects and external objects.

440128_TableRule.eps ¬

DatesandTimesinOpenStep;¬Dates and Times in OpenStep

In OpenStep you represent dates and times as objects that inherit from NSDate. The major advantage of dates and times as objects is common to all objects that represent basic values: they yield functionality that, although commonly found in most operating systems, is not tied to the internals of any particular operating-system.

NSDates hold dates and times as values of type NSTimeInterval and express these values as seconds. The NSTimeInterval type makes possible a wide and fine-grained range of date and time values, giving accuracy

within milliseconds for dates 10,000 years apart.

NSDate and its subclasses compute time as seconds relative to an absolute reference date (the first instant of January 1, 2001). NSDate converts all date and time representations to and from NSTimeInterval values that are relative to this reference date.

NSDate provides methods for obtaining NSDate objects (including **date**, which returns the current date and time as an NSDate), for comparing dates, for computing relative time values, and for representing dates as strings.

The NSCalendarDate class, which inherits from NSDate, generates objects that represent dates conforming to western calendrical systems. NSCalendarDate objects also adjust the representations of dates to reflect their associated time zones. Because of this, you can track an NSCalendarDate object across different time zones. You can also present date information from time-zone viewpoints other than the one for the current locale.

Each NSCalendarDate object also has a calendar format string bound to it. This format string contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. NSCalendarDate can interpret user-entered dates that conform to this format string.

NSCalendar has methods for creating NSCalendarDate objects from formatted strings and from component time values (such as minutes, hours, day of week, and year). It also supplements NSDate with methods for accessing component time values and for representing dates in various formats, locales, and time zones.

601243_TableRule.eps ¬

TheStructureofMulti-DocumentApplications;¬The Structure of Multi-Document Applications

From a user's perspective, a document is a unique body of information usually contained by its own window. Users can create an unlimited number of documents and save each to a file. Common documents are word-

processing documents and spreadsheets.

From a programming perspective, a document comprises the objects and resources unarchived from an auxiliary nib file and the controller object that loads and manages these things. This *document controller* is the owner of the auxiliary nib file containing the document interface and related resources.To manage a document, the document controller makes itself the delegate of its window and its ªcontentº objects. It tracks edited status, handles window-close events, and responds to other conditions.

When users choose the New (or equivalent) command, a method is invoked in the application's controller object. In this method, the application controller creates a document-controller object, which loads the document nib file in the course of initializing itself. A document thus remains independent of the application's ªcoreº objects, storing state data in the document controller. If the application needs information about a document's state, it can query the document controller.

When users chose the Save command, the application displays a Save panel and enables users to save the document in the file system. When users chose the Open command, the application displays an Open panel, allowing users to select a document file and open it.

DocCreationSeq.eps ¬

Document Management Techniques

When you make the application controller and the document controller delegates of the application (NSApp) and the document window, they can receive messages sent at critical moments of a running application. These moments include the closure of windows (**windowShouldClose:**), window selection (**windowDidResignMain:**), application start-up (**applicationWillFinishLaunching:**) and application termination (**applicationShouldTerminate:**). In the methods handling these messages, the controllers can then do the appropriate thing, such as saving a document's data or displaying an empty document.

Several NSViews also have delegation messages that facilitate document management, particularly text fields, forms, and other controls with editable text (**controlText...**) and NSText objects (**text...**). One important such message is **textDidChange:** (or **controlTextDidChange:**), which signals that the document's textual content was modified. In responding to this message, controllers can set the window's close button to have a ªbrokenº X with the **setDocumentEdited:** message; later, they can determine whether the document needs to be saved by sending **isDocumentEdited** to the window.

Document controllers often need to communicate with the application controller or other objects in the application. One way to do this is by posting notifications. Another way is to use the key relationships within the core program framework (see page 143) to find the other object (assuming it's a delegate of an Application Kit object). For example, the application controller can send the following message to locate the current document controller:

```
[[NSApp mainWindow] delegate]
```

The document controller can find the application controller with:

```
[NSApp delegate]
```

255213_TableRule.eps ¬

CoordinateSystemsinOpenStep;¬Coordinate Systems in OpenStep

The screen's coordinate system is the basis for all other coordinate systems used for positioning, sizing, drawing, and event handling. You can think of the entire screen as occupying the upper-right quadrant of a two-dimensional coordinate grid. The other three quadrants, which are invisible to users, take negative values along their x-axis, their y-axis, or both axes. The screen's quadrant has its origin in the lower left corner; the positive x-axis extends horizontally to the right and the positive y-axis extends vertically upward. A unit along either axis is expressed as a pixel.

The screen coordinate system has just one function: to position windows on the screen. When your

application creates a new window, it must specify the window's initial size and location in screen coordinates.You can ªhideº windows by specifying their origin points well within one of the invisible quadrants. This technique is often used in off-screen rendering in buffered windows.

ScreenCoords.eps ¬

The reference coordinate system for a window is known as the *base* coordinate system. It differs from the screen coordinate system in only two ways:

- It applies only to a particular window; each window has its own base coordinate system.
- Its origin is at the lower left corner of the window, rather than the lower left corner of the screen. If the window moves, the origin and the entire coordinate system move with it.

CoordSystem.eps ¬

For drawing, each NSView uses a coordinate system transformed from the base coordinate system or from the coordinate system of its superview. This coordinate system also has it origin point at the lower-left corner of the NSView, making it more convenient for drawing operations. NSView has several methods for converting between base and local coordinate systems. When you draw, coordinates are expressed in the application's *current* coordinate system, the system reflecting the last coordinate transformations to have taken place within the current window.

769781_TableRule.eps ¬

TheApplicationQuartet:NSResponder,NSApplication,NSWindow,andNSView;¬The Application Quartet: NSResponder, NSApplication, NSWindow, and NSView

Many classes of the Application Kit stand out in terms of relative importance. NSControl, for example, is the superclass of all user-interface devices, NSText underlies all text operations, and NSMenu has obvious significance. But four classes are at the core of a running application: NSResponder, NSApplication, NSWindow, and NSView. Each of these classes plays a critical role in the two primary activities of an application: drawing the user interface and responding to events. The structure of their interaction is sometimes called the core program framework.

AppQuartet.eps ¬

NSResponder

NSResponder is an abstract class, but it enables event handling in all classes that inherit from it. It defines the set of messages invoked when different mouse and keyboard events occur. It also defines the mechanics of event processing among objects in an application, especially the passing of events up the *responder chain* to each *next responder* until the event is handled. See the ``Events and the Event Cycle'' ;ToDoConcepts.rtfd;linkMarkername EventsandtheEventCycle;¬ for more on the responder chain and a description of *first responder*.

NSApplication

Every application must have one NSApplication object to act as its interface with the Window Server and to supervise and coordinate the overall behavior of the application. This object receives events from the Window Server and dispatches them to the appropriate NSWindows (which, in turn, distribute them to their NSViews). The NSApplication object manages its windows and detects and handles changes in their status as well as in its own status: hidden and unhidden, active and inactive. The NSApplication object is represented in each application by the global variable NSApp. To coordinate your own code with NSApp, you can assign your own custom object as its delegate.

NSWindow

An NSWindow object manages each physical window (that is, each window created by the Window Server) on the screen. It draws the title bar and window frame and responds to user actions that close, move, resize, and otherwise manipulate the window.

The main purpose of an NSWindow is to display an application's user interface (or part of it) in its *content area*: that space below the title bar and within the window frame. A window's content is the NSViews it encloses, and at the root of this *view hierarchy* is the *content view*, which fills the content area. Based on the location of a user event, NSWindows assigns an NSView in its content area to act as *first responder*.

An NSWindow allows you to assign a custom object as its delegate and so participate in its activities.

NSView

Any object you see in a window's content area is an NSView. (Actually, since NSView is an abstract class, these objects are instances of NSView subclasses.) NSView objects are responsible for drawing and for responding to mouse and keyboard events. Each NSView owns a rectangular region associated with a particular window; it produces images within this region and responds to events occurring within the rectangle.

NSViews in a window are logically arranged in a *view hierarchy*, with the *content view* at the top of the hierarchy (see below for more information). An NSView references its window, its superview, and its subviews. It can be the first responder for events or the next responder in the responder chain. An NSView's frame and bounds are rectangles that define its location on the screen, its dimension, and its coordinate system for drawing.

The View Hierarchy

Just inside each window's content areaÐthe area enclosed by the title bar and the other three sides of the frameÐlies the content view. The content view is the root (or top) NSView in the window's view hierarchy.

Conceptually like a tree, one or more NSViews may branch from the content view, one one or more other NSViews may branch from these subordinate NSViews, and so on. Except for the content view, each NSView has one (and only one) NSView above it in the hierarchy. An NSView's subordinate views are called its *subviews*; its superior view is known as the *superview*.

On the screen *enclosure* determines the relationship between superview and subview: a superview encloses its subviews. This relationship has several implications for drawing:

- It permits construction of a superview simply by arrangement of subviews. (An NSBrowser is an instance of a compound NSView.)
- Subviews are positioned in the coordinates of their superview, so when you move an NSView or transform its coordinate system, all subviews are moved and transformed in concert.
- Because an NSView has its own coordinate system for drawing, its drawing instructions remain constant regardless of any change in position in itself or of its superview.

CoreProgram.eps ¬

Fitting Your Application In

The core program framework provides ways for your application to access the participating objects and so to enter into the action.

- The global variable NSApp identifies the NSApplication object. By sending the appropriate message to NSApp, you can obtain the application's NSWindow objects (**windows**), the key and main windows (**keyWindow** and **mainWindow**), the current event (**currentEvent**), the main menu (**mainMenu**), and the application's delegate (**delegate**).

- Once you've identified an NSWindow object, you can get its content view (by sending it **contentView**) and from that you can get all subviews of the window. By sending messages to the NSWindow object you can also get the current event (**currentEvent**), the current first responder (**firstResponder**), and the delegate (**delegate**).
- You can obtain from an NSView most objects it references. You can discover its **window**, its **superview**, and its **subviews**. Some NSView subclasses can also have delegates, which you can access with **delegate**.

By making your custom objects delegates of the NSApplication object, your application's NSWindows, and NSViews that have delegates, you can integrate your application into the core program framework and participate in what's going on.

219710_TableRule.eps ¬

EventsandtheEventCycle;¬Events and the Event Cycle

You can depict the interaction between a user and an OpenStep application as a cyclical process, with the Window Server playing an intermediary role (see illustration below). This cycleÐthe *event cycle*Ðusually starts at launch time when the application (which includes all the OpenStep frameworks it's linked to) sends a stream of PostScript code to the Window Server to have it draw the application interface.

Then the application begins its main event loop and begins accepting input from the user (see facing page). When users click or drag the mouse or type on the keyboard, the Window Server detects these actions and processes them, passing them to the application as events. Often the application, in response to these events, returns another stream of PostScript code to the Window Server to have it redraw the interface.

In addition to events, applications can respond to other kinds of input, particularly timers, data received at a port, and data waiting at a file descriptor. But events are the most important kind of input.

EventCycle.eps ¬

Events

The Window Server treats each user action as an event; it associates the event with a window and reports it to the application that created the window . Events are objects: instances of NSEvent composed from information derived from the user action.

All event methods defined in NSResponder (such as **mouseDown:** and **keyDown:**) take an NSEvent as their argument. You can query an NSEvent to discover its window, the location of the event within the window, and the time the event occurred (relative to system start-up). You can also find out which (if any) modifier keys were pressed (such as Command, Alternate, and Control), the codes identifying characters and keys, and various other kinds of information.

An NSEvent also divulges the type of event it represents. There are many event types (NSEventType); they fall into five categories:

- **Keyboard events.** Generated when a key is pressed down, a pressed key is released, or a modifier key changes. Of these, key-down events are the most useful. When you handle a key-down event, you often determine the character or characters associated with the event by sending the NSEvent a characters message.
- **Mouse events.** Mouse events are generated by changes in the state of the mouse buttons (that is, down and up) for both left and right mouse buttons and during mouse dragging. Events are also generated when the mouse simply moves, without any button pressed.
- **Tracking-rectangle events.** If the application has asked the window system to set a tracking rectangle in a window, the window system creates mouse-entered and mouse-exit events when the cursor enters the rectangle or leaves it.

- **Periodic events.** A periodic event notifies an application that a certain time interval has elapsed. An application can request that periodic events be placed in its event queue at a certain frequency. They are usually used during a tracking loop. (These events aren't passed to an NSWindow.)
- **Cursor-update events.** An cursor-update event is generated when the cursor has crossed the boundary of a predefined rectangular area.

The Event Queue and Event Dispatching

When an application starts up, the NSApplication object (NSApp) starts the main event loop and begins receiving events from the Window Server (see page 111). As NSEvents arrive, they're put in the *event queue* in the order they're received. On each cycle of the loop, NSApp gets the topmost event, analyzes it, and sends an *event message* to the appropriate object. (Event messages are defined by NSResponder and correspond to particular events.) When NSApp finishes processing the event, it gets the next event, and repeats the process again and again until the application terminates.

The object that is ªappropriateº for an event depends on the type of event. NSApp sends most event messages to the NSWindow in which the user action occurred. If the event is a keyboard or mouse event, the NSWindow forwards the message to one of the objects in its view hierarchy: the NSView within which the mouse was clicked or the key was pressed. If the NSView can respond to the eventÐthat is, it accepts first responder status and defines an NSResponder method corresponding to the event messageÐit handles the event.

If the NSView cannot handle an event, it forwards the message to the next responder in the responder chain (see below). It travels up the responder chain until an object handles it.

NSWindow handles some events itself, and doesn't forward them to an NSView, such as window-moved, window-resized, and window-exposed events. (Since these are handled by NSWindow itself, they are not defined in NSResponder.) NSApp also processes a few kinds of events itself; these include cursor-update, and

application-activate and -deactivate events.

EventDispatching.eps ¬

First Responder and the Responder Chain

Each NSWindow in an application keeps track of the object in its view hierarchy that has *first responder* status. This is the NSView that currently receives keyboard events for the window. By default, an NSWindow is its own first responder, but any NSView within the window can become first responder when the user clicks it with the mouse.

You can also set the first responder programmatically with the NSWindow's **makeFirstResponder:** method. Moreover, the first-responder object can be a target of an action message sent by an NSControl, such as a button or a matrix. Programmatically, you do this by sending **setTarget:** to the NSControl (or its cell) with an argument of **nil**. You can do the same thing in Interface Builder by making a target/action connection between the NSControl and the First Responder icon in the Instances display of the nib file window.

Recall that all NSViews of the application, as well as all NSWindows and the application object itself, inherit from NSResponder, which defines the default message-handling behavior: events are passed up the responder chain. Many Application Kit objects, of course, override this behavior, so events are passed up the chain until they reach an object that does respond.

The series of next responders in the responder chain is determined by the interrelationships between the application's NSView, NSWindow, and NSApplication objects. For an NSView, the next responder is usually its superview; the content view's next responder is the NSWindow. From there, the event is passed to the NSApplication object.

For action messages sent to the first responder, the trail back through possible respondents is even more detailed. The messages are first passed up the responder chain to the NSWindow and then to the

NSWindow's delegate. Then, if the previous sequence occurred in the key window the same path is followed for the main window. Then the NSApplication object tries to respond, and failing that, it goes to NSApp's delegate.

684600_TableRule.eps ¬

AShortGuidetoDrawingandCompositing;¬A Short Guide to Drawing and Compositing

Besides responding to events, all objects that inherit from NSView can render themselves on the screen. They do this rendering through image composition and PostScript drawing.

NSViews draw themselves as an indirect result of receiving the **display** message (or a variant of **display**); this message is sent explicitly or through conditions that cause automatic display. The display message leads to the invocation of an NSView's **drawRect:** method and the **drawRect:** methods of all subviews of that NSView. The **drawRect:** method should contain all code needed to redraw the NSView completely.

An NSView can be automatically displayed when:

- Users scroll it (assuming it supports scrolling).
- Users resize or expose the NSView's window.
- The window receives a **display** message or is automatically updated.
- For some Application Kit objects, when an attribute changes.

An NSView represents a context within which PostScript drawing can take place. This context has three components:

- A rectangular frame within a window to which drawing is clipped.

- A coordinate system
- The current PostScript graphics state

Frame and Bounds

An NSView's *frame* specifies the location and dimensions of the NSView in terms of the coordinate system of the NSView's superview. It is a rectangle that encloses the NSView. You can programmatically move, scale, and rotate the NSView by reference to its frame (**setFrameOrigin:**, **setFrameSize:**, and so on).

To draw efficiently, the NSView must have its frame rectangle translated into its own coordinate system. This translated rectangle, suitable for drawing, is called the *bounds*. The bounds rectangle usually specifies exactly the same area as the frame rectangle, but it specifies that area in a different coordinate system. In the default coordinate system, an NSView's bounds is the same as its frame, except that the point locating the frame becomes the origin of the bounds (x = 0.0, y = 0.0). The x- and y-axes of the default coordinate system run parallel to the sides of the frame so, for example, if you rotate the frame the default coordinate system rotates with it.

This relationship between frame and bounds has several implications important in drawing and compositing.

- Each NSView's coordinate system is a transformation of its superview's.
- Drawing instructions don't have to account for an NSView's location on the screen or its orientation.
- Changes in a superview's coordinate system are propagated to its subviews.

NSView allows you to flip coordinate systems (so the positive y-axis runs downward) and to otherwise alter coordinate systems.

FrameAndBounds.eps ¬

Focusing

Before an NSView can draw it must lock focus to ensure that it draws in the correct window, place, and coordinate system. It locks focus by invoking NSView's **lockFocus** method. Focusing modifies the PostScript graphics state by:

- Making the NSView's window the current device
- Creating a clipping path around the NSView's frame
- Making the PostScript coordinate system match the NSView's coordinate system

After drawing, the NSView should unlock focus (**unlockFocus**).

PostScript Drawing

In OpenStep, NSViews draw themselves by sending binary-encoded PostScript code to the Window Server. The Application Kit and the Display PostScript frameworks provide a number of C-language functions that send PostScript code to perform common drawing tasks. You can use these functions in combinations to accomplish fairly elaborate drawing.

The Application Kit has functions and constants, declared in **NSGraphics.h**, for (among other things):

- Drawing, filling, highlighting, clipping and erasing rectangles
- Drawing buttons, bezels, and bitmaps
- Computing window depth and related display information

You also call OpenStep-compliant drawing routines defined in **dpsOpenStep.h**. These routines (such as

DPSDoUserPath()) draw a specified path. In addition, you can call the functions declared in **psops.h**. These functions correspond to single PostScript operators, such as **PSsetgray()** and **PSfill()**.

You can also write and send your own custom PostScript code. **pswrap** is a program (in **/usr/bin**) that converts PostScript code into C-language functions that you can call within your applications. It is an efficient way to send PostScript code to the Window Server. The following pswrap function draws grid lines:

PSCode.tiff ¬

Compose the function in a file with a **.psw** extension and add it to the Other Source project ªsuitcaseº in Project Builder. When you next build your project, Project Builder runs the **pswrap** program, generating an object file and a header file (matching the file name of the **.psw**) file, and links these into the application. To use the code, import the header file and call the function when you want to do the drawing:

```
DrawGrid(5.0, 5.0, 1.0);
```

Compositing Images

The other technique NSViews use to render their appearance is image compositing. By compositing (with the SOVER operator) NSViews can simply display an image within their frame. You usually composite an image using NSImage's **compositeToPoint:** operation: (or a related method).

NSImage allows you to copy images into your user interface. It uses various subclasses of NSImageRep to store the multiple representations of the same imageÐcolor, grayscale, TIFF, EPS, and so onÐand chosing the representation appropriate for a given type or display. NSImage can read image data from a bundle (including the application's main bundle), from the pasteboard, or from an NSData object.

Compositing allows you to do more than simply copy images. Compositing builds a new image by overlaying images that were previously drawn. It's like a photographer printing a picture from two negatives, one placed on top of the other. Various compositing operators (NSCompositingOperation, defined in **dpsOpenStep.h**)

determine how the source and destination images merge.

You can achieve interesting effects with compositing when the initial images are drawn with partially transparent paint. (Transparency is specified by coverage, a PostScript indicator of paint opacity.) In a typical compositing operation, paint that's partially transparent won't completely cover the image it's placed on top of; some of the other image will show through. The more transparent the paint is, the more of the other image you'll see.

CompositeOps.eps ¬

701118_TableRule.eps ¬

MakingaCustomView;¬Making a Custom View

If you want an object that draws itself differently than any other Application Kit object, or responds to events in a special way, you should make a custom subclass of NSView. Your custom subclass should complete at least the steps outlined below.

Note: If you make a custom subclass of any class that inherits from NSView, and you want to do custom drawing or event handling, the basic procedure presented here still applies.

Interface Builder

1. Define a subclass of NSView in Interface Builder. Then generate header and implementation files.

2. Drag a CustomView object from the Views palette onto a window and resize it. Then, with the CustomView object still selected, choose the Custom Class display of the Inspector panel and select the custom class. Connect any outlets and actions.

MakingCustomView.tiff ¬

Initializing Instances

3. Override the designated initializer, **initWithFrame:** to return an initialized instance of self. The argument of this method is the frame rectangle of the NSView, usually as set in Interface Builder (see step 2). You might want to display the custom view at this point.

Handling Events

In the next section, you'll make a subclass of NSButtonCell that uniquely responds to mouse clicks. The way custom NSViews handle events is different. If you intend your custom NSView to respond to user actions you must do a couple of things:

4. Override **acceptsFirstResponder** to return YES if the NSView is to handle selections. (The default NSView behavior is to return NO.)

5. Override the desired NSResponder event methods (**mouseDown:**, **mouseDragged:, keyDown:**, etc.)

```
- (void)mouseDown:(NSEvent *)event {
    if (([event modifierFlags] &
        NSControlKeyMask){
        doSomething();
}
```

You can query the NSEvent argument for the location of the user action in the window, modifier keys pressed, character and key codes, and other information.

Drawing

When you send display to an NSView, its drawRect: method and each of its subview's drawRect: are invoked. This method is where an NSView renders its appearance.

6. Override **drawRect:**. The argument is usually the frame rectangle in which drawing is to occur. This tells the Window Server where the NSView's coordinate system is located. To draw the NSView, you can do one or more of the following:

   - Composite an NSImage.
   - Call Application Kit functions such as **NSRectFill()** and **NSFrameRect ()** (**NSGraphics.h**).
   - Call C functions that correspond to single PostScript operations, such as **PSsetgray()** and **PSfill()**.
   - Call custom drawing functions created with **pswrap**.

See ``A Short Guide to Drawing and Compositing,'' above, for more information on drawing techniques and requirements.

430832_TableRule.eps ¬

WhyChoseNSButtonCellasSuperclass?;¬Why Chose NSButtonCell as Superclass?

ToDoCell's superclass is NSButtonCell. This choice prompts two questions:

- Why a button cell and not the button itself?
- Why this particular superclass?

NSCell defines state as an instance variable, and thus all cells inherit it. Cells instead of controls hold state information for reasons of efficiencyÐone control (a matrix) can manage a collection of cells, each cell with its own state setting. NSButton does provide methods for getting and setting state values, but it accesses the state value of the cell (usually NSButtonCell) that it contains.

NSButtonCell is ToDoCell's superclass because button cells already have much of the behavior you want. By

virtue of inheritance from NSActionCell, button cells can hold target and action information. Button cells also have the unique capability to display an image and text simultaneously. These are all aspects of behavior needed for ToDoCell.

When you think that you need a specialized subclass of an OpenStep class, you should first spend some time examining the header files and reference documentation on not only that class, but its superclasses and any ªsiblingº classes.

86261_TableRule.eps ¬

TickTockBrrrring:RunLoopsandTimers;¬Tick Tock Brrrring: Run Loops and Timers

A run loopÐan instance of NSRunLoopÐmanages and processes sources of input. These sources include mouse and keyboard events from the window system, file descriptor, inter-thread connections (NSConnection), and timers (NSTimer).

Applications typically won't need to either create or explicitly manage NSRunLoop objects. When a thread is created, an NSRunLoop object is automatically created for it. The NSApplication object creates a default thread and therefore creates a default run loop.

NSTimer creates timer objects. A timer object waits until a certain time interval has elapsed and then fires, sending a specified message to a specified object. For example, you could create an NSTimer that periodically sends messages to an object, asking it to respond if an attribute changes.

NSTimer objects work in conjunction with NSRunLoop objects. NSRunLoops control loops that wait for input, and they use NSTimers to help determine the maximum amount of time they should wait. When the NSTimer's time limit has elapsed, the NSRunLoop fires the NSTimer (causing its message to be sent), then checks for new input.