Text System Overview

User Experience: Text Layout

2009-11-30

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures

Introduction to Text System Overview

Text System Overview provides a survey of the Cocoa text system. The articles introduce important features and describe aspects of the text system as a whole.

Who Should Read This Document

Every developer who uses the text system directly should read this document.

To understand the information in this document you should have a general knowledge of Cocoa programming paradigms and, to understand the code examples, familiarity with the Objective-C language.

Organization of This Document

This document contains the following articles:

- "Text System Architecture" (page 9) presents a high-level discussion of the design goals and capabilities of the text system, and it explains the roles of some important text classes.
- "The Cocoa Text System, MLTE, and ATSUI" (page 11) briefly explains the position of the Cocoa text system relative to other Mac OS X text-handling technologies.
- "Typographical Features of the Cocoa Text System" (page 13) explains the concepts of typography—such as glyphs, fonts, typefaces, and layout—that correlate with features of the text system.
- "Text Fields, Text Views, and the Field Editor" (page 19) introduces the main user interface objects of the text system.
- "The Text System and MVC" (page 23) explains how objects of the text system relate to the model-view-controller paradigm of object-oriented programming.
- "Common Configurations" (page 25) describes various ways in which you can configure text system objects to accomplish different text-handling goals.
- "Class Hierarchy of the Cocoa Text System" (page 29) presents an inheritance diagram of the text system classes.
- "Building a Text Editor in 15 Minutes" (page 31) is a tutorial showing how you can quickly and easily create a very capable text editing program using Cocoa.
- "Simple Text Tasks" (page 39) presents programming techniques to accomplish several text-related goals using the text system.
- "Assembling the Text System by Hand" (page 41) shows how to instantiate and explicitly hook together objects of the text system programmatically.

See Also

The following documents discuss specific aspects of the text system architecture in greater detail:

- *Text System User Interface Layer Programming Guide* describes the high-level interface to the Cocoa text system.
- *Text System Storage Layer Overview* discusses the lower-level facilities that the Cocoa text system uses to store text.

For further reading, see other text-related documents in the User Experience topic in the Mac OS X Reference Library. In addition, please refer to the Cocoa text-related code samples in the Sample Code resource type in the library. You can find text-related examples in the Data Management and User Experience topics.

Text System Architecture

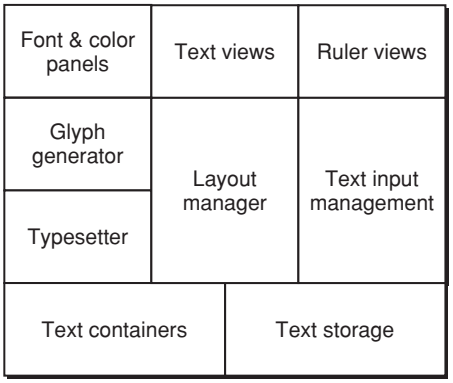
The text-handling component of any application presents one of the greatest challenges to software designers. Even the most basic text-handling system must be relatively sophisticated, allowing for text input, layout, display, editing, copying and pasting, and many other features. But these days developers and users commonly expect even more than these basic features, requiring even simple editors to support multiple fonts, various paragraph styles, embedded images, spell checking, and other features.

The Cocoa text system provides all these basic and advanced text-handling features, and it also satisfies additional requirements that are emerging from our ever more interconnected computing world: support for the character sets of the world's living languages, powerful layout capabilities to handle various text directionality and nonrectangular text containers, and sophisticated typesetting capabilities including control of kerning and ligatures. Cocoa's text system is designed to provide all these capabilities without requiring you to learn about or interact with more of the system than is necessary to meet the needs of your application.

For most developers, the general-purpose programmatic interface of the NSTextView class is all you need to learn. NSTextView provides the user interface to the text system. If you need more flexible, programmatic access to the text, you'll need to learn about the storage layer and the NSTextStorage class. And, of course, to access all the available features, you can learn about and interact with any of the classes that support the text-handling system.

Figure 1 shows the major functional areas of the text system with the user interface layer on top, the storage layer on the bottom, and, in the middle region, the components that interpret keyboard input and arrange the text for display.

Figure 1 Major functional areas of the Cocoa text system



The text classes exceed most other classes in the Application Kit in the richness and complexity of their interface. One of their design goals is to provide a comprehensive set of text-handling features so that you'll rarely need to create a subclass. Among other things, a text object such as NSTextView can:

- Control whether the user can select or edit text.
- Control the font and layout characteristics of its text by working with the Font menu and Font panel (also called the Fonts window).

- Let the user control the format of paragraphs by manipulating a ruler.
- Control the color of its text and background.
- Wrap text on a word or character basis.
- Display graphic images within its text.
- Write text to or read text from files in the form of RTFD—Rich Text Format files that contain TIFF or EPS images, or attached files.
- Let another object, the delegate, dynamically control its properties.
- Let the user copy and paste text within and between applications.
- Let the user copy and paste font and format information between NSText objects.
- Let the user check the spelling of words in its text.

Graphical user-interface building tools (such as Interface Builder) may give you access to text objects in several different configurations, such as those found in the NSTextField, NSForm, and NSScrollView objects. These classes configure a text object for their own specific purposes. Additionally, all NSTextFields, NSForms, NSButtons within the same window—in short, all objects that access a text object through associated cells—share the same text object, called the field editor, reducing the memory demands of an application. Thus, it's generally best to use one of these classes whenever it meets your needs, rather than create text objects yourself. But if one of these classes doesn't provide enough flexibility for your purposes, you can create text objects programmatically.

Text objects typically work closely with various other objects. Some of these—such as the delegate or an embedded graphic object—require some programming on your part. Others—such as the Font panel, spell checker, or ruler—take no effort other than deciding whether the service should be enabled or disabled.

To control layout of text on the screen or printed page, you work with the objects that link the NSTextStorage repository to the NSTextView that displays its contents. These objects are of the NSLayoutManager and NSTextContainer classes.

An NSTextContainer object defines a region where text can be laid out. Typically, a text container defines a rectangular area, but by creating a subclass of NSTextContainer you can create other shapes: circles, pentagons, or irregular shapes, for example. NSTextContainer isn't a user-interface object, so it can't display anything or receive events from the keyboard or mouse. It simply describes an area that can be filled with text. Nor does an NSTextContainer object store text—that's the job of an NSTextStorage object.

A layout manager object, of the NSLayoutManager class, orchestrates the operation of the other text handling objects. It intercedes in operations that convert the data in an NSTextStorage object to rendered text in an NSTextView object's display. It also oversees the layout of text within the areas defined by NSTextContainer objects.

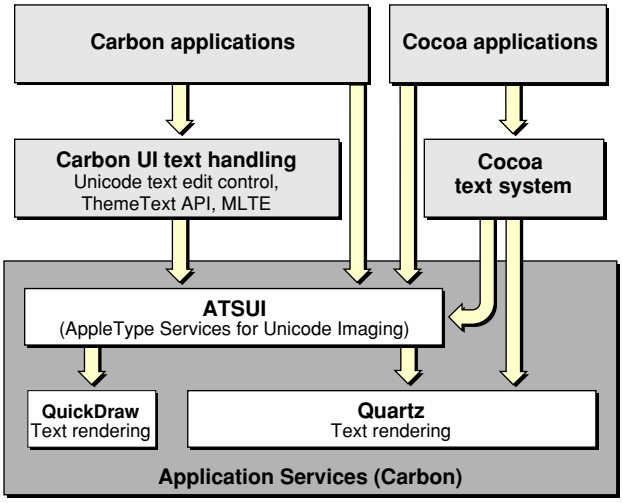
The Cocoa Text System, MLTE, and ATSUI

The Cocoa text system is an object-oriented framework designed to provide all the text services needed by Cocoa applications. Carbon applications, on the other hand, use the APIs of other text-oriented components such as the Multilingual Text Engine (MLTE), which provides editing features, and Apple Type Services for Unicode Imaging (ATSUI), which provides typography and layout services.

From the developer's perspective, the Cocoa text system and ATSUI are two parallel APIs that both handle line layout and character-to-glyph transformations, calling into the Quartz Core Graphics library for text rendering. A developer using the Cocoa text system for a given project is not likely to use ATSUI and vice versa.

The relationships among the Cocoa text system, MLTE, ATSUI, and other text-related components of the Mac OS X development environment are shown in Figure 1.

Figure 1 Text handling in Mac OS X



For more information about the Core Graphics rendering engine, see the Quartz 2D documentation.

Typographical Features of the Cocoa Text System

The Cocoa text system is responsible for the processing and display of all visible text in Cocoa. It provides a complete set of high-quality typographical services through the Application Kit classes. This article defines typographical concepts relevant to the text system.

Characters and Glyphs

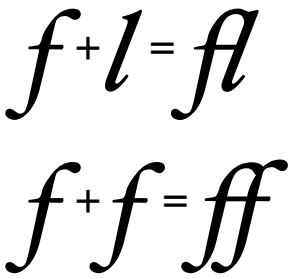
A **character** is the smallest unit of written language that carries meaning. Characters can correspond to a particular sound in the spoken form of the language, as do the letters of the Roman alphabet; they can represent entire words, such as Chinese ideographs; or they can represent independent concepts, such as mathematical symbols. In every case, however, a character is an abstract concept.

Although characters must be represented in a display area by a recognizable shape, they are not identical to that shape. That is, a character can be drawn in various forms and remain the same character. For example, an "uppercase A" character can be drawn with a different size or a different stroke thickness, it can lean or be vertical, and it can have certain optional variations in form, such as serifs. Any one of these various concrete forms of a character is called a **glyph**. Figure 1 shows different glyphs that all represent the character "uppercase A."

Figure 1 Glyphs of the character A

Characters and glyphs do not have a one-to-one correspondence. In some cases a character may be represented by multiple glyphs, such as an "é" which may be an "e" glyph combined with an acute accent glyph "´". In other cases, a single glyph may represent multiple characters, as in the case of a **ligature**, or joined letter. Figure 2 shows individual characters and the single-glyph ligature often used when they are adjacent. A ligature is an example of a contextual form in which the glyph used to represent a character changes depending on the characters next to it. Other contextual forms include alternate glyphs for characters beginning or ending a word.

Figure 2 Ligatures



Computers store characters as numbers mapped by encoding tables to their corresponding characters. The encoding scheme native to Mac OS X is called **Unicode**. The Unicode standard provides a unique number for every character in every modern written language in the world, independent of the platform, program, and programming language being used. This universal standard solves a longstanding problem of different computer systems using hundreds of conflicting encoding schemes. Unicode provides the encoding used to store characters in Cocoa. It also has features that simplify handling bidirectional text and contextual forms.

Glyphs are also represented by numeric codes called glyph codes. The glyphs used to depict characters are selected by the Cocoa layout manager (NSLayoutManager) during composition and layout processing. The layout manager determines which glyphs to use and where to place them in the display, or view. The layout manager caches the glyph codes in use and provides methods to convert between characters and glyphs and between characters and view coordinates. (See "Text Layout" (page 15) for more information about the layout process.)

Typefaces and Fonts

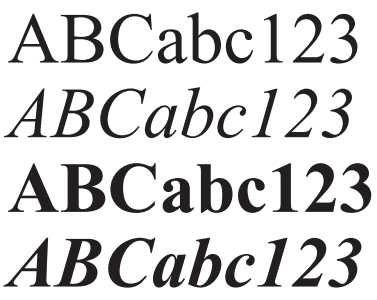
A **typeface** is a set of visually related shapes for some or all of the characters in a written language. For example, Times is a typeface, designed by Stanley Morrison in 1931 for *The Times* newspaper of London. All of the letter forms in Times are related in appearance, having consistent proportions between stems (vertical strokes) and counters (rounded shapes in letter bodies) and other elements. When laid out in blocks of text, the shapes in a typeface work together to enhance readability.

A **typestyle,** or simply style, is a distinguishing visual characteristic of a typeface. For example, roman typestyle is characterized by upright letters having serifs and stems thicker than horizontal lines. In italic typestyle, the letters slant to the right and are rounded, similar to cursive or handwritten letter shapes. A typeface usually has several associated typestyles.

A **font** is a series of glyphs depicting the characters in a consistent size, typeface, and typestyle. A font is intended for use in a specific display environment. Fonts contain glyphs for all the contextual forms, such as ligatures, as well as the normal character forms.

A **font family** is a group of fonts that share a typeface but differ in typestyle. So, for example, Times is the name of a font family (as well as the name of its typeface). Times Roman and Times Italic are the names of individual fonts belonging to the family. Figure 3 shows several of the fonts in the Times font family.

Figure 3 Fonts in the Times font family



In some cases a particular style of a typeface is a separate font, but in other cases where such a font is not available, the system can produce the typestyle programmatically by varying certain characteristics of another font in the family. For example, an algorithm can create a bold typestyle by increasing the line weight of the regular font of the same family. The Cocoa text system can substitute these variations as necessary. Styles, also called traits, that are available in Cocoa include variations such as bold, italic, condensed, expanded, narrow, small caps, poster fonts, and fixed pitch.

Text Layout

Text **layout** is the process of arranging glyphs on a display device, in an area called a text view, which represents an area similar to a page in traditional typesetting. The way in which glyphs are laid out relative to each other is called **text direction**. In English and other languages derived from Latin, glyphs are placed side by side to form words that are separated by spaces. Words are laid out in lines beginning at the top left of the text view proceeding from left to right until the text reaches the right side of the view. Text then begins a new line at the left side of the view under the beginning of the previous line, and layout proceeds in the same manner to the bottom of the text view.

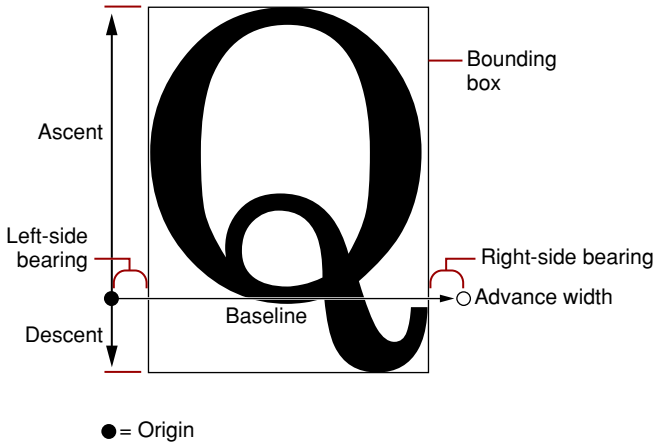
In other languages, glyph layout can be quite different. For example, some languages lay out glyphs from right to left or vertically instead of horizontally. It is common, especially in technical writing, to mix languages with differing text direction, such as English and Hebrew, in the same line. Some writing systems even alternate layout direction in every other line (an arrangement called boustrophedonic writing). Some languages do not group glyphs into words separated by spaces. Moreover, some applications call for arbitrary arrangements of glyphs; a graphic layout may require glyphs to be arranged on a nonlinear path.

The Cocoa layout manager (an instance of the NSLayoutManager class) lays out glyphs along an invisible line called the **baseline**. In Roman text, the baseline is horizontal, and the bottom edge of most of the glyphs rest on it. Some glyphs extend below the baseline, including those for characters like "g" that have **descenders**, or "tails," and large rounded characters like "O" that must extend slightly below the baseline to compensate for optical effects. Other writing systems place glyphs below or centered on the baseline. Every glyph includes an **origin** point that the layout manager uses to align it properly with the baseline.

Glyph designers provide a set of measurements with a font, called **metrics**, which describe the spacing around each glyph in the font. The layout manager uses these metrics to determining glyph placement. In horizontal text, the glyph has a metric called the **advance width**, which measures the distance along the baseline to the origin point of the next glyph. Typically there is some space between the origin point and the left side of the glyph, which is called the **left-side bearing**. There may also be space between the right side of the glyph and the point described by the advance width, which is called the **right-side bearing**. The vertical dimension of the glyph is provided by two metrics called the **ascent** and the **descent**. The ascent is the distance from the origin (on the baseline) to the top of the tallest glyphs in the font. The descent, which is

the distance below the baseline to the bottom of the font's deepest descenders. The rectangle enclosing the visible parts of the glyph is called the **bounding rectangle** or bounding box. Figure 4 illustrates these metrics and similar ones used for vertical glyph placement.

Figure 4 Glyph metrics



By default, typesetters place glyphs side-by-side using the advance width, resulting in a standard interglyph space. However, in some combinations text is made more readable by **kerning**, which is shrinking or stretching the space between two glyphs. A very common example of kerning occurs between an uppercase W and uppercase A, as shown in Figure 5. Type designers include kerning information in the metrics for a font. The Cocoa text system provides methods to turn kerning off, use the default settings provided with the font, or tighten or loosen the kerning throughout a selection of text.

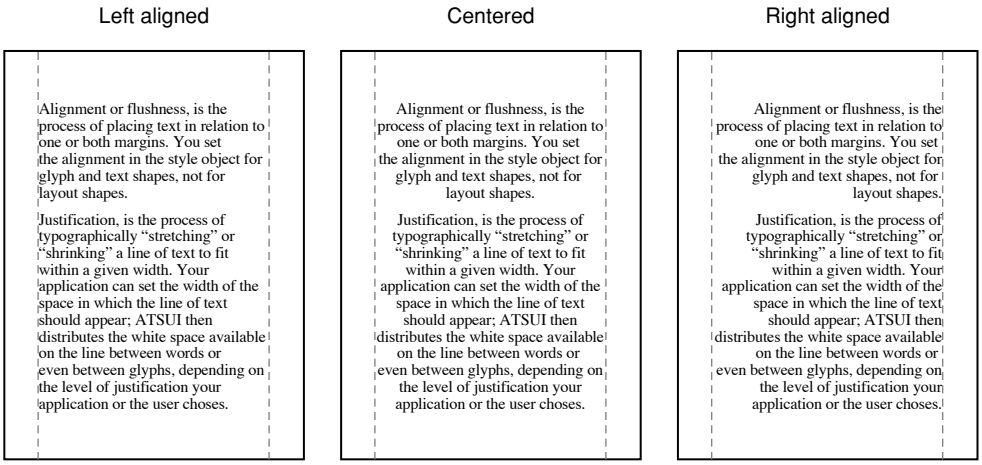
Figure 5 Kerning

Type systems usually measure font metrics in units called **points**, which in Mac OS X measure exactly 72 per inch. Adding the distance of the ascent and the descent of a font provides the font's **point size**.

Space added during typesetting between lines of type is called **leading**, after the slugs of lead used for that purpose in traditional metal-type page layout. (Leading is sometimes also called linegap.) The total amount of ascent plus descent plus leading provides a font's **line height**.

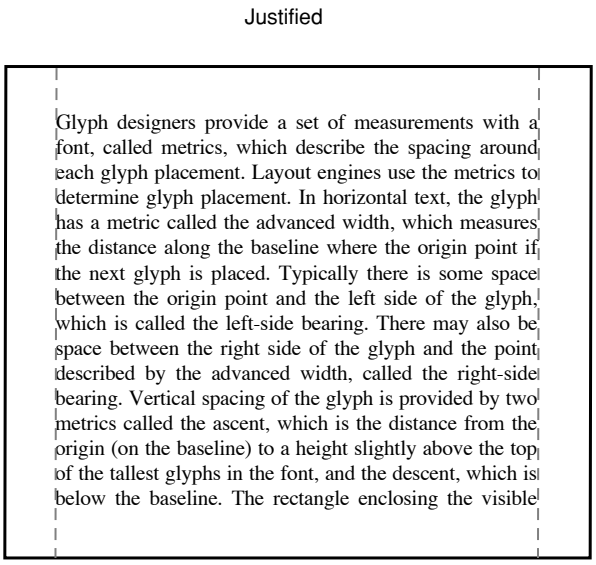
Although the typographic concepts of type design are somewhat esoteric, most people who have created documents on a computer or typewriter are familiar with the elements of text layout on a page. For example, the **margins** are the areas of white space between the edges of the page and the text area where the layout engine places glyphs. **Alignment** describes the way text lines are placed relative to the margins. For example, horizontal text can be aligned right, left, or centered, as shown in Figure 6.

Figure 6 Alignment of text relative to margins



Lines of text can also be **justified**; for horizontal text the lines are aligned on both right and left margins, as shown in Figure 7.

Figure 7 Justified text



To create lines from a string of glyphs, the layout engine must perform **line breaking** by finding a point at which to end one line and begin the next. In the Cocoa text system, you can specify line breaking at either word or glyph boundaries. In Roman text, a word broken between glyphs requires insertion of a hyphen glyph at the breakpoint. Once the text stream has been broken into lines, the system performs alignment and justification, if requested.

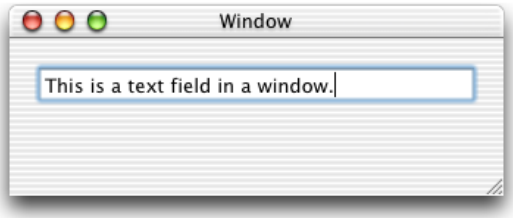
Text Fields, Text Views, and the Field Editor

Text fields, text views, and the field editor are important objects in the Cocoa text system because they are central to the user's interaction with the system. They provide text entry, manipulation, and display. If your application deals in any way with user-entered text, you should understand these objects.

Text Fields

A text field is a user interface control object instantiated from the NSTextField class. Figure 1 shows a text field. Text fields display small amounts of text, typically (although not necessarily) a single line. Text fields also provide places for users to enter text responses, such as search parameters. Like all controls, a text field has a target and an action. By default, text fields send their action message when editing ends—that is, when the user presses Return or moves focus to another control. You can also control a text field's shape and layout, the font and color of its text, background color, whether the text is editable or read-only, whether it is selectable or not (if read-only), and whether the text scrolls or wraps when the text exceeds the text field's visible area.

Figure 1 A text field



To create a secure text field for password entry, you use NSSecureTextField, a subclass of NSTextField. Secure text fields display bullets in place of characters entered by the user, and they do not allow cutting or copying of their contents. You can get the text field's value using the `stringValue` method, but users have no access to the value.

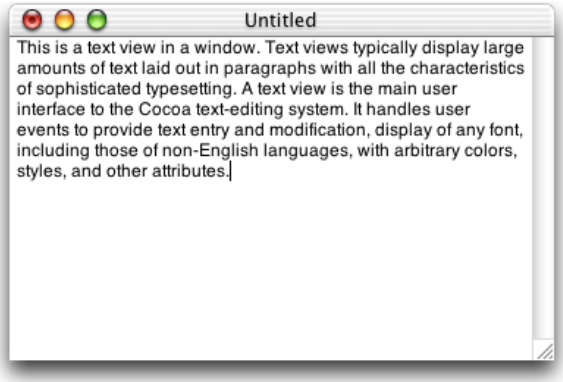
The usual way to instantiate a text field is to drag an NSTextField object from the Cocoa-Views palette in Interface Builder and place it in a window of your application's user interface. Then, if you then want to convert the text field to a secure text field, you select it, open the Info window (Command-Shift-I), choose the Custom Class pane (Command-5), and select NSSecureTextField.

See *Text Fields* more information.

Text Views

Text views are user interface objects instantiated from the NSTextView class. Figure 2 shows a text view. Text views typically display multiple lines of text laid out in paragraphs with all the characteristics of sophisticated typesetting. A text view is the main user interface to the Cocoa text-editing system. It handles user events to provide text entry and modification, and to display any font, including those of non-English languages, with arbitrary colors, styles, and other attributes.

Figure 2 A text view

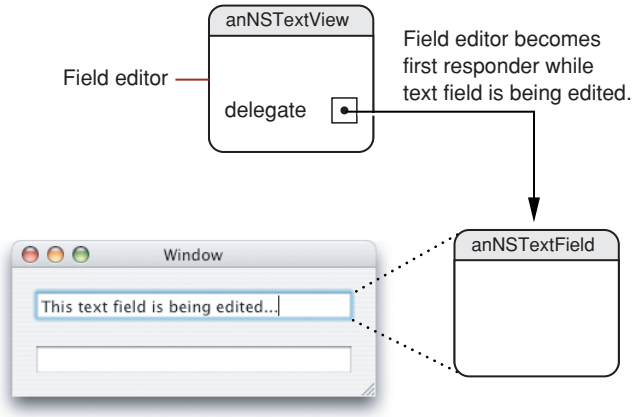


The Cocoa text system supports text views with many other underlying objects providing text storage, layout, font and attribute manipulation, spell checking, undo and redo, copy and paste, drag and drop, saving of text to files, and other features. NSTextView is a subclass of NSText, which is a separate class for historical reasons. You don't instantiate NSText, although it declares many of the methods you use with NSTextView. When you put an NSTextView object in an NSWindow object, you have a full-featured text editor whose capabilities are provided "for free" by the Cocoa text system. (See "Building a Text Editor in 15 Minutes" (page 31) for more information.)

The Field Editor

The field editor is a single NSTextView object that is shared among all the controls, including text fields, in a window. This text view object inserts itself into the view hierarchy to provide text entry and editing services for the currently active text field. When the user shifts focus to a text field, the field editor begins handling keystroke events and display for that field. The field editor designates the current text field as its delegate, enabling the text field to control changes to its contents. When the focus shifts to another text field, the field editor attaches itself to that field instead. Figure 3 illustrates the disposition of the field editor in relation to the text field it is editing.

Figure 3 The field editor



Because only one of the text fields in a window can be active at a time, the system needs only one NSTextView instance per window to be the field editor. Among its other duties, the field editor maintains the selection. Therefore, a text field that's not being edited typically does not have a selection at all. (However, developers can substitute custom field editors, in which case there could be more than one field editor.)

For more information about the field editor, see "Working With the Field Editor."

The Text System and MVC

The Cocoa text system's architecture is both modular and layered to enhance its ease of use and flexibility. Its modular design reflects the model-view-controller paradigm (originating with Smalltalk-80) where the data, its visual representation, and the logic that links the two are represented by separate objects. In the case of the text system, NSTextStorage holds the model's text data, NSTextContainer models the geometry of the layout area, NSTextView presents the view, and NSLayoutManager intercedes as the controller to make sure that the data and its representation onscreen stay in agreement.

This factoring of responsibilities makes each component less dependent on the implementation of the others and makes it easier to replace individual components with improved versions without having to redesign the entire system. To illustrate the independence of the text-handling components, consider some of the operations that are possible using different subsets of the text system:

- Using only an NSTextStorage object, you can search text for specific characters, strings, paragraph styles, and so on.
- Using only an NSTextStorage object you can programmatically operate on the text without incurring the overhead of laying it out for display.
- Using all the components of the text system except for an NSTextView object, you can calculate layout information, determining where line breaks occur, the total number of pages, and so forth.

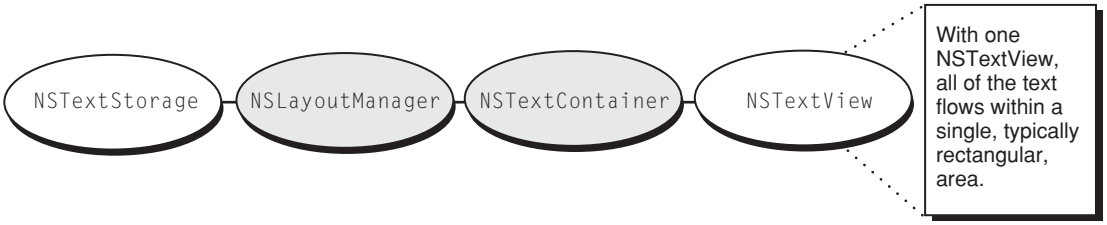
The layering of the text system reduces the amount you have to learn to accomplish common text-handling tasks. In fact, many applications interact with this system solely through the API of the NSTextView class.

Common Configurations

The following diagrams give you an idea of how you can configure objects of the four primary text system classes—NSTextStorage, NSLayoutManager, NSTextContainer, and NSTextView—to accomplish different text-handling goals.

To display a single flow of text, arrange the objects as shown in Figure 1.

Figure 1 Text object configuration for a single flow of text



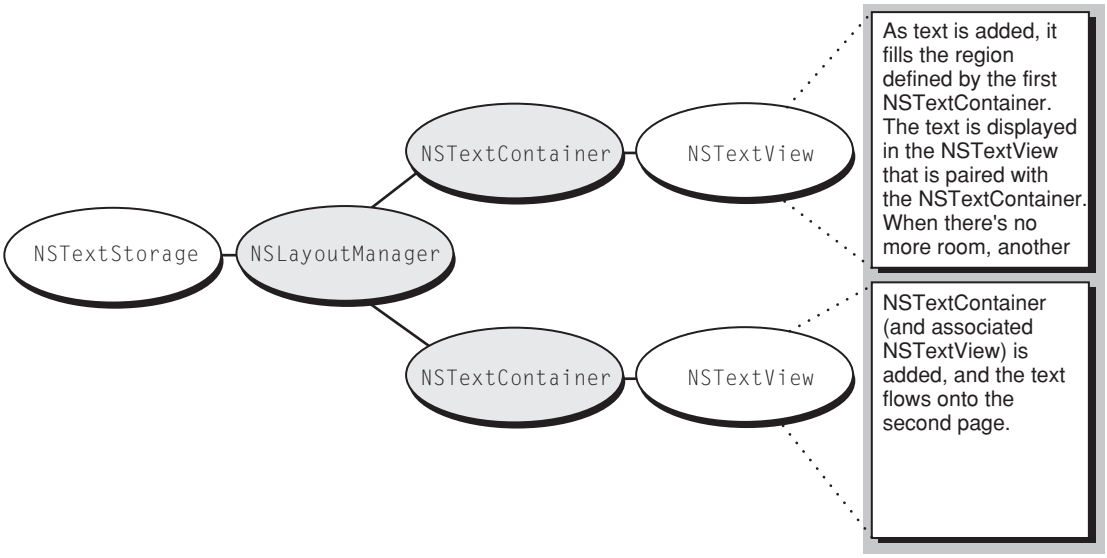
The NSTextView provides the view that displays the glyphs, and the NSTextContainer object defines an area within that view where the glyphs are laid out. Typically in this configuration, the NSTextContainer's vertical dimension is declared to be some extremely large value so that the container can accommodate any amount of text, while the NSTextView is set to size itself around the text using the `setVerticallyResizable:` method defined by NSText, and given a maximum height equal to the NSTextContainer's height. Then, with the NSTextView embedded in an NSScrollView, the user can scroll to see any portion of this text.

If the NSTextContainer's area is inset from the NSTextView's bounds, a margin appears around the text. The NSLayoutManager object, and other objects not pictured here, work together to generate glyphs from the NSTextStorage's data and lay them out within the area defined by the NSTextContainer.

This configuration is limited by having only one NSTextContainer-NSTextView pair. In such an arrangement, the text flows uninterrupted within the area defined by the NSTextContainer. Page breaks, multicolumn layout, and more complex layouts can't be accommodated by this arrangement.

By using multiple NSTextContainer-NSTextView pairs, more complex layout arrangements are possible. For example, to support page breaks, an application can configure the text objects as shown in Figure 2.

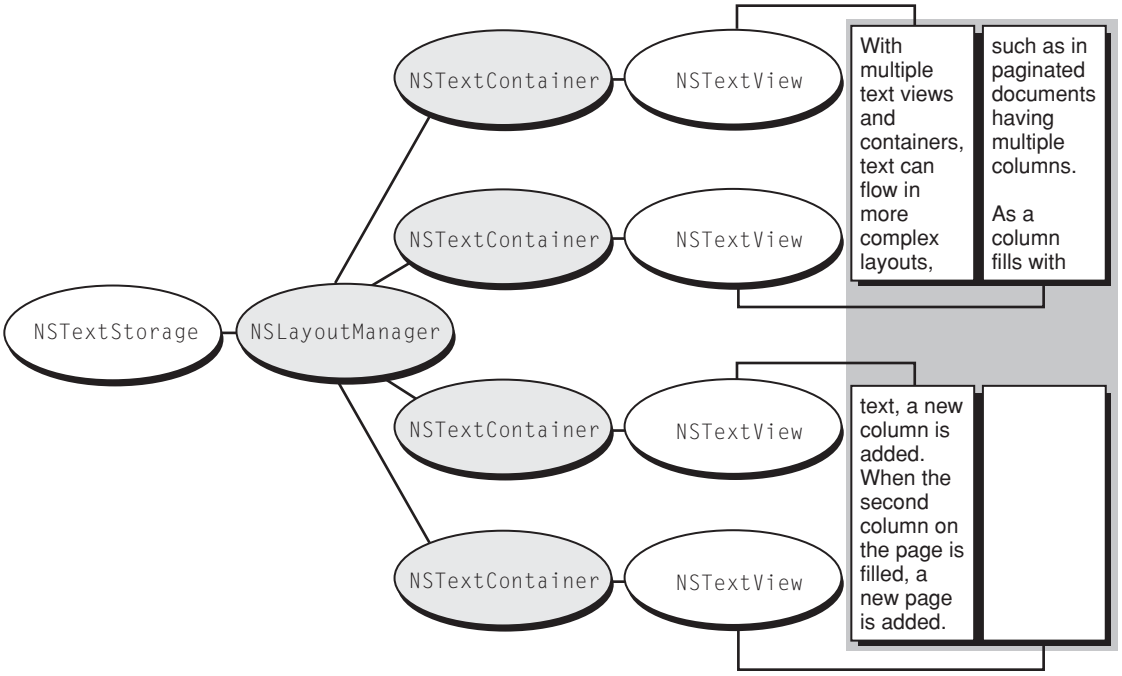
Figure 2 Text object configuration for paginated text



Each NSTextContainer-NSTextView pair corresponds to a page of the document. The gray rectangle in the diagram above represents a custom view object that your application provides as a background for the NSTextViews. This custom view can be embedded in an NSScrollView to allow the user to scroll through the document's pages.

A multicolumn document uses a similar configuration, as shown in Figure 3.

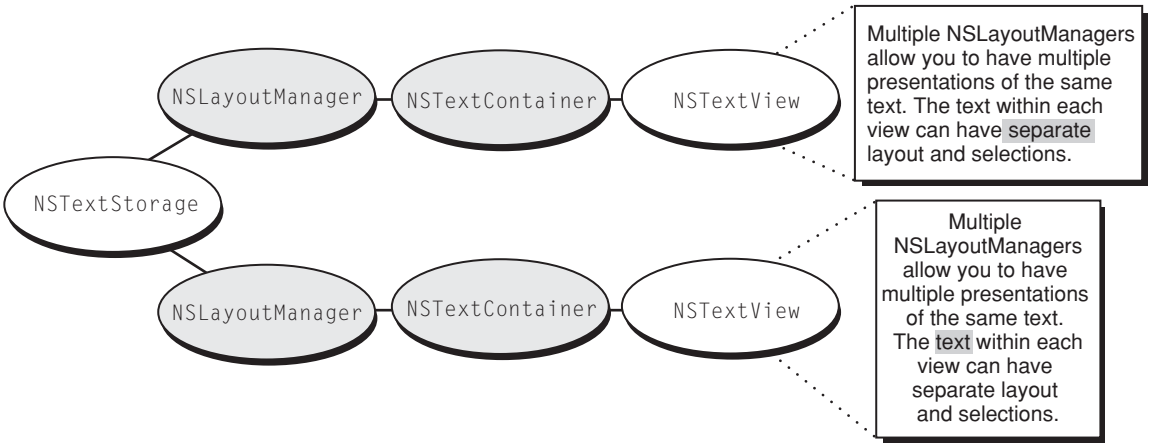
Figure 3 Text object configuration for a multicolumn document



Instead of having one NSTextView-NSTextContainer pair correspond to a single page, there are now two pairs—one for each column on the page. Each NSTextContainer-NSTextView controls a portion of the document. As the text is displayed, glyphs are first laid out in the top-left view. When there is no more room in that view, the NSLayoutManager informs its delegate that it has finished filling the container. The delegate can check whether there's more text that needs to be laid out and add another NSTextContainer and NSTextView. The NSLayoutManager proceeds to lay out text in the next container, notifies the delegate when finished, and so on. Again, a custom view (depicted as a gray rectangle) provides a canvas for these text columns.

Not only can you have multiple NSTextContainer-NSTextView pairs, you can also have multiple NSLayoutManagers accessing the same NSTextStorage. Figure 4 illustrates the simplest arrangement with multiple layout managers.

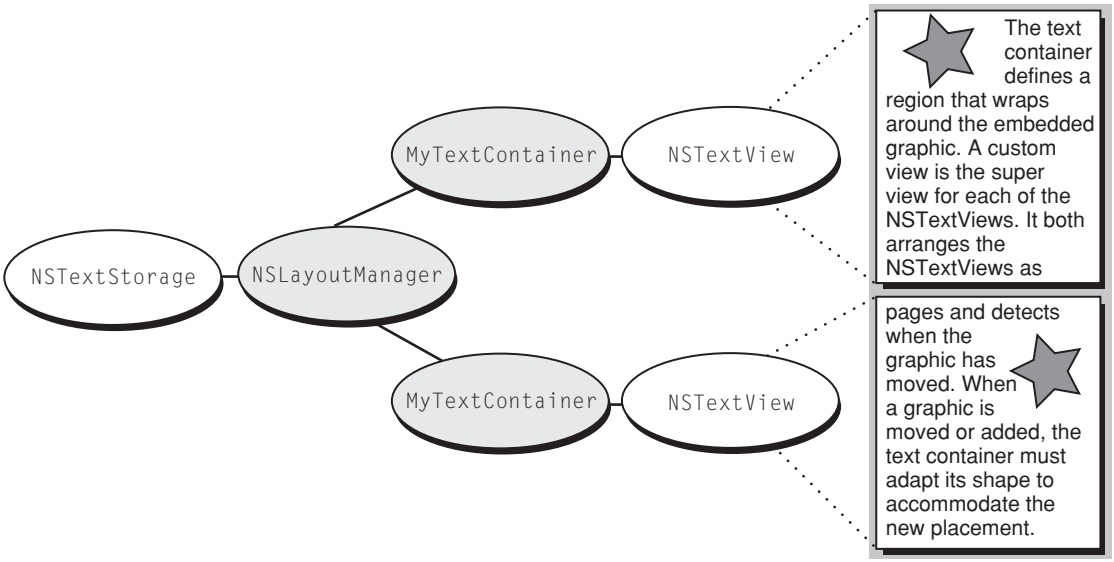
Figure 4 Text object configuration for multiple views of the same text



The effect of this arrangement is to give multiple views on the same text. If the user alters the text in the top view, the change is immediately reflected in the bottom view (assuming the location of the change is within the bottom view's bounds).

Finally, complex page layout requirements, such as permitting text to wrap around embedded graphics, can be achieved by a configuration that uses a custom subclass of NSTextContainer. This subclass defines a region that adapts its shape to accommodate the graphic image and uses the object configuration shown in Figure 5.

Figure 5 Text object configuration with custom text containers



Class Hierarchy of the Cocoa Text System

In addition to the four principal classes in the text system—NSTextStorage, NSLayoutManager, NSTextContainer, NSTextView—there are a number of auxiliary classes and protocols. The diagrams below give you a more complete picture of the text system. Names between angle brackets, such as `<NSCopying>`, are protocols.

Here are some other text-system–related classes:

- NSFileWrapper
- NSInputManager
- NSInputServer
- NSFont
- NSFontPanel
- NSFontManager
- NSFontDescriptor
- NSGlyphGenerator
- NSGlyphInfo
- NSGlyphStorage protocol

- NSRulerView
- NSRulerMarker
- NSTextField
- NSSecureTextField
- NSSpellChecker
- NSTextBlock
- NSTextTable
- NSTextTableBlock
- NSTextList

Building a Text Editor in 15 Minutes

This article shows how you can use Cocoa to create a simple but highly capable text editor in less than 15 minutes using Xcode and Interface Builder. The Cocoa document architecture provides many of the features required for this type of application automatically, requiring you to write a minimal amount of code.

This tutorial uses Xcode version 3.2 and Interface Builder version 3.2 with Mac OS X version 10.6.

Here is a simplified summary of the steps needed to complete this task:

- Use Xcode to create a new document-based application.
- Use Interface Builder to add an `NSTextView` object to the application's window.
- Add some code to the document's controller class.
- Connect the user interface to the code.

You can build and test the application at several stages of completion. The following steps expand and explain this procedure. The steps assume that you have a basic acquaintance with Xcode and Interface Builder.

Create the User Interface

This section shows how to use Xcode and Interface Builder to create the project and build its user interface.

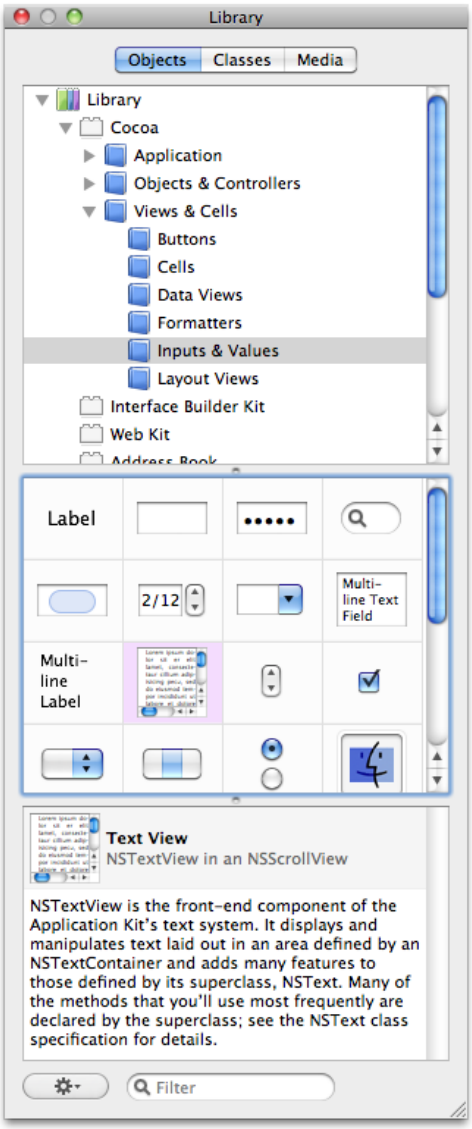
1. In Xcode, create a new project. Choose the Mac OS X template for a Cocoa application, and select the document-based application option.

2. Open the `MyDocument.xib` file, in the Resources folder in the Groups & Files pane of the Xcode window, by double-clicking the `MyDocument.xib` icon, which launches Interface Builder. `MyDocument.xib` is the nib file containing the user interface for your `NSDocument` subclass, named `MyDocument` by the document-based application template.

3. In the Window object, click the text field "Your document contents here" once and press the Delete key to delete the text field. (If you click twice, you begin editing its text content.) If the Library window is not already showing, choose Tools > Library. Click the Objects tab and navigate to Views & Cells > Inputs & Values.

Select a text view as shown in Figure 1. Drag the text view onto the Window object. Resize the text view to almost fill the window, leaving margins as indicated by Interface Builder's guide lines (which appear when the margins are correct).

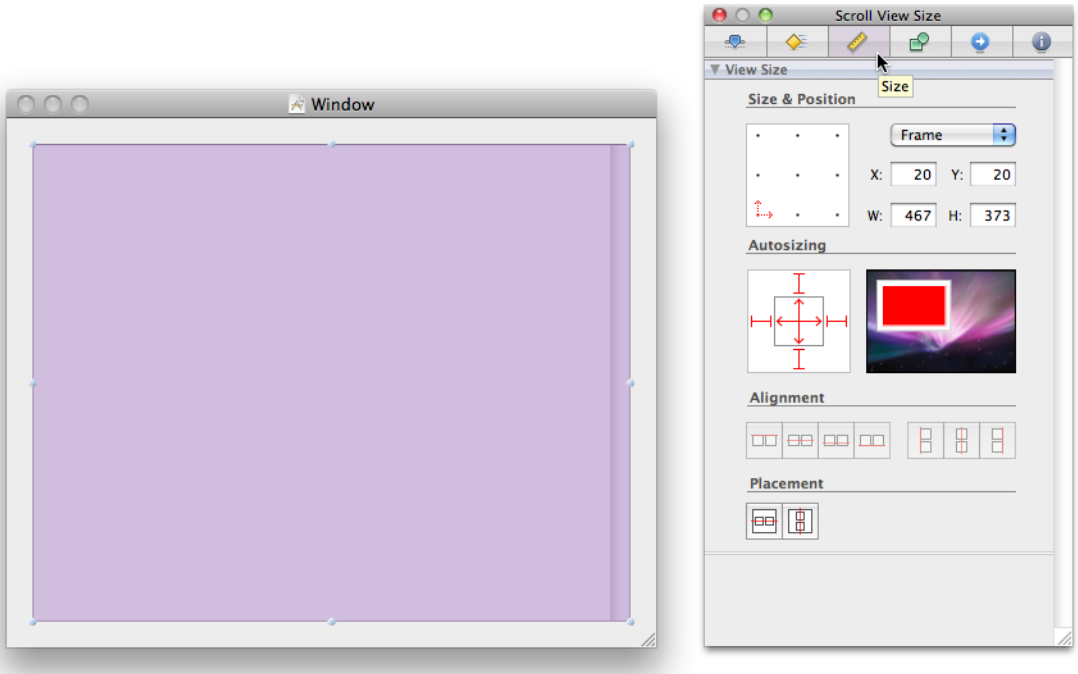
Figure 1 The Text View object selected in the Inputs & Values group



4. Click in the center of the view area and choose Tools > Size Inspector to open the inspector window. Ensure that the inspector window title reads "Scroll View Size." Set the view to resize with the window by clicking both inner sets of crossed arrows in the Autosizing area, as shown in Figure 2 (page 33). The inspector animation should indicate that the scroll view resizes with the window in both vertical and horizontal directions.

Note: Clicking in the center of the view area selects the scroll view containing the text view, as indicated by the title of the inspector window. Clicking inside the view area near the top selects the text view itself. Be sure to set the resize characteristics of the scroll view.

Figure 2 Set the resize characteristics of the scroll view



5. Select the text view by clicking near the top of the view area. Choose Tools > Attributes Inspector (or click the Attributes tab of the inspector window), and ensure that the inspector window title reads "Text View Attributes." Ensure that the Selection pop-up menu is set to Character and the Text Direction pop-up menu is set to Natural. Ensure that the following (default) options are selected: Editable, Selectable, Rich Text, Undo, Non-contiguous Layout, Draws Background, Find Panel, Font Panel, Ruler, Continuous Spell Checking, Smart Insert and Delete, and Autoresizes Subviews.

6. Choose File > Simulate Interface, and resize the window to ensure that the text view resizes properly along with the window. Note that you can already edit text in the text view. When you drag the Text View object from the Library, Interface Builder automatically instantiates all the Cocoa text objects required for a complete editing and layout implementation. Choose Cocoa Simulator > Quit Cocoa Simulator to return to Interface Builder.

7. Save the `MyDocument.xib` file and return to Xcode. Build and test the new application.

At this stage of your editor's development, it has many sophisticated features. You can enter text, which you can then edit, cut, copy, and paste. You can find and replace text using the Find window. You can undo and redo editing actions. You can also format text, setting its font, size, style, and color attributes. You can control text alignment, justification, baseline position, kerning, and ligatures. You can display a ruler that provides a graphical interface to manipulate many text and layout attributes, as well as setting tab stops. You can even use the spelling checker.

In addition to its many editing features, your editor can open multiple documents, each with its own text view and contents. What it lacks most prominently are the abilities to open files and save text in files (that is, archiving and unarchiving documents). It also lacks such features as displaying its own icon in the Finder and presenting useful information in the About window.

Quit your new application before proceeding to the next section.

Implement Document Reading and Writing

This section explains how to enable your editor to open and save documents in files.

1. Add an instance variable for the text view, so you can connect the text view with the code in your `NSDocument` subclass that handles archiving and unarchiving of documents in files. You also need to add an instance variable to hold the text string being edited (the document's data model). In Xcode, put the variable declarations in `MyDocument.h` as follows:

```
#import <Cocoa/Cocoa.h>
@interface MyDocument: NSDocument
{
    IBOutlet NSTextView *textView;
    NSAttributedString *mString;
}
@end
```

2. Initialize the string instance variable. Put the following lines in the `init` method (which has a stub implementation) in `MyDocument.m`:

```
    if (mString == nil) {
        mString = [[NSAttributedString alloc] initWithString:@""];
    }
```

3. Write getter and setter methods for the string instance variable. Put them in `MyDocument.m` as follows:

```
- (NSAttributedString *) string { return [[mString retain] autorelease]; }

- (void) setString: (NSAttributedString *) newValue {
    if (mString != newValue) {
        if (mString) [mString release];
        mString = [newValue copy];
    }
}
```

4. Add method declarations for the getter and setter methods, so that the header file `MyDocument.h` looks like this:

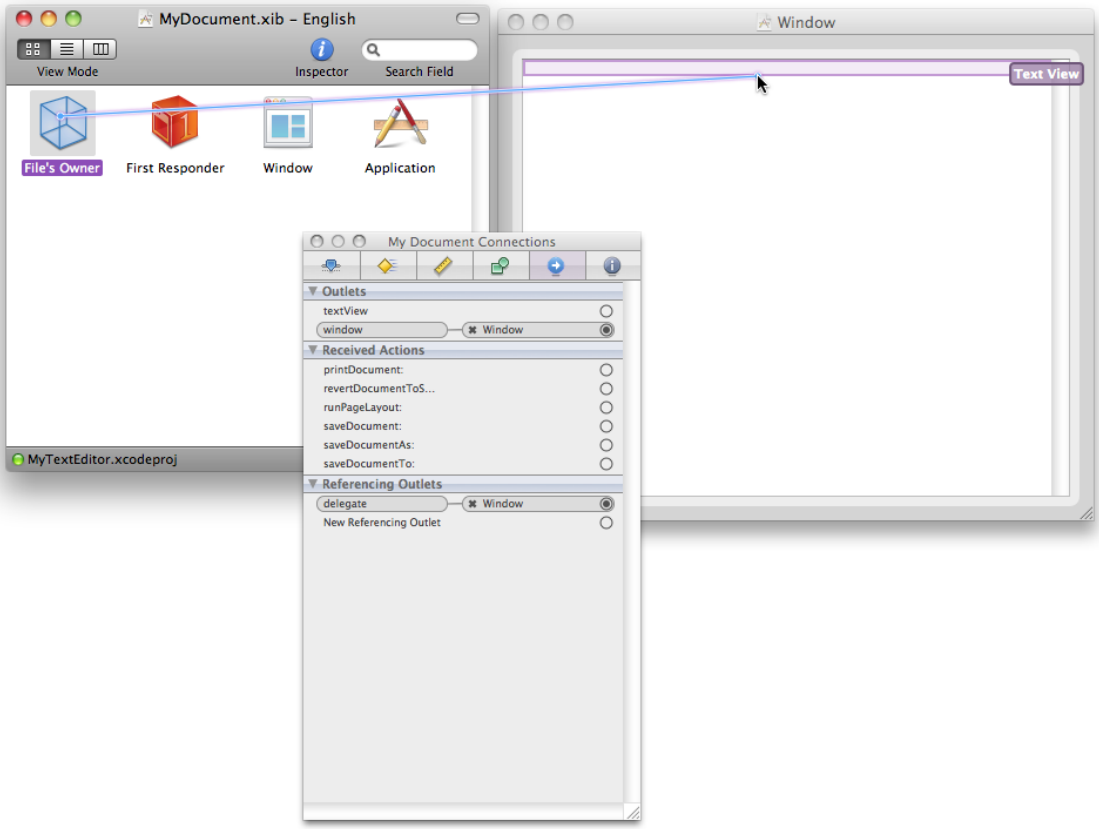
```
#import <Cocoa/Cocoa.h>
@interface MyDocument: NSDocument
{
    IBOutlet NSTextView *textView;
    NSAttributedString *mString;
}
- (NSAttributedString *) string;
- (void) setString: (NSAttributedString *) value;
@end
```

5. If it is not already open, double-click `MyDocument.xib` to open the document window in Interface Builder. From Xcode, drag the `MyDocument.h` file icon onto the document window of `MyDocument.xib`. This step informs the `MyDocument.xib` file that the `MyDocument` object interface now has an outlet variable named `textView`.

6. In the document window of Interface Builder, double-click the Window icon to open the Window object. Open the inspector. In the Window object, click inside the view area at the top to select the text view object. Be sure you select the text view object and not its containing scroll view object, as indicated by the title of the inspector window.

7. Connect the `textView` outlet of the File's Owner (that is, the `MyDocument` object) by Control-dragging from the File's Owner icon in the document window of `MyDocument.xib` to the text view, which is selected in the window, as shown in Figure 3. When the connection is made, Interface Builder displays a list of prospective outlets under the mouse pointer. Click the `textView` outlet. (Alternatively, you can select the File's Owner icon in the document window and use the Connections tab of the inspector to make the connection by dragging from the circle opposite the `textView` outlet to the text view selected in the Window object.)

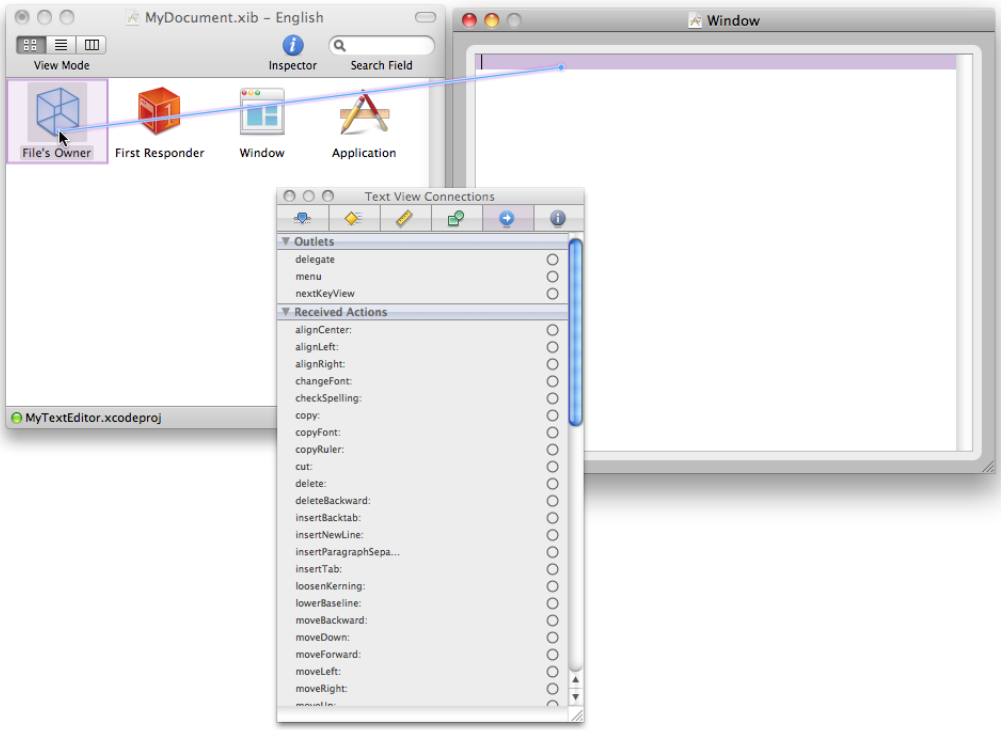
Figure 3 Connect the text view outlet of the File's Owner



8. In Interface Builder, make the File's Owner (that is, the `MyDocument` object) the delegate of the text view. Select the text view in the Window object and Control-drag from the text view to the File's Owner icon, as shown in Figure 4. When the connection is made, Interface Builder displays the text view's outlets under the mouse pointer. Click the delegate outlet. (Alternatively, you can select the text view in the

Window object and use the Connections tab of the Inspector to make the connection by dragging from the circle opposite the `delegate` outlet to the File's Owner icon, which is selected in the document window.)

Figure 4 Connect the delegate of the text view



9. Implement the text view's delegate method `textDidChange:` in `MyDocument.m` to synchronize the text string in the document's data model (the `mString` instance variable) with the text storage belonging to the text view, whenever the user changes the text.

```
- (void) textDidChange: (NSNotification *) notification
{
    [self setString: [textView textStorage]];
}
```

10. Implement document reading and writing methods. When you initially created the project, Xcode placed stubs for these methods in `MyDocument.m`. Fill in the method bodies as follows:

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError **)outError
{
    BOOL readSuccess = NO;
    NSAttributedString *fileContents = [[NSAttributedString alloc]
                initWithData:data options:NULL documentAttributes:NULL
                error:outError];
    if (fileContents) {
        readSuccess = YES;
        [self setString:fileContents];
        [fileContents release];
    }
```

```
    return readSuccess;
}

- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError
{
    NSData *data;
    [self setString:[textView textStorage]];
   NSMutableDictionary *dict = [NSDictionary dictionaryWithObject:NSRTFTextDocumentType
                                        forKey:NSDocumentTypeDocumentAttribute];
    [textView breakUndoCoalescing];
    data = [[self string] dataFromRange:NSMakeRange(0, [[self string] length])
                       documentAttributes:dict error:outError];
    return data;
}
```

11. Add three lines of code to the `windowControllerDidLoadNib:` method to place the contents of the window's data model into the text view when the window's nib file is initially loaded. Leave the call to the superclass `windowControllerDidLoadNib:` method and add the following lines after it:

    ```
    if ([self string] != nil) {
        [[textView textStorage] setAttributedString: [self string]];
    }
    ```

12. Build and test your application.

Your editor should now be able to save documents that you create to files, and it should be able to open those documents again and continue editing them. If you attempt to close a document that has been changed since it was last saved, the editor should display a warning dialog and let you save the document.

At this stage of its development, your editor opens and saves documents only with an extension of `????`. To enable your application to save and open documents with a recognized file type, you need to use Xcode to configure the document types settings in the application's property list file in the Resources folder in Xcode. (The Xcode template names the file with your project name followed by `-Info.plist`.) You can edit this file in Xcode by selecting the file in the Groups & Files list and using the built-in editor. Click the disclosure triangles to edit the value of the first item under `CFBundleTypeExtensions` to the preferred extension for your document files.

For more information about property list files, see "Storing Document Types Information in the Application's Property List" in *Document-Based Applications Overview*. For complete details about application property lists, see *Runtime Configuration Guidelines*.

You can also specify your application's icon file in the application's `Info.plist` file. You can add useful information to be displayed in the About box by editing the `Credits.rtf` file in the Resources folder in Xcode.

For more examples of Cocoa text applications, refer to the Sample Code resource type of the Mac OS X Reference Library. You can find text-related examples in the Data Management and User Experience topics.

Simple Text Tasks

This article explains some programmatic techniques using the Cocoa text system to accomplish simple tasks which may not be obvious until you see how they're done.

Appending Text to a View

This section shows how to use NSTextView methods to append a text string to the text in the view. It also scrolls the text in the view to ensure that the newly appended text is visible.

This code fragment defines a zero-length range of text beginning at the end of the NSTextStorage belonging to the text view. Then it replaces the zero-length range with the string, effectively appending it to the original text storage string, accessed through the text view. Finally, it resets the length of the range to that of the full string in the text view and scrolls the view to make the end of the new range visible.

```
NSTextView *myView;
NSString *myText;
NSRange endRange;

endRange.location = [[myView textStorage] length];
endRange.length = 0;
[myView replaceCharactersInRange:endRange withString:myText];
endRange.length = [myText length];
[myView scrollRangeToVisible:endRange];
```

Setting Font Styles and Traits

This section shows how to programmatically set font styles, such as bold or italic, and font attributes, such as underlining, in an attributed string.

Underlining is an attribute that can be easily set on an attributed string, using the `NSUnderlineStyleAttributeName` constant, as explained in the Cocoa Foundation reference documentation for NSMutableAttributedString. Use the following method:

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)aRange
```

Pass `NSUnderlineStyleAttributeName` for the *name* argument with a *value* of `[NSNumber numberWithInt:1]`.

Unlike underlining, bold and italic are traits of the font, so you need to use a font manager instance to convert the font to have the desired trait, then add the font attribute to the mutable attributed string. For a mutable attributed string named `attributedString`, use the following technique:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];
```

```
unsigned idx = range.location;
NSRange fontRange;
NSFont *font;

while (NSLocationInRange(idx, range)){
    font = [attributedString attribute:NSFontAttributeName atIndex:idx
                           longestEffectiveRange:&fontRange inRange:range];
    fontRange = NSIntersectionRange(fontRange, range);
    [attributedString applyFontTraits:NSBoldFontMask range:fontRange];
    idx = NSMaxRange(fontRange);
}
```

If your mutable attributed string is actually an NSTextStorage object, place this code between `beginEditing` and `endEditing` calls.

Getting the View Coordinates of a Glyph

Glyph locations are figured relative to the origin of the bounding rectangle of the line fragment in which they are laid out. To get the rectangle of the glyph's line fragment in its container coordinates, use

`lineFragmentRectForGlyphAtIndex:effectiveRange:`

Then add the origin of that rectangle to the location of the glyph returned by

`locationForGlyphAtIndex:`

to get the glyph location in container coordinates.

The following code fragment from the CircleView example illustrates this technique.

```
usedRect = [layoutManager usedRectForTextContainer:textContainer];
NSRect lineFragmentRect = [layoutManager lineFragmentRectForGlyphAtIndex:glyphIndex
                           effectiveRange:NULL];
NSPoint viewLocation, layoutLocation = [layoutManager
                           locationForGlyphAtIndex:glyphIndex];
// Here layoutLocation is the location (in container coordinates) where the glyph was
laid out.
layoutLocation.x += lineFragmentRect.origin.x;
layoutLocation.y += lineFragmentRect.origin.y;
```

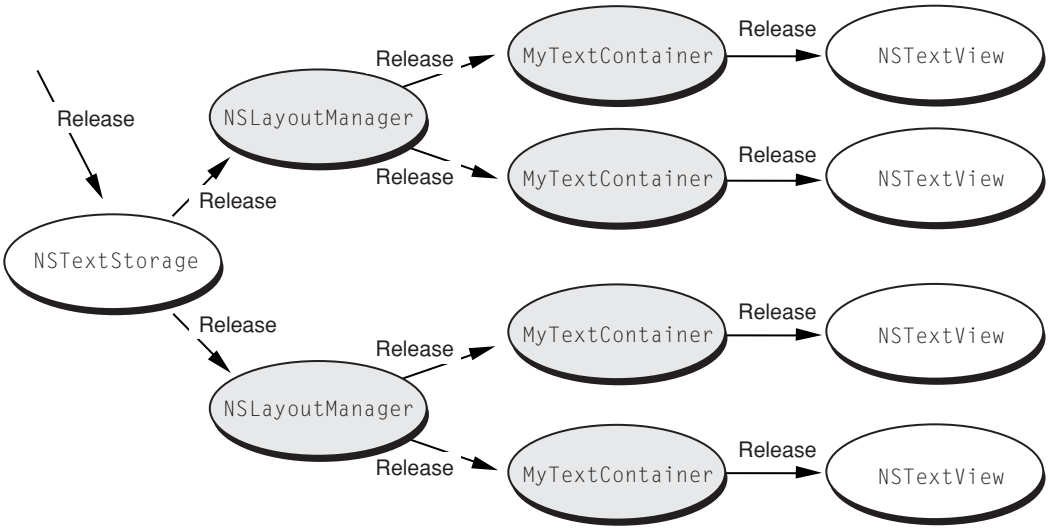
Assembling the Text System by Hand

The vast majority of applications interact with the text system at a high level through one class: NSTextView. It is also possible to build the network of objects that make up the text system from the bottom up, starting with the NSTextStorage object. Understanding how this is done helps illuminate the design of the text-handling system.

In creating the text-handling network by hand, you create four objects but then release three as they are added to the network. You are left with a reference only to the NSTextStorage object. The NSTextView is retained, however, by both its NSTextContainer and its superview; to fully destroy this group of text objects you must send `removeFromSuperview` to the NSTextView object and then release the NSTextStorage object.

An NSTextStorage object is conceptually the owner of any network of text objects, no matter how complex. When you release the NSTextStorage object, it releases its NSLayoutManagers, which release their NSTextContainers, which in turn release their NSTextViews.

Figure 1 Text System Memory Management



However, recall that the text system implements a simplified ownership policy for those whose only interaction with the system is through the NSTextView class. See “Creating an NSTextView Programmatically” for more information.

Set Up an NSTextStorage Object

You create an NSTextStorage object in the normal way, using the `alloc` and `init...` messages. In the simplest case, where there's no initial contents for the NSTextStorage, the initialization looks like this:

```
textStorage = [[NSTextStorage alloc] init];
```

If, on the other hand, you want to initialize an NSTextStorage object with rich text data from a file, the initialization looks like this (assume `filename` is defined):

```
textStorage = [[NSTextStorage alloc]
    initWithRTF:[NSData dataWithContentsOfFile:filename]
    documentAttributes:NULL];
```

We've assumed that `textStorage` is an instance variable of the object that contains this method. When you create the text-handling system by hand, you need to keep a reference only to the NSTextStorage object as you've done here. The other objects of the system are owned either directly or indirectly by this NSTextStorage object, as you'll see in the next steps.

Set Up an NSLayoutManager Object

Next, create an NSLayoutManager object:

```
NSLayoutManager *layoutManager;
layoutManager = [[NSLayoutManager alloc] init];
[textStorage addLayoutManager:layoutManager];
[layoutManager release];
```

Note that `layoutManager` is released after being added to `textStorage`. This is because the NSTextStorage object retains each NSLayoutManager that's added to it—that is, the NSTextStorage object owns its NSLayoutManagers.

The NSLayoutManager needs a number of supporting objects—such as those that help it generate glyphs or position text within a text container—for its operation. It automatically creates these objects (or connects to existing ones) upon initialization. You only need to connect the NSLayoutManager to the NSTextStorage object and to the NSTextContainer object, as seen in the next step.

Set Up an NSTextContainer Object

Next, create an NSTextContainer and initialize it with a size. Assume that `theWindow` is defined and represents the window that displays the text view.

```
NSRect cFrame = [[theWindow contentView] frame];
NSTextContainer *container;

container = [[NSTextContainer alloc]
    initWithContainerSize:cFrame.size];
[layoutManager addTextContainer:container];
[container release];
```

Once you've created the NSTextContainer, you add it to the list of containers that the NSLayoutManager owns, and then you release it. The NSLayoutManager now owns the NSTextContainer and is responsible for releasing it when it's no longer needed. If your application has multiple NSTextContainers, you can create them and add them at this time.

Set Up an NSTextView Object

Finally, create the NSTextView (or NSTextViews) that displays the text:

```
NSTextView *textView = [[NSTextView alloc]
initWithFrame:cFrame textContainer:container];
[theWindow setContentView:textView];
[theWindow makeKeyAndOrderFront:nil];
[textView release];
```

Note that `initWithFrame:textContainer:` is used to initialize the NSTextView. This initialization method does nothing more than what it says: initialize the receiver and set its text container. This is in contrast to `initWithFrame:`, which not only initializes the receiver, but creates and interconnects the network of objects that make up the text-handling system. Once the NSTextView has been initialized, it's added to the window, which is then displayed. Finally, you release the NSTextView.

Document Revision History

This table describes the changes to *Text System Overview*.

Date	Notes
2009-11-30	Corrected example code in "Building a Text Editor in 15 Minutes." Made editorial revisions.
2009-11-17	Updated tutorial for Mac OS X v10.6.
2009-04-08	Added link in "Building a Text Editor in 15 Minutes" to information explaining how to configure an application's document types settings.
2008-10-15	Fixed problem with illustrations.
2007-10-02	Made minor change to code snippet in "Simple Text Tasks" to set font trait.
2007-03-06	Fixed bug in sample code in "Assembling the Text System by Hand."
2006-08-07	Added Mac OS X v10.4 classes to "Class Hierarchy of the Cocoa Text System." Simplified setter method in tutorial.
2005-08-11	Updated the tutorial for Mac OS X v10.4 and made minor revisions throughout. Changed title from Text System Architecture.
2004-05-27	Made editorial revisions to previously unedited articles.
2004-02-10	Updated introduction and tutorial, made minor editorial corrections throughout, and added an index.
2003-04-30	Added five new articles and rewrote introduction.
2002-11-12	Revision history was added to existing topic.

Index

U

V

W

X