Text System Overview

イントロダクション

このドキュメントはCocoaにおけるテキストシステムを概説するものです。その重要なフューチャーを紹介し、包括的な理解を深めます。

このドキュメントを読むべきは

開発しているプログラムでテキストを直接ハンドリングしなければならないプログラマであり、かつCocoaプログラミング・パラダイムに対する一般的知識および、Objective-Cのスキルがある人たちです。

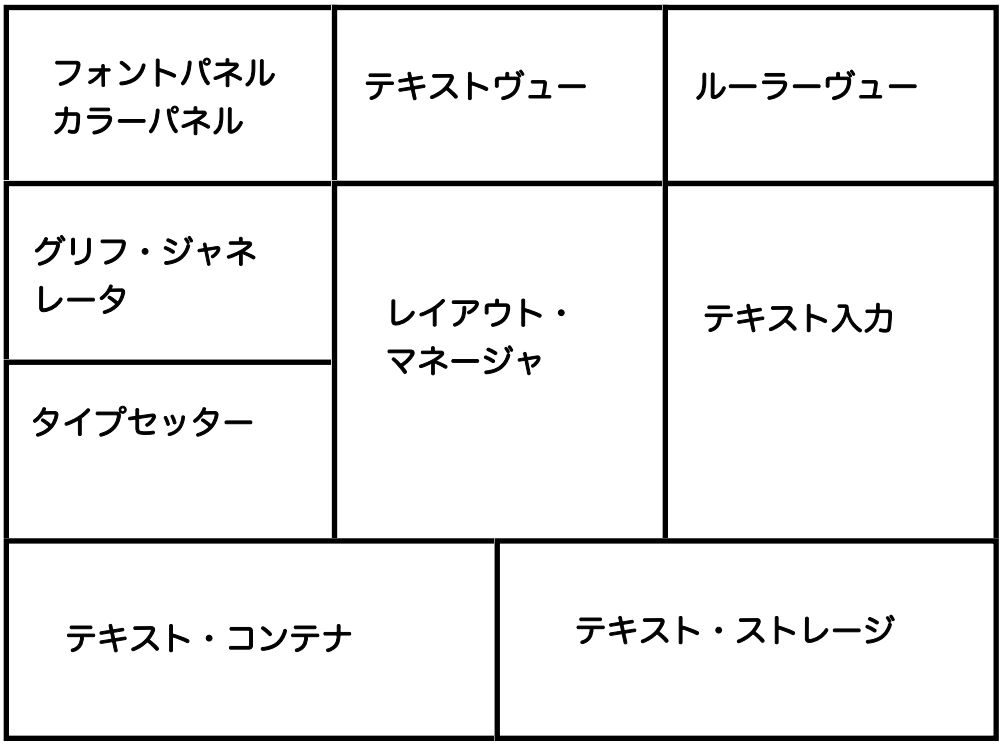
テキストシステム・アーキテクチャー

ソフトウエア設計者にとって、アプリケーションにおけるテキストの取り扱いという問題は、最も頭を悩ませる要素です。最も「基本的な」テキストシステムでも、「入力」、「レイアウト」、「表示」、「編集」、「コピー」、「ペースト」…これらはサポートされていて当たり前です。そのうえ最近は、単なるエディタ（ワードプロセッサではなく！）でさえ複数のフォントやパラグラフ・スタイル、イメージの貼付にスペル・チェックまで要求されるのです。

しかしここにCocoaがあります。Cocoaのテキストシステムは上に挙げた全ての機能を提供します。加えて、相互接続が当然となったコンピュータワールドから次々と生み出される新たな要求：世界各国の固有文字フォントやスクリプトシステムのサポート、非矩形領域へのレイアウト、カーニングやリガチャーを含む洗練されたタイプセットなどに応えます。しかもCocoaのテキストシステムは、これらの実現にあたってプログラマに特別な勉強を要求しません。

ほとんどのデベロッパは、NSTextViewクラスの持つプログラミング・インタフェースさえ学べばそれで十分でしょう。NSTextViewはユーザに対し、テキストシステムへのインタフェースを提供するクラスです。よりフレキシブルなインタフェースをご希望なら、これに加えてNSTextStorageクラスとストレージ・レイヤーについて学んでください。そして、これは当然ですが、Mac OS Xの持つテキストシステムの全ての可能性を余すところなく利用するには、 テキスト・ハンドリング・システムの全てを学ぶ必要があります。

以下は、ユーザインタフェース上層に、ストレージ・レイヤーを下層に配置してテキストシステムの機能を図解したものです。



Application Kitの数あるクラスの中で、テキスト関係のクラスほど多様で複雑なものはありません。その理由は、これらのクラスを利用するプログラマがほとんどサブクラスを作らずに済むような包括的なテキスト・ハンドリングを提供するべく設計されているからです。

例えば、NSTextViewには以下の機能が用意されています：

・ユーザによるテキストの選択（セレクション）と編集の可否制御。
・フォントメニュー、フォントパネルと連動してテキストのフォント、レイアウト統御。
・ルーラによってユーザによるパラグラフ書式指定。
・テキストの表示色、およびその背景色を指定。
・単語、あるいは文字ベースの折り返し（ラッピング）。
・テキスト中へのグラフィック・イメージの表示。
・TIFFまたはEPSイメージを含む、あるいはファイルが添付されたRTFDフォーマットの読み書き。
・別のオブジェクト（delegate）による属性などの動的制御。
・アプリケーション間を含むテキストのコピー＆ペースト。
・NSTextオブジェクト間のフォーマット情報を含むコピー＆ペースト。
・テキストに含まれる単語のスペル・チェック。

グラフィカルなユーザインタフェース構築ツール（Interface BuilderやまたはInterface Builder、あるいはInterface Builderとかですが）を使えば、テキストオブジェクトのいろいろなバリエーション、例えばNSTextField、NSForm、NSScollViewなどを、アプリケーションのユーザインタフェースとして利用することができることはご存知でしょう。これらのオブジェクトはそれぞれ特別な用途にあわせてデザインされたものですが、ついでに言えば、同じウィンドウの上に配置されたNSTextField、NSForm、NSButtonなど、セルを通して利用されるオブジェクト群は、メモリ節約のためフィールドエディタと呼ばれる同じテキスト・オブジェクトをシェアしています。したがって、アプリケーションが要求される機能が、これらのオブジェクトによって十分カバーされるのであれば、わざわざ新たにテキスト・オブジェクトを生成するよりもこれらを利用したほうがはるかにリーズナブルです。もちろんこれらのオブジェクトが要求されている機能を満たさない場合は仕方ありませんが。

テキスト・オブジェクトは普通いろいろな他のオブジェクトと密接に連携して機能します。それらのうちには…delegateや埋め込まれたグラフィック・オブジェクトなど、なんらかのプログラム・コードが要求されるものもあります。その他、フォントパネルやスペル・チェッカー、あるいはルーラーなどのように、それを使用可能とするか否か決める以外は何も気にする必要がないものもあります。

スクリーンや印刷された（される予定の、でしょうか）ページにおけるテキストのレイアウトを制御するには、その内容を表示するNSTextViewを格納するNSTextStorageにリンクするオブジェクト…具体的にはNSLayoutManagerとNSTextContainerを使います。

NSTextContainerオブジェクトは、テキストをレイアウトできる領域を定義します。デフォルトではこの領域は長方形となりますが、プログラマはNSTextContainerのサブクラスを定義することで、円や五角形など任意の図形を定義し、その形にテキストをレイアウトすることができます。なお、NSTextContainerはユーザインタフェース・オブジェクトではないので、何も表示することはできませんし、キーボードやマウスからイベントを受け取ることもできません。これは単に「テキストで満たすべき領域」について記述するクラスであり、その満たされるテキストを保持もしません。それはNSTextStorageの仕事です。

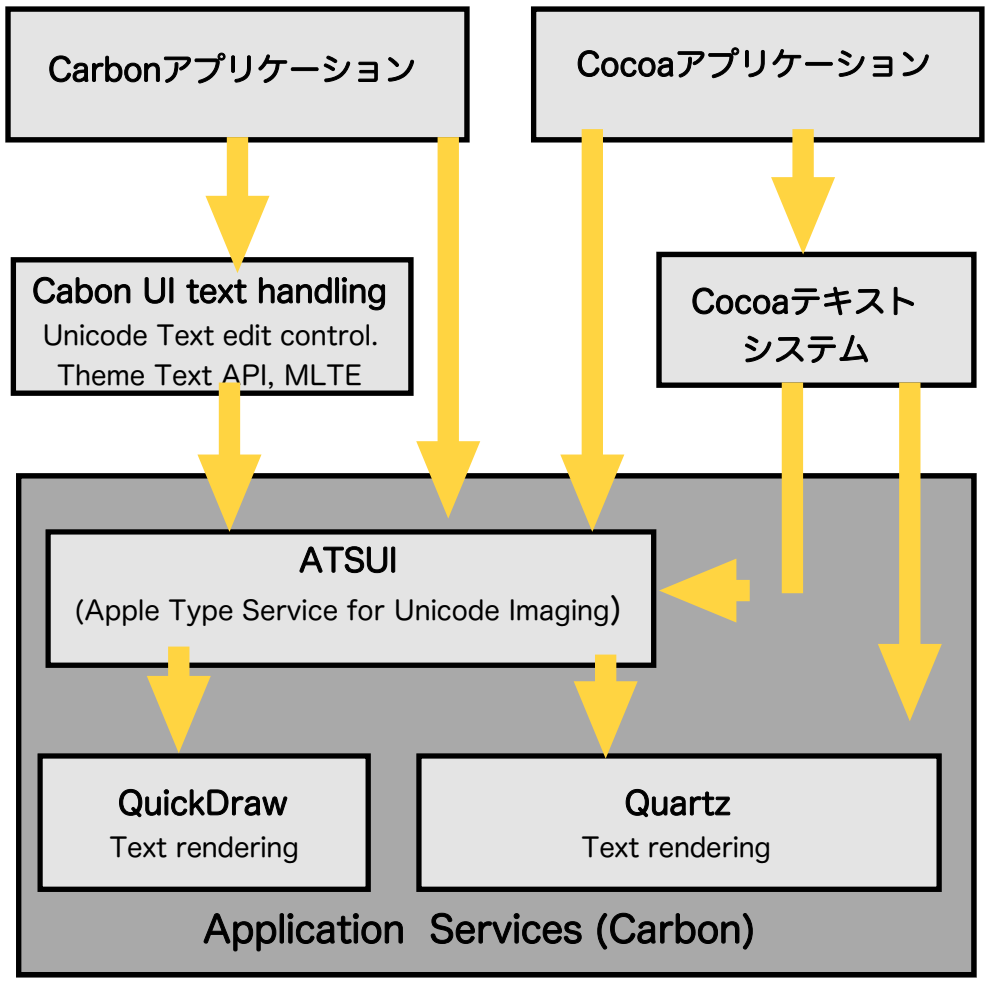
レイアウト・マネージャ・オブジェクト、NSLayputManagerは、他のテキスト・ハンドリング・オブジェクトのオペレーションを統合します。具体的にはNSTextStorageオブジェクトが保持するデータをNSTextViewで表示可能なレンダリングされたテキストに変換する処理および、そのテキストをNSTextContainerによって定義された領域に配置する処理を統括します。

CocoaテキストシステムとMLTEとATSUI

Cocoaテキストシステムは、Cocoaアプリケーションが必要とする全てのテキストサービズを提供するべく設計されたオブジェクト指向フレームワークです。他方、Carbonアプリケーションの開発には、テキスト編集機能を提供するMLTE（Multilingual Text Engine）のようなテキスト・オリエンティド・コンポーネント、そしてタイポグラフィとレイアウト・サービズを提供するATSUI（Apple Type Service for Unicode Imaging）のAPIを使用することになります。

デベロッパから見ると、CocoaテキストシステムとATSUIは共にテキストレンダリングのためにQuartzのCore Graphicライブラリをコールし、ライン＝レイアウトとキャラクター・グリフ変換を行う並列的なAPIです。Cocoaテキストシステムを利用して開発を行うデベロッパはATSUIを使わないでしょうし、逆もまた同様です。

以下は、Mac OS X開発環境におけるCocoaテキストシステム、ATSUI、MLTEその他のテキスト関連コンポーネントの相関を図示したものです。



Core Graphic レンダリングに関しての詳細はQuartz 2Dのドキュメントを参照してください。

Cocoaテキストシステムの印刷機能

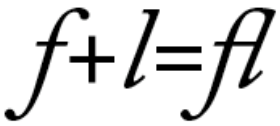
Cocoaテキストシステムは、Cocoaにおける全てのテキスト操作、および描画を統括し、Application Kitのクラスを通して高品質な印刷サービスを提供します。この項では、Cocoaテキストシステムにおける印刷処理の概念を解説します。

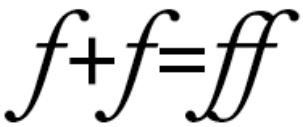
文字とグリフ

「文字」は書き言葉において「意味を持つ最小の単位」です。それらはローマ字におけるアルファベットのように話し言葉の特定の音（音節）に対応したり、中国語の表意文字のように単語そのものを一義に指し示したり、そして数学における各種の記号のように独立した概念を意味することもできます。が、いずれの場合にも「文字」自体は抽象概念に過ぎません。

抽象概念である「文字」が表示される場合、いつでも「それと認識可能な形」によって代替される必要があります。そしてそれらは形は同一とは限りません。すなわち「文字」とは、さまざまな形に描かれながら、同時にそれと認識され続けることができるものなのです。例えば我々は、異なったサイズ、異なった太さで大文字の「A」を描くことができます。直立させたり斜めに傾かせたり、場合によっては「セリフ」のような飾りをつけることも可能です。こうした「文字」の具体的な形のことを「グリフ」と呼びます。下の図は「A」という同じ文字の、いろいろなグリフの例です。

また、文字とグリフの対応関係は必ずしも一対一ではありません。例えば「e」と「’」の組み合わせである「é」のように、一つの文字が複数のグリフの組み合わせで表現されることも、また逆にリガチャー（連結文字）のように複数の文字を１つのグリフで表現することもあります。以下の図は、ある2つの文字が隣り合わせた場合にしばしば用いられるグリフの例です。また、単語の先頭や末尾にその文字が来るときのみ使われるグリフなども存在します。





コンピュータは文字をエンコーディングされた番号の羅列として管理しています。Mac OS Xで標準的に使われているエンコーディングはユニコードと呼ばれるもので、この規格は、使用するプラットフォーム、プログラム、そしてプログラム言語の違いにかかわらず、世界中の現代語の書き言葉に使用される全ての文字にユニークな番号を割り当て、異なる数百のエンコーディングが互いにコンフリクトするというコンピュータリゼーションにおける永年の懸案に解決するものです。ユニコードはまた、Cocoaが文字を保存するエンコーディングであり、テキストや書式の双方向的な操作を簡略化します。

グリフもまた、グリフコードと呼ばれる番号によって体系化されています。文字を描くグリフは、Cocoaのレイアウト・マネージャ（NSLayoutManager）によって選択されます。レイアウト・マネージャは、どのグリフをどのビューのどこに配置するかを決定し、文字・グリフ変換を効率化するため、グリフコードをキャッシュします。

タイプフェースとフォント

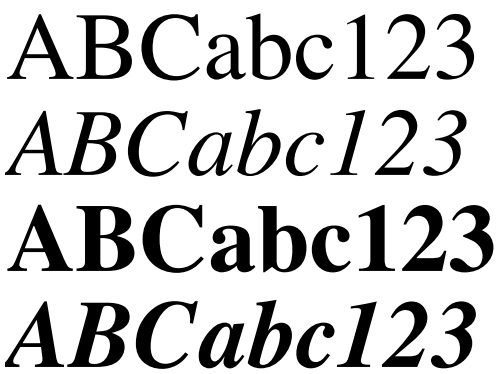
タイプフェースとは、書き言葉の全ての（あるいは一部の）文字に対して関係付けられた「見た目」のことです。例えば「Times」というタイプフェースは1931年、 スタンリー・モリソンによってロンドンのタイムズ新聞のためにデザインされました。Timesの全ての字形は、縦軸とカウンタ（文字の湾曲した部分）、そしてその他の部分の比率が等しく作られており、統一感のある外観をしています。このことは文字が紙面に並んだときに読みやすさとなって機能するのです。

タイプスタイル、単にスタイルとも呼びますが、は、タイプフェースの外見的特徴です。例えば「Roman」はセリフと横線より縦線が有意に太いことで特徴づけられますし、「Intalic」は右に傾きつつ丸みを帯びた、手書きを模したスタイルです。通常、一つのタイプフェースに対して複数のタイプスタイルが存在します。

フォントは、共通のサイズ、タイプフェース、そしてタイプスタイルに則ってデザインされたグリフの集合です。フォントは印刷や画面への表示など、使用環境に合わせて用意されており、またリガチャーなど一般的に使用される全てのグリフを含んでいます。

フォント・ファミリーとは同じタイプフェースに属し、タイプスタイルが異なるフォントのグループのことです。ですから、例えばタイプフェースの名前である「Times」はフォント・ファミリーの名前としても通用します。対して「Times Roman」や「Times Italic」となると、「Times」ファミリーに含まれる個々のフォントの名称になります。

以下の図は「Times」ファミリーに属する字体のサンプルです。



タイプフェースの特定のスタイルは独立した別のフォントとして用意されていることもありますが、そうしたフォントがない場合も少なくはなく、システムはそうした場合のために、既存のフォントからプログラムによって別のタイプスタイルを生成する機能をサポートしています。例えばそのファミリーの通常の字体の線を太くすることによって「Bold」スタイルを生成する、というようなことです。Cocoaテキストシステムでは必要に応じてこうして生成したフォントのバリエーションを使用できるようになっていて、その種類は「Bold」、「Italic」、「Condensed」、「Expanded」など多岐に渡ります。

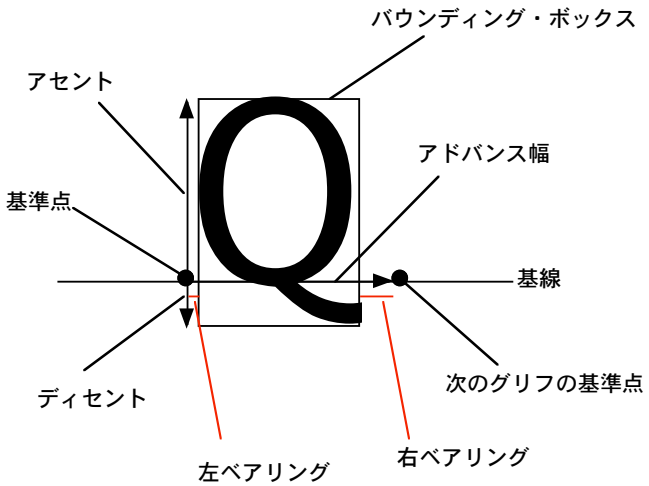
テキスト・レイアウト

テキスト・レイアウトとは、簡単に言えば表示装置にグリフを配置する過程になります。ここでいう表示装置とは、タイプセッターの1ページを模して構成されたテキスト・ビューと呼ばれる領域です。ここでグリフが展開される形式をテキスト・ディレクションといい、英語を始めとするラテン語起源の言語ではグリフはそれぞれ横方向に隣り合って配置され、単語同士の間には空白が挟み込まれます。単語はテキスト・ビューの左上から始まり右へ向かう「行」に沿って置かれて行き、行がビューの右端に達すると、一段下の左端に戻って新しい行を始めます。

グリフのレイアウト方法は言語によって大きく異なります。グリフを右から左に配置していく言語もあれば、横方向ではなく縦方向に配置していく言語さえあります。技術文書などを書く場合には、異なるテキスト・ディレクションを持つ言語…例えば英語とヘブライ語など、が同じ行の中で混在するのはよくあることです。行ごとにテキスト・ディレクションが入れ替わる場合もあります。いくつかの言語では単語ごとにグリフに空白を挟むことをしませんし、グリフの恣意的な配置を実現するあるアプリケーションにいたっては、グリフを非線形にすることさえ要求されます。

Cocoaのレイアウト・マネージャ（具体的にはNSLayoutManagerのインスタンス）は、基線（baseline）と呼ばれる見えない線に沿ってグリフを配置していきます。英語に代表されるローマン・テキストでは、基線は水平であり、大部分のグリフのボトムエッジに接します。なかには「g」や「q」のように「シッポ」が基線の下に跳び出すものや、大文字の「O」のように光学的理由からその曲線がわずかに基線より下まで出ているものもありますが…。ローマン以外のシステムでは、グリフを基線の下に並べるもの、基線を中心として配置するものなどがあります。そして全てのグリフは、基線に対して正しい位置に配置されるための基順点（origin point）を含んでいます。

グリフのデザイナーは、フォントと共に1セットの測定値…メトリックと呼ばれるフォントに属する個々のグリフの周囲のスペースに関する数値を提供します。レイアウト・マネージャはこの値を参照しながらグリフの配置を決定するわけです。横書きテキストの場合、グリフはアドバンス幅と呼ばれる値を持ち、この値が基線に沿った次のグリフの基準点への距離を表しています。グリフの基準点はその左の端からなにがしかの間隔を開けて設定されているのが普通で、これを左サイドベアリング（あるいは左ベアリング）と呼びます。また、アドバンス幅がグリフの右端を超えていることがあり、この右端からアドバンス幅の到達点までの間隔を右サイトベアリング（あるいは右ベアリング）と呼んでいます。また、グリフにおける垂直方向の寸法はアセントとディセントと呼ばれる2つのメトリックによって示されます。アセントは基線からグリフの上端までの距離、ディセントは同じく下端までの距離です。最後に、グリフの見える部分を全て囲い込む矩形を、外枠（バウンディング・レクタングル、あるいはバウンディング・ボックス）と呼びます。下の図はこれらの値を図示したものです。



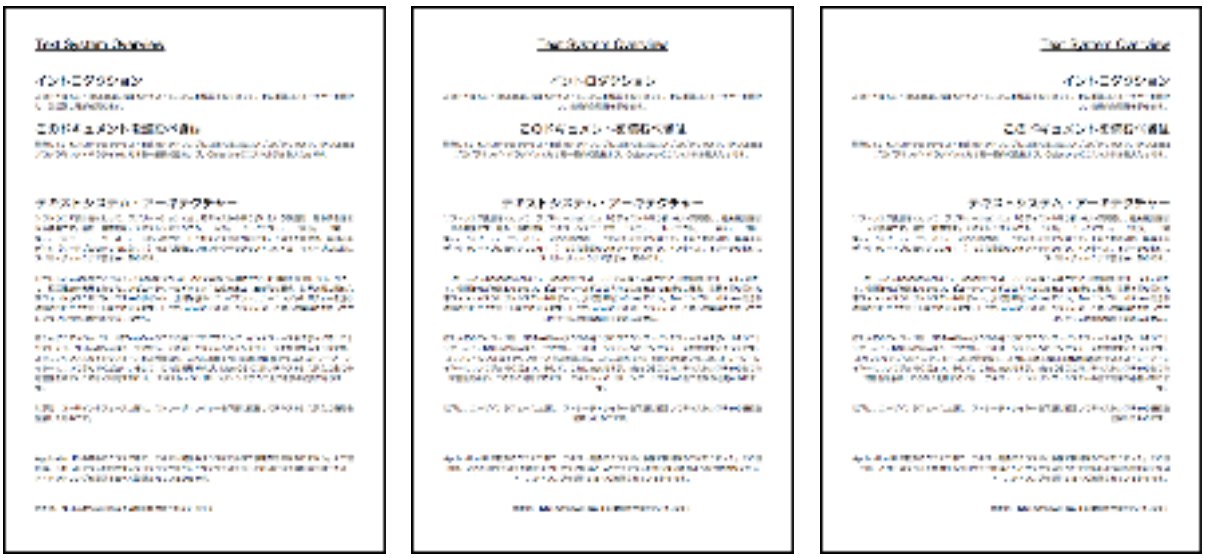
通常、タイプセッターはアドバンス幅を標準的なグリフ間隔として使用します。しかし、テキストの組み合わせによっては、その間隔を狭めたり広げたりしたほうが読みやすくなる場合があります。この調整をカーニングといい、下の図の大文字の「W」と「A」の間にその効果的な例が見られます（上がカーニングの実行例）。デザイナーは、このカーニング情報もメトリックの一つとしてフォントに付与します。Cocoaテキストシステムには、このカーニング情報の使用・不使用、フォントに付与された値を使うか、あるいはそれをもっときつくするか、ゆるめるか、を指定できるきめ細かなAPIが用意されています。

タイプシステムは通常フォントのメトリックを「ポイント」という単位で扱います。1ポイントは1/72インチ。アセントとディセントのポイント数の和がそのフォントのポイントサイズとなります。

タイプセッターにおいて行と行の間に開けられる間隔をリーディング、あるいはラインギャップ（行間）といいます。グリフのアセント、ディセントにこのリーディングを加えたものがライン幅（行幅）です。

タイプデザインにおける印刷概念のいくつかは少々難解ですが、コンピュータやタイプライターを使ってドキュメントを作ったことのある人であれば、1枚の紙の上にテキストをレイアウトする際に使われるそれであればなじみがあるでしょう。例えば「マージン」はページの縁の余白のこと、「アラインメント」はテキストをマージンに沿って配置する方法です。例えば次の図に示すように横書きのテキストは左揃え、中央揃え、右揃えのアラインメントが可能です。

左揃え　　中央揃え　　右揃え



また、行幅を整えることも可能です。次の図では横書きテキストを左右のマージンで整列させています。

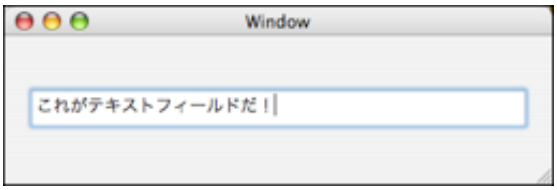
一連のグリフを複数の行に配置するために、レイアウト・エンジンは文字列の中から改行可能な位置を見つけ、そこで改行を行います。Cocoaテキストシステムでは、プログラマは単語と単語の間で改行を行うか、それろもグリフの境界で行うかを指定することがでする必要があります。なお、改行が発生すると、システムは必要に応じて行の整列を行います。

テキストフィールド、テキストビュー、フィールドエディタ

テキストフィールド、テキストビュー、そしてフィールドエディタはCocoaテキストシステムの重要なオブジェクト、これらのオブジェクトはシステムとユーザを仲介するインタフェースの中枢です。これらのオブジェクトはテキスト入力、操作、表示機能を提供します。これらに対する理解なくしてアプリケーションはユーザにテキスト処理を提供できません。

テキストフィールド

テキストフィールドはNSTextFieldクラスのインスタンスで、少量のテキスト、通常は（もちろん例外もありますが）1行だけのテキストを表示したり、ユーザによるテキスト入力を行うコントロール・オブジェクトです。他の全てのコントロールと同様、テキストフィールドにもtargetとactionが設定でき、デフォルトではユーザがテキストの編集を終えたとき……というのはユーザがリターンキーを押すか、他のコントロールにフォーカスを移したときですが、にtargetに向けてactionメッセージを送ります。プログラマはテキストフィールドの形、レイアウト、フォントや表示色、背景色を指定でき、また編集の可否や、編集不可の場合に選択だけは可能にするかなどを設定できます。

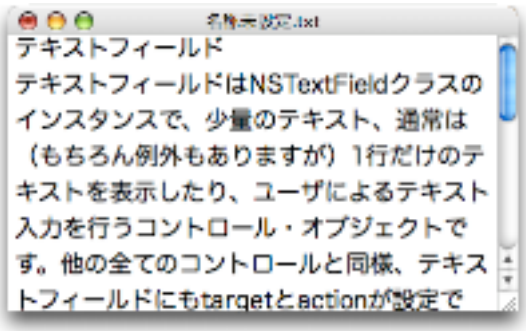


また、NSTextFieldのサブクラスであるNSSecureTextFieldを使えば、パスワート入力などのために入力した文字がエコーバックしないテキストフィールドを提供することができます。このフィールドはユーザの入力した文字をエコーバックする代わりに「●」を表示し、編集メニューによるカットやコピーも受け付けません。プログラマはこのオブジェクトにsetringValueメッセージを送って入力された文字列を得ることができます。

テキストフィールドを生成する最も簡単な方法はInterface Bulderを使ってCocoa-Viewsパレットからテキストフィールドをドラッグし、ウインドウの上でドロップすることです。もしこのフィールドをNSSecureTextFieldのインスタンスにしたいなら、フィールドを選択してインフォ・ウインドウを開き（command + shift + I）、カスタムクラス・ペーン（command + 5）からNSSecureTextFieldを選びます。

テキストビュー

テキストビューはNSTextViewクラスからインスタンシエイトされ、複数行にわたるテキストを表示することができる、Cocoaテキスト編集システムの主要インタフェースです。

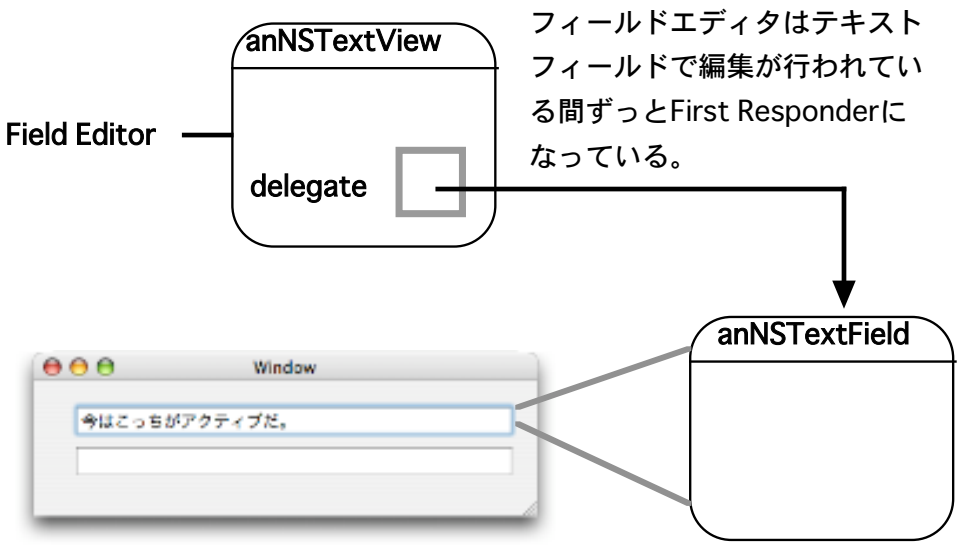


テキストの入力、変更に関わるユーザ・イベントを処理し、英語以外の言語を含むあらゆるフォントを、任意の色、スタイル、その他の属性で表示します。

Cocoaテキストシステムはテキストビューその他の基本的なオブジェクトを使って、テキストストレージ、レイアウト、フォント、属性操作、スペル・チェッキング、アンドゥとリドゥ、コピーとペースト、ドラッグ＆ドロップ、テキストのファイルへの保存などの機能をサポートします。NSTextViewはNSTextのサブクラスですが、この2つのクラスは分けて定義されているのは歴史的理由によるもので、NSTextViewではなくNSTextクラスをインスタンシエイトする意味はありません。プログラマはNSTextViewオブジェクトをウインドウの上に配置するだけで、Cocoaテキストシステムが提供する全てのサービスをフル装備したテキストエディタを作ることができます。後で「15分でテキストエディタが作れちゃう」という説明をします。お楽しみに。

フィールドエディタ

フィールドエディタはウインドウ内の全てのテキストフィールドで共有される単一のNSTextViewオブジェクトです。このテキストビューオブジェクトは、アクティブなテキストフィールドに対するテキスト編集機能をサポートするために、自身をビューの階層構造の中に割り込ませます。そうしてユーザがあるテキストフィールドにフォーカスを移すと、そのフィールドに対するキー入力イベントを処理し、テキストを表示し始めます。また、フィールドエディタはアクティブなテキストフィールドを自身のdelegateに指定し、テキストフィールドオブジェクトがその内容を変更できるようにします。そしてフォーカスが他のテキストフィールドに移動すると、フィールドエディタは自身をそのフィールドに割り当て直すわけです。以下の図はこのフィールドエディタとテキストフィールドの関係を示したものです。



一つのウインドウの中で一度にアクティブでありうるテキストフィールドは一つだけですから、システムは各ウインドウに一つずつフィールドエディタのインスタンスを用意すればいいわけです。もう一つ、テキストの選択を維持することもフィールドエディタの重要な仕事です。したがって、編集状態にない（アクティブでない）テキストフィールドは通常「何も選択されていない」状態にあります（もちろんプログラマにはそれが可能なエディタを作って置き換えることもできるわけですが）。

テキストシステムとMVC

Cocoaテキストシステムは、その使い勝手と柔軟性のためにモジュール化され、また多層構造を持つよう設計されています。そのモジュール化デザインは、Smalltalk-80の流れを汲むモデル＝ビュー＝コントローラーパラダイムを反映し、データとその視覚表現、ロジックを別々のオブジェクトとして定義します。テキストシステムの場合には、NSTextStorageがモデルのテキストデータを保持し、NSTextContainerがレイアウト領域を設定、NSTextViewがビュー（視覚表現）を受け持ち、NSLayoutManagerがコントローラーとしてこれらのオブジェクトの働きを制御しています。

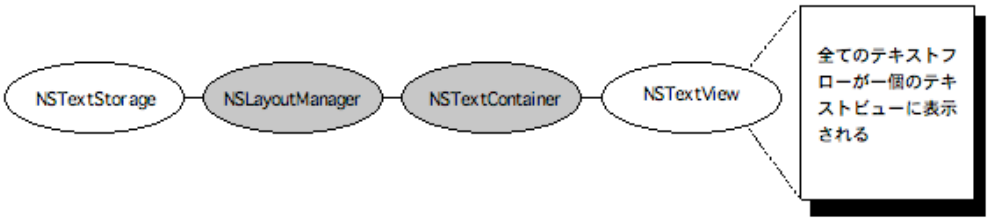
このようにオブジェクトごとの責任分担を明確にし、コンポーネント相互の依存関係をできるだけ少なくすることで、システム全体の設計をいじらずにコンポーネント単位での改良や仕様変更をすることが可能になります。テキスト・ハンドリングにおけるコンポーネント独立の効用は、それぞれシステムのある一部だけを使って実現される以下の操作をみてもおわかりになると思います。

・文字あるいは文字列、パラグラフスタイルなどの検索はNSTextStorageオブジェクトだけで可能です。
・テキストに対するプログラムからの加工もNSTextStorageオブジェクトだけで可能なので、処理に際して表示・レイアウトに関わるオーバーヘッドを被らずに済みます。
・テキストが実際にレイアウトされるときにどこで改行が起きるかとか、全部で何ページになるのかなどの計算も全てNSTextViewを使わずに（それ以外のオブジェクトで）行うことができます。

また、テキストシステムの多層構造は一般的なテキスト・ハンドリング処理を実現するにあたってのプログラマの負担（学習）を軽減します。実際、多くのアプリケーションがNSTextViewクラスのAPIだけを使ってそれらの処理を実現しているのです。

一般的な構成

以下のいくつかのダイアグラムは、それぞれ異なったテキスト・ハンドリングを実現するために、テキストシステムの基本的な4つのオブジェクト、NSTextStorage、NSLayoutManager、NSTextContainer、NSTextViewをどう構成するかを示したものです。
まずは最も単純なもの、以下の図は一つのテキストフローを表示する場合です：

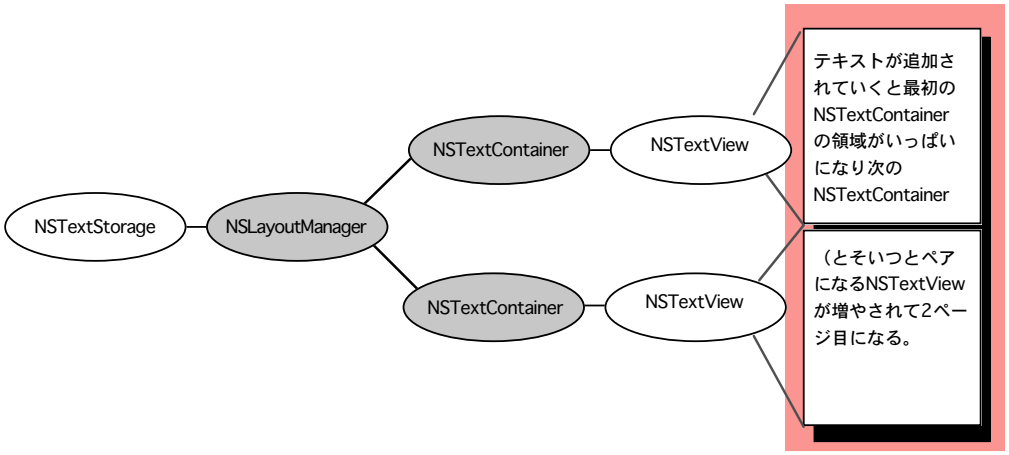


NSTextViewオブジェクトはグリフを表示するビューを提供します。そしてNSContainerはそのビューの中でグリフがレイアウトされる領域を定義します。上のような構成では、通常、NSContainerの縦方向の値は、どんな量のテキストにも対応できるように非常に大きく設定されています。そしてそのNSTextViewをNSScrollViewのサブビューにすることで、ユーザはその多量のテキストをスクロールして見ることができるわけです。

NSTextContainerが定義する領域が、NSTextViewの外枠を設定したものであれば、テキストの周囲にはマージンが設けられています。NSLayoutManagerは、その他ここでは言及しないいくつかのオブジェクトと協力し、NSTextStorageが保有するデータをグリフに変換し、それをNSTextContainerの定義する領域に配置します。

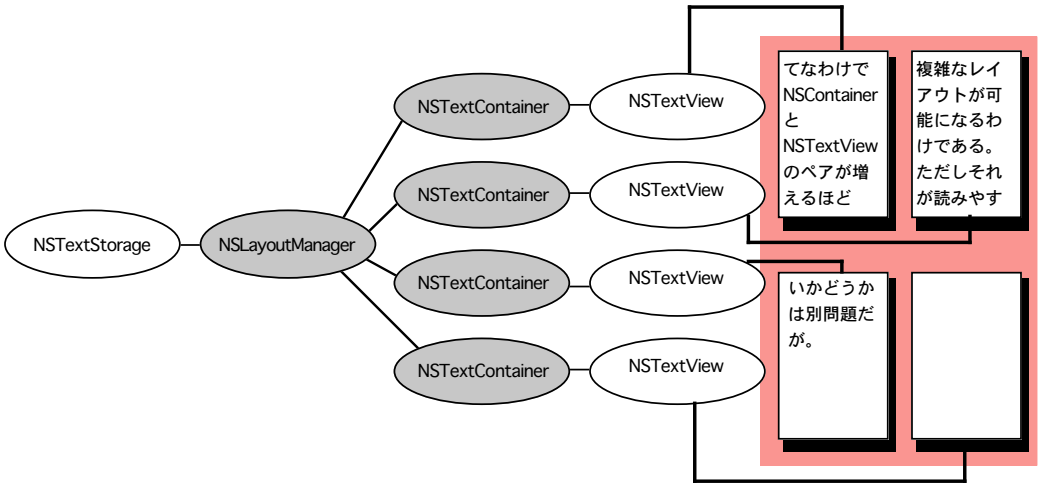
この構成にはNSTextCntainerとNSTextViewのペアが1対しかないので、テキストはNSContainerが定義する領域の中に連続していなければなりません。つまり、下位ページやマルチコラム・レイアウト、その他より複雑なテキストレイアウトを実現することはできません。

それら複雑なレイアウトを実現するためには複数のNSTextContainer-NSTextViewペアを使います。以下の図は改ページをサポートできる構成の例です。



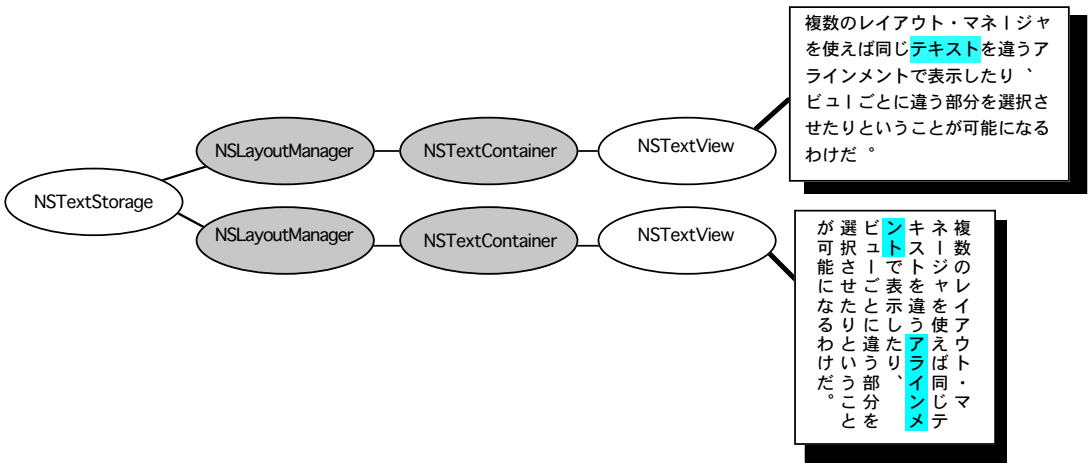
この図で2つあるNSTextContainer-NSTextViewペアは、ぞれぞれドキュメントの1ページに対応しています。赤い色で示した領域はこのアプリケーションがNSTextViewのバックグラウンドに用意するカスタムビュー・オブジェクトを表しています。ユーザが複数のページを持つドキュメントを連続してスクロールできるように、このカスタムビューをNSScrollViewのサブビューにするわけです。

同様に、マルチコラム・レイアウトの場合は以下のようになります。



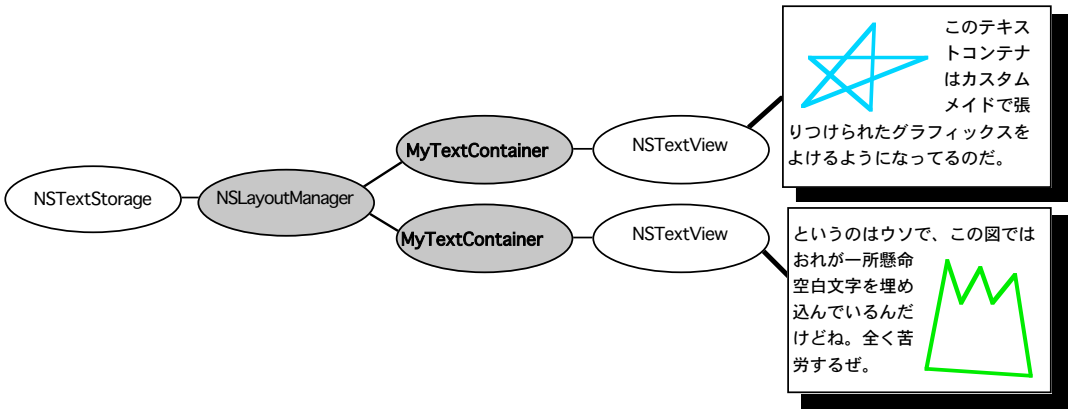
この場合、2組のNSTextContainer-NSTextViewペアに1ページを対応させて、その各ペアはそれぞれドキュメントの一部をコントロールしています。テキストを表示するとき、グリフはビューの左上からレイアウトされていきます。次のグリフを表示する余地がなくなると、NSLayoutManagerはその旨をdelegateに通知します。delegateはまだ表示されていないテキストが残っているかどうか確認し、必要があればNSTextContainer-NSTextViewペアを追加します。追加されたペアの途中で全てのテキストを表示し終えると、NSLayoutManagerはそれもdelegateに知らせるわけです。この例で赤く示された領域はテキストコラムのためのキャンバスに当たります。

NSTextContainer-NSTextViewペアではなく、複数のNSLayoutManagerを同じNSTextStorageにアクセスさせることも可能です。以下の図は複数のレイアウト・マネージャを使った最も簡単な例です。



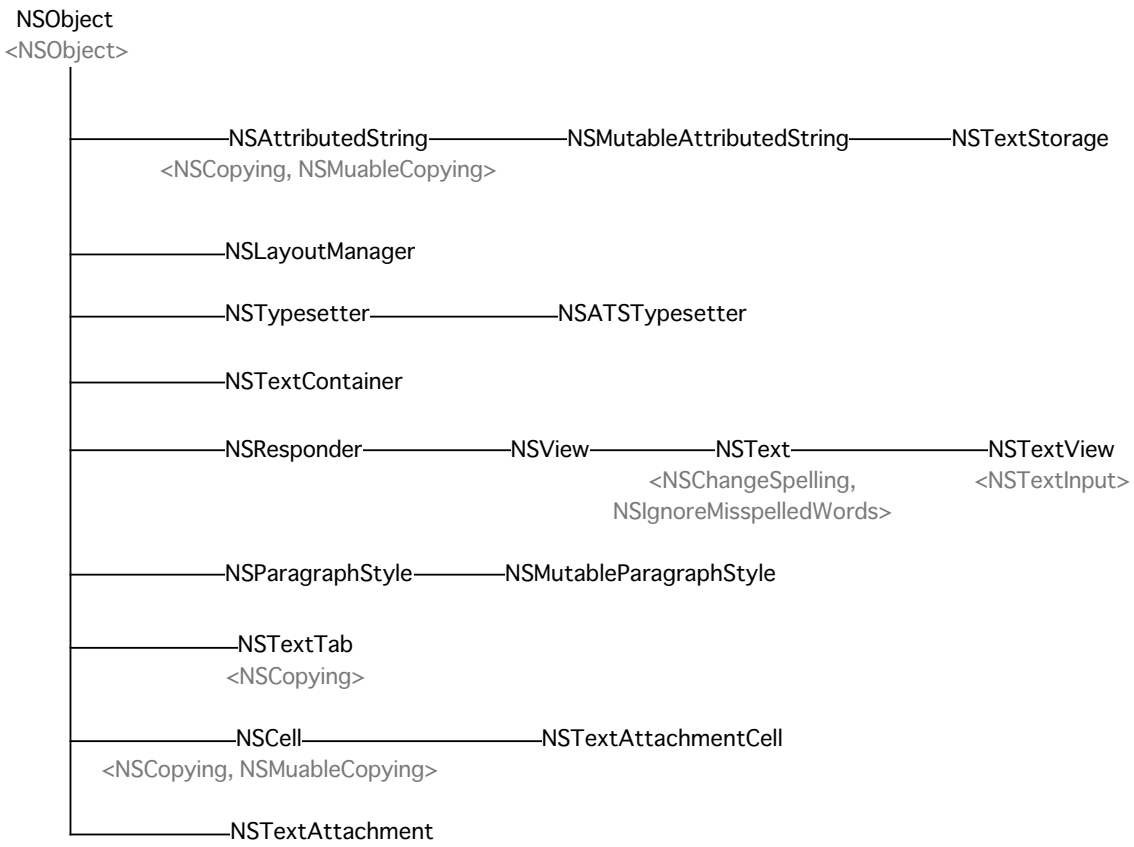
この構成を使うと同じテキストを違うビューで表示できます。トップビューでユーザがテキストに変更を加えれれば（そしてその部分がもうひとつのビューでも見える位置にあれば、ですが）、その変更は即座に片影されます。

最後に、NSTextContainerのサブクラスを使用して、テキストが埋め込まれたグラフィックスを取り囲むような複雑なレイアウトを実現する例です。このサブクラスは埋め込まれたグラフィックスの形状に対応してレイアウト領域の形を変更するわけです。


NSTextStorage
NSLayoutManager
MyTextContainer
MyTextContainer
NSTextView
NSTextView
このテキストコンテナはカスタムメイドで張りつけられたグラフィックスをよけるようになってるのだ。
というのはウソで、この図ではおれが一所懸命空白文字を埋め込んでいるんだけどね。全く苦労するぜ。

Cocoaテキストシステムのクラス・ヒエラルキー

Cocoaテキストシステムには、既に説明した4つの基本的なクラス（NSTextStorage、NSLayoutManager、NSTextContainer、NSTextView）に加え、多くの補助クラス、そしてプロトコルが用意されています。以下のダイアグラムはその一覧です。<NSCopying>のように<>で囲まれているものはプロトコルを表します。



その他の関連クラス

- NSFileWrapper
- NSInputManager
- NSInputServer
- NSFont
- NSFontPanel
- NSFontManager
- NSFontDescriptor
- NSGlyphGenerator
- NSGlyphInfo
- NSGlyphStorage protocol
- NSRulerView
- NSRulerMaker
- NSTextField
- NSSecureTextField
- NSSpellChecker

15分でテキストエディタが作れちゃう

お待ちかね。XcodeとInterface Builderを使って簡単に、でも非常に高機能なテキストエディタを作成する方法です。Cocoaのドキュメント・アーキテクチャには、この種の仕事を行う場合に必要となる多くの機能が揃っており、プログラマが書かなければならないコードの量をほんのわずかにしてくれます。

テキストエディタは以下の手順で作ります：

＊Xcodeを使って新規にドキュメントベースのアプリケーションを作ります。
＊Interface Builderを使ってアプリケーションのウインドウにNSTextViewオブジェクトを追加します。
＊ドキュメント・コントローラのクラスにちょっとだけコードを追加します。
＊ユーザ・インタフェースをそのコードに連結します。

いくつかの段階で実際のアプリケーションをビルドし機能を確認することができます。

それでは以下、上の手順を細かく見ていきましょう。これからの解説にあたっては、XcodeとInterface Builderについて基本的な知識があることが前提となります。

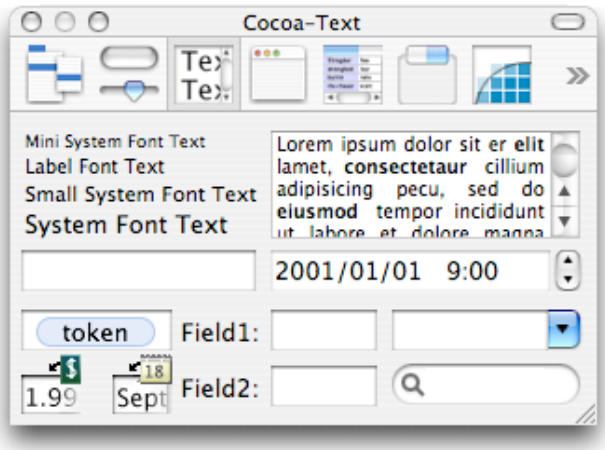
ユーザー・インタフェースを作る

まずは、Xcodeでプロジェクトを作り、InterfaceBuilderを使ってユーザ・インタフェースを組み立てます。

1. Xcodeを起動してファイルメニューから「新規プロジェクト...」を選択、現れるウインドウから「Cocoa Document-based Application」を選びます。

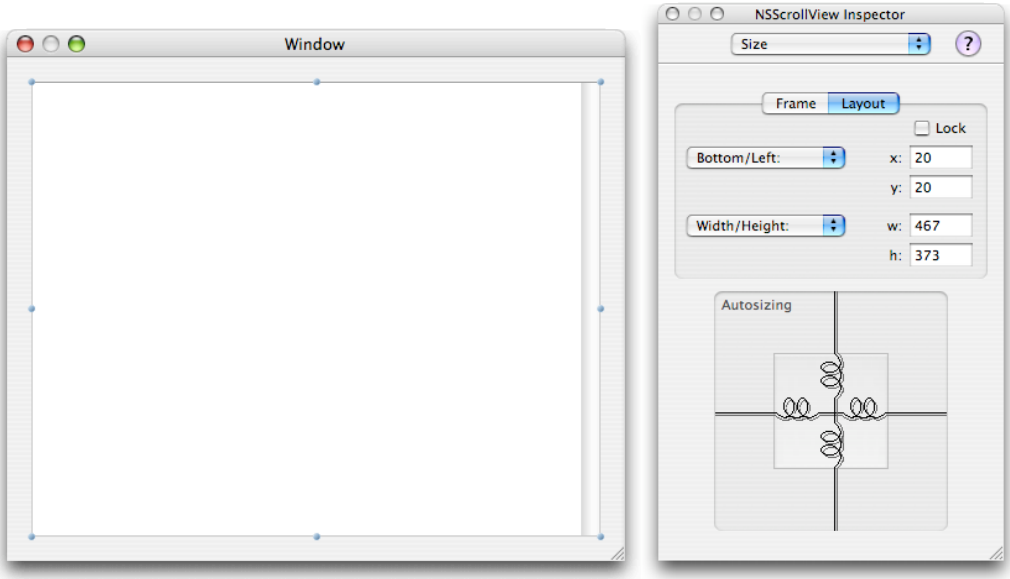
2. Interface Builderを使って、「Resources」フォルダの中の「MyDocument.nib」というファイルを開きます（ダブルクリックすればInterface Builderが起動します）。

3. ウインドウに貼り付けてある「Your document contents here」のラベルを削除し、Cocoa-Textパレット（下の図）からここにNSTextViewをドラッグしてきて貼り付けます。Interface Buiderが表示するガイドラインに沿って、ウインドウのほとんどを覆うようにこれをリサイズします。



4. テキストビューを一度クリックして、Toolsメニューから「Show Inspector」を選びます。ポップアップメニューから「Size」を選び、以下の図のようにしてウインドウとビューのサイズの相関を設定します。

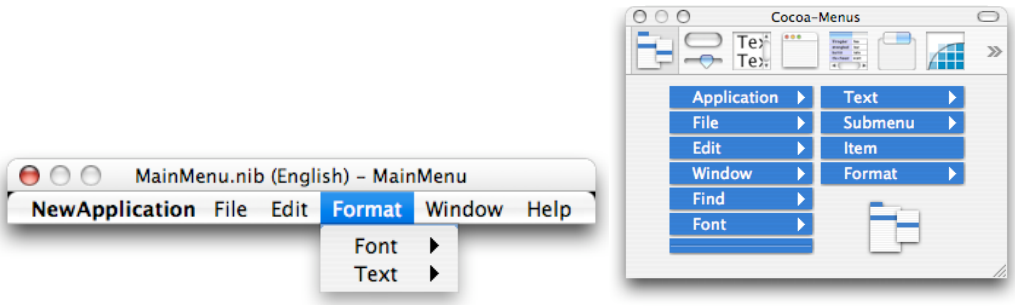
注意：ここでテキストビューを一度クリックするのは貼り付けたNSTextViewを内包するNSScrollViewを選択するためです。ウインドウのリサイズに合わせてリサイズされるのはこのNSScrollViewだからです。NSTextViewそのものを選択したいときにはダブルクリックします。



5. インスペクタ・ウインドウのポップアップメニューから今度は「Attributes」を選択し、以下のオプションがオンになっているのを確認します。「Editable」、「Multiple fonts allowed」、「Undo allowed」、「Continuous Spell Checking」、「Use Find Panel」、そして「Show Scroller」。

6. Interface BuilderのFileメニューから「Test Interface」を選んで、ウインドウのリサイズが正しく動作することを確認します。ついでにもうちゃんと文字を入力できることにちょっと驚いてもらえれば完璧です。ウインドウにNSTextViewのオブジェクトをドラッグして来るだけで、Interface Builderはそこにこれまで説明した機能を全て備えた完全なCocoaテキストシステムを作り上げてしまうのです。

7. アプリケーションにFormatメニューを追加します。XcodeでMainMenu.nibのアイコンをダブルクリックして開き、Cocoa-MenusパレットからFormatメニューをドラッグしてきて以下の図のように配置します。



8. 2つのNibファイルを保存してXcodeに戻り、アプリケーションをビルドしてテストしましょう。

ここまでで、開発中のこのエディタは既に多くの洗練された機能を備えています。ユーザはテキストの入力と編集、カット、コピー、ペーストができます。Findウインドウを使ってテキストのなかから特定の文字を探すことや、それを置き換えることも可能です。アンドゥ、リドゥもサポートされていますし、好きなフォント、サイズ、スタイル、色を使うことができます。

もちろん右寄せ左寄せなども指定できますし、カーニングやリガチャーも設定可能です。ルーラを表示してタブ位置を設定することもできますし、……これは日本人にはあんまり関係ありませんが、スペルチェッカーも搭載されているんです。
それら充実した編集機能に加えて、このエディタは既に複数のドキュメントを同時に開くことができます。今このエディタに欠けている最大の機能はそうして編集したテキストをファイルに保存したり、既存のファイルから保存されているテキストを読み込む能力です。それから、プログラムの名刺ともいうべきAboutウインドウを表示する機能もですね。
ではいったんアプリケーションを終了して読み込み、保存機能の実装に入りましょう。

ドキュメントを読み込んだり保存したりする機能を実装する

ここでは開発中のエディタに編集したドキュメントをファイルに保存する機能を追加しましょう。

1. まず、ドキュメントの読み込み、保存を司るNSDocumentのサブクラスに、編集対象のNSTextViewとの連結のため、それから編集された文字列を保持するためのインスタンス変数を追加します。以下の変数宣言をMyDocument.hに追加しましょう。

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

2. 上のインスタンス変数のうち、文字列の方は初期化しておく必要があります。MyDocument.mのinitメソッドに以下のステートメントを加えます。

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

3. この変数のためのゲッター、セッターメソッドを用意します。MyDocument.mに以下を追加します。

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

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

4. ゲッター、セッターのためのメソッド宣言をMyDocument,hに追加します。MyDocument.hはこれで以下のようになったはずです。

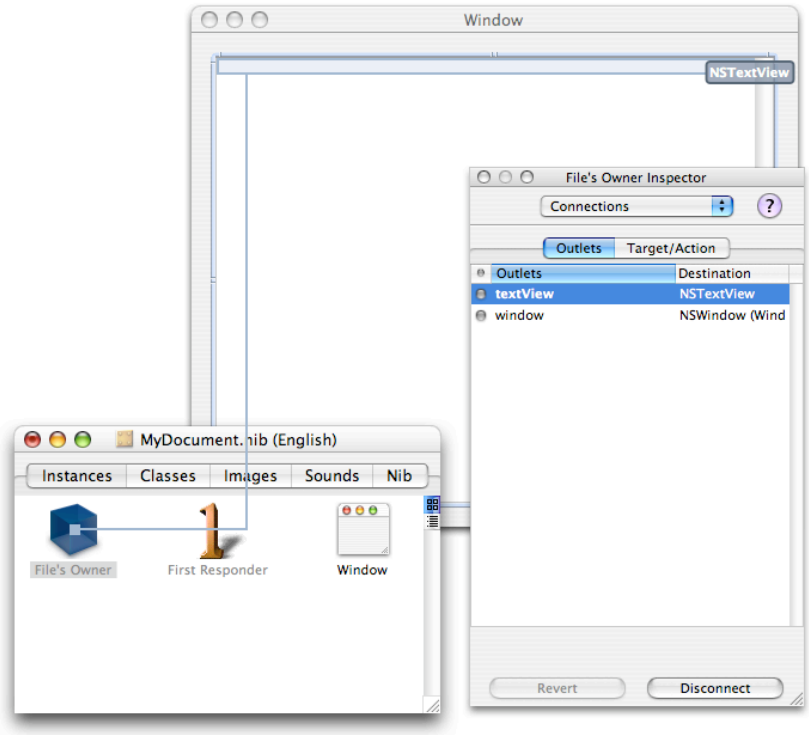
```
#import <Cocoa/Cocoa.h>
```

```
@interface MyDocument: NSDocument
{
	IBOutlet NSTextView *textView;
	NSAttributedString *mString;
}
- (NSAttributedString *) string;
- (void) setString: (NSAttributedString *) value;
@end
```

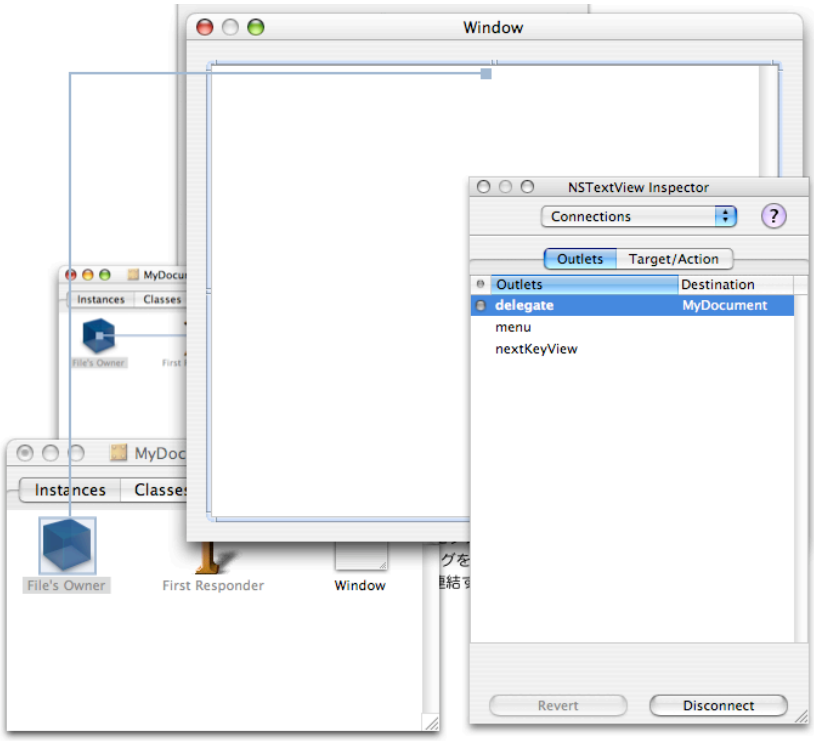
5. XcodeからInterface Buiderで開いたMyDocument.nibのInstanceペーンにMyDocument.hをドラッグします。こうすることで、MyDocumentのインタフェースにはtextViewというアウトレット変数があることをMyDocument.nibに通知するわけです。

6. Interface BiulderでNSTextViewをダブルクリックして選択します。一回だとNSScrollViewが選択されてしまいますのでよく確認してください。

7. MyDocument.nibのInstanceペーンにあるFile Ownerアイコンから、先ほど選択したウインドウのNSTextViewへ、コントロール+ドラッグして図のように連結させます。File Ownerのインスペクタ・ウインドウを開いて連結を確認してください。



8. interface Builderを使い、NSTextViewインスタンスのdelegateにFile Owner（これがつまりはMyDocumentオブジェクトです）を指定します。ビューをダブルクリックして選択し、先ほどとは逆にビューからFile Ownerのアイコンにコントロール+ドラッグを行います。以下の図のように、NSTextViewのインスペクタ・ウインドウでdelegateアウトレットに連結すればOKです。



9. ユーザによる変更に対して、テキストビューの中のテキストストレージとドキュメントのデータモデルであるインスタンス変数 mStringの同期を確保するため、MyDocument.mにdelegeteメソッドtextDidChangeを用意します。

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

10. 読み込みと保存のメソッドを実装します。プロジェクトタイプに「Cocoa Document-based Application」を選択しているのでXcodeはこれらのひな形を用意しています。それらを以下の用に変更します。

```
- (NSData *)dataRepresentationOfType:(NSString *)aType
{
	NSData *data;
	[self setString:[textView textStorage]];
	data = [NSArchiver archivedDataWithRootObject:[self string]];
	return data;
}

- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType
{
	NSAttributedString *tempString = [NSUnarchiver unarchiveObjectWithData:
	data];
	[self setString:tempString];
	return YES;
}
```

Mac OS X 10.4以降のシステムをターゲットにしたアプリケーションであれば、これらの代わりにdataOfType:error: とreadFromData:ofType:error: をオーバーライドすることになります。

11. ウインドウのNibファイルが最初にロードされたときに、テキストビューにドキュメントのデータモデルの内容を表示するためにwindowControllerDidLoadNibメソッドに次のコードを追加します。スーパークラスのwindowControllerDidLoadNib呼び出しをそのまま残し、その後に以下のコードを続けてください。

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

12. アプリケーションをビルドしてテストを行いましょう。

開発中のエディタはいまや、ユーザがテキストビューの上で入力し編集したテキストをファイルに保存でき、過去に保存したそれを読み込んでまた編集することも可能になりました。しかもユーザが変更したドキュメントを保存せずにウインドウを閉じたりアプリケーションを終了しようとすると保存を促す警告まで出します。ここまで15分、もっとかかってしまったでしょうか？

ただ、いまのところ、このエディタが開いたり保存したりできるドキュメントは拡張子が「????」という変なファイルだけです。しかもこのファイルをユーザがダブルクリックしてもこのエディタが起動されるわけではありません。ちゃんとしたファイルタイプで読み込みや保存が行えるようにするためには、Xcodeでアプリケーションのドキュメント・タイプを設定しなければなりません。

Cocoaアプリケーションのサンプルコードは以下のURLからダウンロードすることができます。

http://developer.apple.com/samplecode/Cocoa/idxTextFonts-date.html

また、OSに付属しているテキストエディットのソースがXcodeと一緒にインストールされる以下のディレクトリにありますので参照してください。

/Developer/Examples/AppKit/

簡単なテキスト処理

ここでは、いくつかの簡単なテキスト処理をCocoaテキストシステムを使って行うちょっとしたプログラミング・テクニックについて解説します。

テキストをビューに追加する

まずはテキストビューに新たにテキストを追加する方法です。同時にその追加したテキストがユーザに見えるようスクロールすることも必要です。以下のコードではまず、myViewというテキストビューに属するテキストストレージが現在保持しているテキストの最後の位置から始まる長さゼロの範囲を定義します。次にその範囲を追加したいテキスト、myTextで置き換えます。最後に、置き換えに使った範囲、endRangeの長さを追加したテキストの長さで置き換え、この範囲までスクロールするようテキストビューのscrollRangeToVisible: メソッドをコールします。

```
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];
```

フォントスタイルやトレイトを指定する

次は太字や斜体といったフォントスタイルや、アンダーラインなどの属性を指定する方法です。例えばアンダーラインはCocoa Foundationリファレンスに説明されているように、NSUnderlineStyleAttributeNameという定数を使って属性付き文字列（NSMutableAttributedStringかまたはそのサブクラスのインスタンス）に簡単に設定できる属性です。以下のメソッドを使います。

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

nameに定数NSUnderlineStyleAttributeNameをvalueに[NSNumber numberWithInt:1]を指定します。太字や斜体のようなフォントスタイルの指定はもうちょっとだけ複雑です。フォントマネージャのインスタンスをフォントを変換し、その属性を文字列に設定しなければなりません。以下のコードを参照してください。attrributedStringというのが属性をセットしたい文字列です。

```
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);
       font = [fontManager convertFont:font toHaveTrait:NSBoldFontMask];
       [attributedString addAttribute:NSFontAttributeName value:font range:fontRange];
       idx = NSMaxRange(fontRange);
}
```

NSTextStorageオブジェクトに対してこの種の設定を行う場合には、上のコードをbeginEditingとendEditingの間に置きます。

グリフのビュー座標を獲得する

グリフの位置は、それがレイアウトされている行の外枠（バウンディング・レクタングル）の基点との相対で計算されます。特定のグリフの外枠を獲得するにはNSLayoutManagerのメソッド、

`lineFragmentRectForGlyphAtIndex:effectiveRange:`

を使います。そしてこの矩形の基点座標に、同じくNSLayoutManagerの

`locationForGlyphAtIndex:`

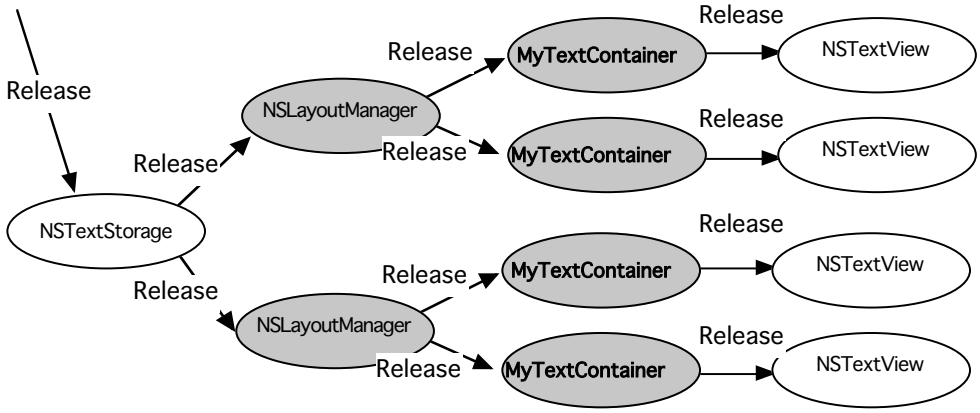
が返す、グリフの位置を加算します。

以下のコードはこのテクニックのサンプルです。

```
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;
```

テキストシステムを手作りする

大多数のアプリケーションはNSTextViewのハイレベルなAPIだけを使ってテキストシステムと情報をやりとりします。が、NSTextStorageオブジェクトから始めて、ボトムアップでテキストシステムのオブジェクト・ネットワークを構築することも可能です。実際にそんなことをしなくても、 その道筋を理解しておけば、 テキストシステムの設計への理解を深めることができるでしょう。
テキスト処理のネットワークを構築する際には4つのオブジェクトを作りますが、それらのうち３つはネットワークに追加したら解放してしまいます。つまりそこには、それらを順繰りにretainしているトップのNSTextSorageオブジェクトへの参照だけが残されるわけです。ただし、NSTextViewはNSTextContainerだけでなくそのスーパービューによってもretainされていますから、NSTextStorageを解放する前にNSTextViewにremoveFromSuperViewメッセージを送ってこのretainを外しておかなければなりません。
どんなに複雑なシステムであっても、概念的にテキスト処理オブジェクトネットワークの「所有者」はNSTextStorageオブジェクトです。プログラマがNSTextStorageオブジェクトを解放すれば、それはNSLayoutManagerを解放し、NSLayoutManagerはNSTextContainerを、NSTextCntainerはNSTextViewを解放することになります。すなわちメモリの関係は以下の図のようになります。



ともあれ、テキストシステムは基本的にNSTextViewを通してのインタラクションを簡単にすることを主眼に設計されています。これ以上の詳細は「Creating an NSTextView Programmatically」というドキュメントに当たってください。

NSTextStorageオブジェクトをセットアップする

NSTextStorageを生成する方法には何も特別なものはありません。allocメッセージでメモリを確保し、initメッセージを使って初期化します。つまり……最も単純な形は こうなります。

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

もちろん、何かのデータを与えてそれで初期化することも可能です。例えばファイルに収められているリッチテキストデータで初期化したければ こんな感じです。

```
NSAttributedString *attrString = [NSAttributedString
attributedStringFromRTF:[NSData dataWithContentsOfFile:filename]];
textStorage = [[NSTextStorage alloc] initWithAttributedString:attrString];
```

上の2つの例では、変数textStorageはこのメソッドを含むオブジェクトのインスタンス変数であると仮定しています。テキスト処理システムを手作りする場合は、この例のように必ずtextStorageオブジェクトへの参照だけを保持、使用します。NSLayoutManagerなど他のオブジェクトは、次に見るように全てこのtextStorageから参照できます。

NSLayoutManagerオブジェクトをセットアップする

では次にNSLayoutManagerを生成するコードです。

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

生成したオブジェクトはtextStorageに追加したら解放してしまいます。これは例えばオブジェクトをNSDictionaryやNSArrayにセットするのと同じで、textStorageがそのオブジェクトをretainする（そして前に見たように自身の解放と共に解放する）からです。

NSLayoutManagerはその役割（例えば文字をグリフに変換したりテキストを配置したり）を果たすため、いくつかの補助的なオブジェクトを必要とします。それらは初期化の際に自動的に生成（既にあるものを接続する場合もありますが）されます。プログラマがしなければならないのは、上のようにNSLayoutManagerをtextStorageに接続すること、そして次に見るように、初期化の際にNSTextContainerをNSLayoutManagerに接続することだけです。

NSTextContainerオブジェクトをセットアップする

次はNSTextContainerの生成です。このオブジェクトの初期化にはサイズが必要です。以下の例に出てくる変数theWindowは、テキストビューを表示するウインドウとして定義済みと仮定します。

```
NSRect cFrame = [[theWindow contentView] frame];
NSTextContainer *container;
container = [[NSTextContainer alloc] initWithContainerSize:cFrame.size];
[layoutManager addTextContainer:container];
[container release];
```

生成したオブジェクトはNSLayoutManagerに登録し、解放してしまいます。textStorageがNSLayoutManagerをretainしているのと同じ構造の繰り返しですからもうくどくどと説明しません。アプリケーションが複数のNSTextContainerを必要とするなら、ここで生成、追加、解放を繰り返します。

NSTextViewオブジェクトをセットアップする

最後に、NSTextView（あるいはNSTextViews……クラス名を複数形にするのはどうなんですかね。NSTextView Objectsの方が正しい英語のような気がしますが）を生成します。

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

NSTextViewの初期化メソッドが initWithFrame: textContainer:であることに注目してください。つまりこのコードにより、NSTextViewは渡されたNSTextContainerオブジェクトと接続された状態で生成されるわけです。これを通常の initWithFrame:と比較すると、この初期化メソッドは単にオブジェクトを初期化するだけでなく、テキスト処理システムのネットワーク構築の一翼を担っていることが解るでしょう。次に、生成されたNSTextViewをそれを表示するウインドウにセットし、解放します。これでテキスト処理に関わるオブジェクトのネットワークが完成したわけです。

ドキュメント更新履歴

オリジナル・ドキュメントは「Text System Overview」2005/08/11版。

要約は2006/02/26。アップルからオリジナルドキュメントにある図の使用許可が下りないため、できるだけ似せて（翻訳する必要があるところは翻訳し）作成したが、見苦しいところがあればご容赦願いたい。