Document-Based Applications Overview

イントロダクション

このドキュメントでは、複数のドキュメントファイルを作成し、オープンし、ロードし、セーブすることができるアプリケーションを作成するために、CocoaのApplication Kit が用意しているアーキテクチャについて解説したものです。

誰が読めばいいの？

Application Kit のドキュメント・アーキテクチャを使いたいデベロッパは全員このドキュメントを読むべきです。また、その内容を理解するためには、Cocoa プログラミング・パラダイムに関する知識およびObjective-Cに慣れ親しんでいる必要があるでしょう。

ドキュメント＝ベース・アプリケーション・アーキテクチャ

ドキュメント＝ベースのアプリケーションはよくあるアプリケーションの一つの形です。それはその内容こそさまざまですが、ファイルの中に保存できるデータを扱うという意味で共通のフレームワークを持っています。ワードプロセッサ、スプレッドシード・アプリケーションなどはドキュメント＝ベースのアプリケーションの代表的な例です。一般にドキュメント＝ベースのアプリケーションとは以下のことを行います。

・ドキュメントを新規に作成します。

・ファイルのなかに格納されている既存のドキュメントを開きます。

・ドキュメントをユーザの指定した場所にユーザの指定した名前で保存します。

・ドキュメントに加えられた変更を破棄し、最後に保存された時の状態に戻します。

・ドキュメントを閉じます（内容に変更があった場合にはユーザにその保存を促します）

・ユーザによるページレイアウトの指定を反映させてドキュメントを印刷します。

・内部では見た通りではないデータをユーザに見やすく提示する。

・ドキュメントの編集をモニタ、記録し、メニュー項目の選択の可否を統御する。

・そのタイトルの設定を含むドキュメント・ウインドウの管理を行う。

・アプリケーションとウインドウの間のデリゲーション・メソッドに応える（アプリケーションの終了時など）。

主要なクラス

Application Kit の3つのクラスが「ドキュメント・アーキテクチャ」と呼ばれるドキュメント＝ベース・アプリケーションのアーキテクチャを提供します。これは上にあげた各種の処理をインプリメントしなければならないデベロッパの作業を簡単にするものです。その3つのクラスとは、 NSDocument、NSWindowController、 そしてNSDocumentControllerです。

これらのクラスのオブジェクトはアプリケーションのドキュメントを作成し、サーブし、オープンし、管理する仕事を分担し、協調します。１つのアプリケーションは１つのNSDocumentControllerを持ち、１つのNSDocumentControllerはファイルごとにNSDocumentを作成して管理します。１つのNSDocumentは1個以上の NSWindowControllerを持ってファイルの内容を表示します。さらにこれらのうちいくつかのオブジェクトは、NSWindow、そしてNSApplicationのデリゲート・オブジェクトとなっており、ウインドウのクローズやアプリケーションの終了などのイベントに対処します。

ドキュメントって何でしょう？

概念的に、ドキュメントとはそれがディスクファイルに保存される時につけられる名称によって特定される、情報ののコンテナです。この意味で、ドキュメントはファイルは違います。ドキュメントはドキュメント・データを保持し管理するメモリ上のオブジェクトです。Application Kit に関わる文脈では、ドキュメントはその中身がどのようにウインドウに表現されるかを知っている NSDocument のサブクラスのインスタンスです。ドキュメントはファイルからデータを読み込み、またファイルにデータを保存することができます。そしてドキュメントに関わる多くのメニューコマンド、保存、復帰、プリントなどのコマンドに対するファースト・レスポンダでもあります。ドキュメントはウインドウ（に表示しているデータ）の編集状態をモニターし、アンドゥ、リドゥをセットアップします。そしてウインドウが閉じられるときにはその可否を尋ねられます。

NSDocumentのサブクラスを作成するには、いくつかのメソッドをオーバーライドすることが必須です。そして場合によっては他のメソッドもオーバーライドする必要があります。NSDocumentクラス自体は、ドキュメントの内容をひとつのまとまったデータとして扱う術を知っています。が、その特定のタイプをどう扱うかについては何も知りません。なのでサブクラスでは必ず、リード・ライトに関わるメソッドをオーバーライドして、読み書きが可能なタイプを特定し、そのデータが内部でどのように構造化されるかを記述しなければなりません。サブクラスはまた、ドキュメント・ウインドウを管理するウインドウ・コントローラを生成し、アンドゥとリドゥをインプリメントしなければなりません。ここまでが絶対に必要なオーバーライドということになりますが、アプリケーションの中で稼働しているNSDocument のサブクラスは、ドキュメントの管理のためにこれ以上のあれやこれやを行っているのが普通です。

ドキュメント＝ベース・アプリケーションにおけるキー・オブジェクトの役割

先に挙げたドキュメント・アーキテクチャの主要な３クラスのインスタンスは、互いに異なる役割をしかし協調しながら果たしています。

NSDocumentControllerの役割

NSDocumentControllerの第一の仕事はドキュメントの作成とオープン、そしてそれらを管理することです。ユーザがファイルメニューから「新規」を選ぶと、NSDocumentControllerオブジェクトは アプリケーションのプロパティ・リストの中の CFBundleDocumentTypes を参照し、設定されているNSDocumentサブクラスを知り、メモリをアローケートして initWithType:error: メソッドを呼び出してこれを初期化します。また、ユーザがファイルメニューから「開く」を選んだ場合には、オープン・パネルを表示し、ユーザの選択を待ちます。選択されたファイルに適した NSDocument のサブクラスを探し、これをアロケートしてから初期化、次に initWithContentsOfURL:ofType:error: を使ってドキュメント・データをロードします。どちらの場合もNSDocumentConttroller オブジェクトは今後の管理を容易にするために、作成されたドキュメントを内部のリストに追加します。このとき追加されるドキュメントはそのウインドウをメイン・ウインドウにされ（ユーザ・イベントの主な対象になる最前面のウインドウをメイン・ウインドウと呼びます）、そのドキュメントはカレント・ドキュメントとして記憶されます。

NSDocumentControllerオブジェクトはまた、アプリケーションが最近処理したドキュメントのURLを維持することで、アプリケーションの「最近使った書類」メニューを管理します。ここには最近オープンされた、セーブされた、リバートされた、あるいはクローズされたドキュメントが記憶されます。

NSDocumentControllerオブジェクトはは特定のアプリケーション・イベント、アプリケーションの起動、その終了、システムのシャット・ダウン、そしてFinderからのドキュメントのオープン、プリントの要求など、に必ず応じるよう結びつけられています。もし望むのであれば、プログラマはNSDocumentControllerをサブクラスするなどして、これらのイベントに自分で応えることも可能です。しかしながらNSDocumentControllerオブジェクトは事実上、デフォルトのままでほとんどの状況に的確に対応できる優れたコントローラです。通常これをサブクラスする必要が生じることはないはずです。アバウト・パネルの表示やアプリケーションの環境設定のハンドリングに関しては、NSDocumentControllerをサブクラスして対応するより、それらを専門に行うカスタム・コントローラを作ることをお勧めします。それでもなお、いやどうしてもワタシはNSDocumentControllerをサブクラスしなければならない、というのであれば、後述の「NSDocumentControllerのサブクラスを作る」の項を精読してください。

NSDocumentの役割

NSDocumentオブジェクトの第一の仕事はドキュメントに付帯するデータを表示し、操作し、セーブし、ってロードすることです。すなわちこれはモデル・コントローラです。アプリケーションのプロパティ・リストの CFBundleDocumentTypesで指定されているタイプのドキュメントに対して以下のことができなければなりません

・内部にドキュメント・タイプのデータを展開し、ウインドウにそれを表示する

・ファイルシステムの指定され対置のファイルにドキュメント・データを保存する

・ファイルの中に保存されているデータを読み込む

ウインドウ・コントローラの助けを借りて、NSDocument オブジェクトはウインドウに対するデータの表示とその編集を管理します。ユーザがドキュメントのセーブ、プリント、復帰、クローズを要求すると、Application Kit はメイン・ウインドウを管理しているNSDocumentオブジェクトをそのイベントに対するファースト・レスポンダとして処理をゆだねます。それらの要求に応えるため、NSDocumentはセーブ・パネルやページ設定パネルをいかに表示し、利用するかを知っています。

フル実装されたNSDocumentオブジェクトは編集状態をどのように追跡し、データをどう印刷し、アンドゥとリドゥをいかに実装するかを知っています。デフォルトでこれらの完全な機能を提供する、というわけにはいきませんが、NSDocumentオブジェクトはこれらの実装をかなり簡単なものにしています。例えば編集状況の追跡に関して言えば、NSDocumentオブジェクトは加えられた変更を記録するカウンターを更新するAPIを提供します。アンドゥ、リドゥに関して言えばそのリクエストに応えるため、NSDocument は NSUndoManager を生成し、またそれが要求されるたびに変更カウンターを更新します。最後にぷりんとに関してもNSDocumentオブジェクトはページ設定パネルをの表示とプリントに使用されるNSPrintInfoオブジェクトの変更を簡便にしています。

NSWindowControllerの役割

NSWindowControllerは、通常nibファイルで定義されドキュメントに結びつけられているウインドウ1つを管理するビュー・コントローラです。もしドキュメントが複数のウインドウを持つなら、そのそれぞれにウインドウ・コントローラ・オブジェクトが生成されます。例えば一つのドキュメントに、そのメインであるデータ入力ウインドウの他に、ユーザに選択肢を提示するウインドウがあるような場合、そのそれぞれに NSWindowController オブジェクトが存在します。自身の所有者であるNSDocument オブジェクトから要求があると、NSWindowControllerオブジェクトはウインドウに関するデータをnibファイルからロードし、画面に表示します。同時に、そのウインドウの（変更されたデータを保存した後の）クローズに関して責任を負います。

また、NSWindowControllerオブジェクトは、ドキュメント＝ベース・アプリケーションに、ウインドウ同士が完全に重なりあわないよう階段状に表示されるような機能を付加します。

NSWindowControllerをサブクラスするのは自由です。アプリケーションはしばしばデフォルトのインスタンスを使用していますが、このオブジェクトをサブクラスすれば、nibファイルのローディングのやり方を変更したり、ウインドウのタイトルをカスタマイズしたりすることができます。

用法パタン

このセクションでは、ドキュメント・アーキテクチャの使い方を3種、簡単な順に説明します。

ドキュメント・アーキテクチャの最も簡単な用法は、１つしかウインドウを持たず、コントローラをモデル・コントローラとビュー・コントローラに分けることにそれほどのメリットがないドキュメントに対して適用することです。この場合、やらなくてはならない作業は単にNSDocumentのサブクラスを作ることだけです。NSDocumentのサブクラスは、モデルのための領域を確保し、ドキュメント・データをロード、セーブできれなければなりません。その他、必要に応じてユーザ・インタフェースのためのアウトレットの管理とアクションを実装します。また、このオブジェクトはスーパークラス（NSDocument）のwindowNibNameメソッドをオーバーライドし、このタイプのドキュメントに使われるnibファイルの名前を返すようにします。NSDocumentオブジェクトは自動的にNSWindowControllerオブジェクトを作成し、このnibファイルの管理をまかせますが、nibファイルの所有者（owner）は変わらずNSDocumentオブジェクトです。

ドキュメントが１つのウインドウしか持たないにも関わらず、コントロール・レイヤーの中で論理的な分離が必要なほど複雑な場合には、NSDocumentだけでなくNSWindowControllerもサブクラスすることができます。この場合、全てのアウトレット、アクション、ユーザ・インタフェース・アイテムの振る舞いがそのNSWindowControllerサブクラスの管理下に入ります。 NSDocumentサブクラスは windowNibNameではなく makeWindowControllers メソッドをオーバーライドして、NSWindowControllerのサブクラスを生成し、 addWindowControllerメソッドを使ってそれをウインドウ・コントローラの管理リストに追加しなければなりません。ビュー・コントロールのロジックとモデル・コントロールのロジックを分けるため、NSWindowControllerオブジェクトはnibファイルの所有者になります。このアプローチは上の最も簡単なケースで済まない場合の第一のお勧めです。

ドキュメントが複数のウインドウを必要とする、またはオプションで表示することができる場合には、NSDocumentクラスだけでなく、NSWindowControllerもサブクラスしなければならないでしょう。上の項で説明したように、 makeWindowControllersをオーバーライドしてNSWindowControllerのサブクラスを生成します。ただ、このケースではときとしてNSWindowControllerの異なる複数のサブクラスから複数のインスタンスを生成する必要があります。そのアプリケーションが１つのドキュメントを表示するために複数の違った種類のウインドウを必要とするような場合です。その場合、それらのウインドウのためにNSWindowControllerのサブクラスが何種類か必要になるでしょうし、オーバーライドしたmakeWindowControllers でそれらのインスタンスを生成しなければなりません。またアプリケーションによっては、1ドキュメントに必要なのは1ウインドウでありながら、ユーザにそのコピーの作成を許容したいものもあります（3Dグラフィックス・ソフトウエアのマルチビュー・ウインドウなど）。このような場合、 makeWindowControllersオーバーライドが生成するNSWindowController オブジェクトは１つかもしれませんが、ユーザに他のウインドウを作成させるためのメニューコマンドなどの実装が必要でしょう。

ドキュメントとスクリプティング

スクリプティングに対するサポートはドキュメント・アーキテクチャに基づくアプリケーションにおいてはほぼ自動的に提供されます。まず第一に、NSDocument 他のドキュメント・アーキテクチャに関わるクラスは直接、標準ドキュメント・スクリプティング・クラスとしてAppleScript で使用可能なように実装されており、ドキュメントに関する多くのスクリプティング・コマンドをサポートします。第二に、ドキュメント・アーキテクチャはモデル＝ビュー＝コントローラ（MVC）セパレーションに基づいてデザインされたアプリケーションと協調して動作するよう作られており、スクリプティング・サポートがこの同じデザインにマッチしているため、そうデザインされていないアプリケーションに比べてスクリティングのサポートを行うのに適しています。最後に、ドキュメントはほとんどのアプリケーションにおけるスクリプティングAPIで重要な役割を担います。NSDocumentクラスはアプリケーションがそのモデル・レイヤーに対するスクリプティング・アクセスをサポートするための良いとっかかりを提供します。

アプリケーションがドキュメント・アーキテクチャに基づいていない場合、それをスクリプタブルにする仕事はそのアプリケーションを作る仕事をもう一度やるくらいのボリウムになります。例えばサンプルプログラムのTextEditはNSDocument に基づかずにどうドキュメント＝ベース・アプリケーションを作るかを示す好例です。これをNSDocumentベースでスクリプタブルを実現しているサンプル、Sketch と比較してみてください。

ドキュメント＝ベース・アプリケーションの実装

ドキュメント＝ベース・アプリケーションをまとめるためにそれほど多くのコードを書く必要はありません。要求される仕様が最低限のものでよければ、Application Kit から提供されるデフォルトのNSWindowController とNSDocumentControllerを使用することができます。プログラマはドキュメント・プロジェクトを作成し、ヒューマン・インタフェースを構成、NSDocumentのサブクラスを実装、あとはそのアプリケーションの処理に必要なカスタム・クラスや動作を作り込めば完成してしまいます。

以下のセクションでは、ドキュメント＝ベース・アプリケーションを開発するにあたってプログラマがしなければならないいくつかの仕事について順を追って解説します。まずはドキュメント＝ベース・アプリケーションの根幹をなす３つのクラスについて以下の表に示します。

クラス	オブジェクトの数	サブクラス
NSDocumentController	アプリケーションごとに1個	必要に応じて（めったにない）
NSDocument	ドキュメントごとに1個	必須
NSWindowController	ウインドウごとに1個	必要に応じて（大抵は必要）

ドキュメント＝ベース・アプリケーション・プロジェクトのひな形

Cocoa の開発環境はドキュメント＝ベース・アプリケーションの開発を始めるためのひな形を Xcodeに用意しています。このひな形は以下のものを提供します。

＊　アプリケーションのドキュメントのためのnibファイル

このnibファイルは、MyDocument.nibと名付けられます。MyDocumentという名前のNSDocumentのサブクラスがこのファイルの所有者になります。それはアウトレットとしてウインドウを（windowという名前になっています）を持ち、そのウインドウには１つだけテキストフィールドが配置され、「Your document contents here」と表示しています。

＊　アプリケーションのメイン nibファイル

このnibファイルはMainMenu.nibと名付けられます。これにはアプリケーションメニュー、ファイルメニュー（ドキュメントに関するコマンドはこの中に含まれます）、テキストの編集やアンドゥ、リドゥ（日本語では「やり直し」などになっています）を含む編集メニューが含まれます。ファイルメニューの全ての項目と同様、これらのメニュー項目はファースト・レスポンダの適切なアクションに結びつけられます。「NewApplicationについて」の項目は、標準のアバウト・ウインドウを表示するアクション、orderFrontStandardAboutPanel: に接続されています。

＊NSDocumentのサブクラスのスケルトン（MyDocument.hとMyDocument.m）

後者のファイルはサブクラスで実装しなければならない dataRepresentationOfType: 、loadDataRepresentation:ofType: という２つのメソッドの空のブロックを含んでいます。このスケルトンはMac OS X 10.3、またはそれ以前のシステムで動作するアプリケーション用です。Mac OS X 10.4以降のシステムに稼働環境を限定できるアプリケーションでは、これらの代わりに dataOfType:error: と readFromData:ofType:error: という２つのメソッドをオーバーライドするべきです。またこのファイルは、ドキュメント・ウインドウの定義ファイルの名前、「MyDocument」を返すようになっている windowNibNameメソッドの実装と、 windowControllerDidLoadNib: のオーバらライドを含んでいます。

＊アプリケーションの情報プロパティ・リスト

Xcode のターゲット・インスペクタを使うと、ファイル、info.plistの内容を編集することができます。このファイルにはアプリケーションが扱えるファイルのタイプを特定する CFBundleDocumentTypes を含むグローバルなアプリケーションキーとそれに対応する値が収められています。

以下のセクションではXcode の Cocoa ドキュメント＝ベース・アプリケーション・テンプレートを使用してドキュメント・ベースのアプリケーションを作成していく過程を説明します。なお、以下の表はテンプレートに用意されるファイルメニューの各項目が初期状態でファースト・レスポンダのどんなアクションに接続されているかを示したものです。

ファイルメニュー項目	ファースト・レスポンダのアクション
新規	newDocument:
開く...	openDocument:
最近使った書類>メニューを消去	clearRecentDocuments:
閉じる	performClose:
保存	saveDocument:
別名で保存	saveDocumentAs:
最後に保存した状態に戻す	revertDocumentToSaved:
ページ設定...	runPageLayout:
プリント...	printDocument:

テンプレートには編集メニュー、ウインドウメニューおよびヘルプメニューの項目について同様の接続があります。アプリケーションがプリントなど、設定される機能についてサポートしないのであれば、nibファイルを編集してこれらの接続を解除しておくべきでしょう。

プロジェクトを作成しインタフェースを構成する

1. Xcodeを起動し、ファイルメニューから新規プロジェクトを選択します。ダイアログが表示されたら、「Cocoa Document-based Application」を選択します。

2. プロジェクトを保存するディスク上の場所と、プロジェクトの名前を決めます。

3. Xcode の「グループとファイル」にある「リソース」グループの中の「MyDocument.nib」ファイルをダブルクリックします。このファイルは Interface Builderによってオープンされます。nibファイルの名前を変えたい場合は、Interface Builderでそれを別の名前で保存し、プロジェクトに追加することができます。ただし、これを行った場合には、自動生成されたNSDocumentサブクラスの実装における、windowNibName メソッドが返す文字列を変更しなければなりません。

4. Interface Builderで、ドキュメント・ウインドウに各種インタフェースを作成します。

5. ドキュメント・ウインドウの上に作成したオブジェクトがアウトレットやアクションを必要とする場合にはそれらをNSDocumentのサブクラス、MyDocumentに追加します。アクションやアウトレットを、nibファイル・ウインドウにあるFile's Owner のアイコンに接続します。

重要：これらの接続を行うためにMyDocumentのインスタンスを生成してはいけません。

NSDocument のサブクラスの名称を MyDocument以外のものにしたい場合には、Interface Builder上の名前と、ファイル名（ヘッダー、MyDocument.hとインプリメンテーション MyDocument.m）を変更してください。これを行った場合は、忘れずにindo.plistファイルの NSDocumentClass キーに対応する文字列も変更してください。

6. ドキュメント・オブジェクトが他のカスタム・オブジェクト、例えば特別な計算を行うモデル・オブジェクトなどと連絡する必要があれば、Interface Builderでそれらのオブジェクトを定義し、接続を行ってください。

情報プロパティ・リストを完成させる

1. Xcodeのターゲットグループの三角形をクリックして、表示されたアプリケーション・オブジェクトを選択します。ツールバーの情報アイコンをクリックします（ファイルメニューから「情報を見る」を選んでも同じです）。現れたウインドウ（「ターゲット”アプリケーション名”の情報」というタイトルがついています）の「プロパティ」タブをクリックします。

2. 情報プロパティ・リストに設定されているデフォルト値をアプリケーションで使用する値に変更します。ウインドウの下の方にある、「”info.plist”をファイルとして開く」というボタンを使うと、このプロパティ・リスト・ファイルをテキストファイルとして編集することもできます。

NSDocumentのサブクラスを実装する

以下の手順はあくまで一般的なガイドラインとして提示するものです。詳細については15ページ「NSDocumentのサブクラスを作成する」を参照してください。また、アンドゥ、リドゥおよびプリントに関してはCocoaの該当するドキュメントをお読みください。

1. Xcodeで、NSDocumentサブクラスのヘッダーファイルを開きます（このファイルはグループ&ファイル・ペーンのクラス・グループにあります）。

2. Interface Builderでこのサブクラスにアウトレットやアクションを加えた場合には、その定義をこのヘッダーに追加します。必要があればインスタンス変数を定義し、interface Builderの接続を介さないメソッド（例えばインスタンス変数に対するアクセサ・メソッドなど）の宣言も追加します。また、ここで新しいアウトレットやアクションを追加し、Interface BuilderのClassesメニューにあるRead Filesコマンドでそれらをnibファイルにインポートすることも可能です。

3. プロジェクトのクラス・グループからサブクラスのインプリメンテーションファイル（拡張子が「.m」のもの）を開きます。

4. 通常その必要はありませんが、プログラマは 指定された初期化メソッド（init）と、特定のサブクラスを初期化するメソッド、 initWithContentsOfFile:ofType: をオーバーライドすることができます。これを行う場合には、必ずスーパークラスのインプリメンテーションを呼び出してください。また、ドキュメントnibファイルから読み出されたオブジェクト（これはドキュメントそのものではありません）を初期化するawakeFromNib メソッドを実装することもできます。

5. データを介したリード／ライト・メソッドをオーバーライドします。Mac OS X 10.4以降だけを稼働環境とするアプリケーションでは、 readFromData:ofType:error: (このメソッドは特定のタイプのドキュメント・データをロードします）そして dataOfType:error: （このメソッドは特定のタイプのドキュメント・データを保存するために準備します）をオーバーライドします。以下のサンプルは、NSTextStorageオブジェクトにドキュメント・データを保持し、アプリケーションはこれを表示するNSTextViewをドキュメント・ウインドウごとに持っている、と仮定したものです。NSDocumentオブジェクトにはドキュメントが保持する NSAttributedString データ・モデルに対するアクセサ・メソッド、text とsetText: が用意されています。

```
- (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 setText:fileContents];
        [fileContents release];
    }
    return readSuccess;
}
```

```
- (NSData *)dataOfType:(NSString *)typeName
        error:(NSError **)outError {
    NSData *data = [textView RTFFromRange:
        NSMakeRange(0,[[textView textStorage] length])];
    if (!data && outError) {
        *outError = [NSError errorWithDomain:NSCocoaErrorDomain
        code:NSFileWriteUnknownError userInfo:nil];
    }
    return data;
}
```

Mac OS X 10.3かそれ以前のシステムで稼働しなければならないアプリケーションでは、これらの代わりにloadDataRepresentation:ofType: と dataRepresentationOfType: メソッドを実装しなければなりません。

6. Mac OS X 10.4以降をターゲットにしたアプリケーションでは、ドキュメント・データ・ファイルにアクセスするために readFromURL:ofType:error: と writeToURL:ofType:error: をオーバーライドします。もしドキュメントの書き込みが、書き換えられようとしているドキュメントのレビジョン表示へのアクセスを必要とする場合には writeToURL:ofType:error: の代わりに writeToURL:ofType:forSaveOperation:originalContentsURL:error: をオーバーライドしてください。また、ドキュメントデータがファイルパッケージの内部に保存されている場合には、 これらの代わりに readFromFileWrapper:ofType:error: と fileWrapperOfType:error: をオーバーライドします。以下は先ほどのサンプルと同じ条件下でURLベースの読み書きを実装したサンプルです。

```
- (BOOL)readFromURL:(NSURL *)inAbsoluteURL
        ofType:(NSString *)inTypeName error:(NSError **)outError {
    BOOL readSuccess = NO;
    NSAttributedString *fileContents = [[NSAttributedString alloc]
                    initWithURL:inAbsoluteURL options:nil
                    documentAttributes:NULL error:outError];
    if (fileContents) {
        readSuccess = YES;
        [self setText:fileContents];
        [fileContents release];
    }
    return readSuccess;
}

- (BOOL)writeToURL:(NSURL *)inAbsoluteURL
        ofType:(NSString *)inTypeName error:(NSError **)outError {
    NSData *data = [[self text] RTFFromRange:NSMakeRange(0,
                    [[self text] length]) documentAttributes:nil];
    BOOL writeSuccess = [data writeToURL:inAbsoluteURL
    options:NSAtomicWrite error:outError];
    return writeSuccess;
}
```

前項と同様、 Mac OS X 10.3かそれ以前のシステムで稼働しなければならないアプリケーションでは、これらの代わりに readFromFile:ofType: と writeToFile:ofType: メソッドをオーバーライドしなければなりません。

7. NSDocument が使用する１つあるいは複数のウインドウ・コントローラ・オブジェクトを生成するメソッドをインプリメントします。ドキュメントが１つのウインドウしか必要としないなら、プロジェクト・テンプレートが用意したデフォルトのインプリメンテーションを使うこともできます。

```
- (NSString *)windowNibName {
    return @"MyDocument";
}
```

ドキュメントが複数のウインドウを必要とするか、あるいはNSWindowController のカスタム・サブクラスを使用する場合には、 makeWindowControllers をオーバーライドして、作成したウインドウ・コントローラを addWindowController: によって必ずウインドウ・コントローラの管理リストに追加するようにしてください。なお、このメソッドはドキュメントにウインドウ・コントローラをリテインさせるので、管理リストに加えたあとでそれをリリースすることをお忘れなく。

8. ウインドウがnibファイルからロードされる前後に必要なタスクを行うために、windowControllerWillLoadNib: および、 windowControllerDidLoadNib: をインプリメントすることができます。以下に例を示します。

```
- (void)windowControllerDidLoadWindowNib:(NSWindowController *)windowController {
    [super windowControllerDidLoadWindowNib:windowController];
    [textView setAllowsUndo:YES];
    if (fileContents != nil) {
        [textView setString:fileContents];
        fileContents = nil;
    }
}
```

9. ドキュメントが変更されたら、ダーティフラッグをマークするようにします。 isDocumentEdited の戻り値は、そのドキュメントに保存されていない変更があるかどうかを示します。NSDocumentはドキュメントがセーブされるかまたは最後に保存された状態に復帰された場合にこのフラグをクリアしますが、プログラムがNSDocument デフォルトのアンドゥ／リドゥ・メカニズムを使用していない場合、編集に応じてこのフラグをセットするのはプログラマの責任になります。通常プログラマは、ユーザがドキュメントを編集すると、デリゲートあるいはノティフィケーションによってそれを知り、 NSChangeDone を引数にして updateChangeCount: メソッドを呼び出すことでダーティフラッグをセットします。

10. ドキュメント・データを印刷するためのコードを書きます。ユーザに印刷機能を提供したいなら、printOperationWithSettings:error: メソッドをオーバーライドし、ことによると NSPrintInfo オブジェクトを変更しなければなりません。

11. コードにアンドゥとリドゥ・グループを登録します。詳しくは NSUndoManager の説明をお読みください。

そして最後に、当然ながらそのプログラムだけが持つ特別な機能、そのアプリケーションを作成する理由となった機能を NSDocument サブクラスに追加しなければなりません。

追加コントローラ・クラスのインプリメント

Application Kit から提供される NSWindowController のインスタンスが要求される仕様を満たさない場合には、このクラスのカスタム・サブクラスを作成することができます。その場合、NSDocument の makeWindowControllers メソッドをオーバーライドしてそのカスタムクラスのインスタンスを生成し、それをドキュメントのウインドウ・コントローラ・リストに追加しなければなりません。また、作成した NSWindowController のサブクラスが、nibファイルの所有者になっていることも保証しなければなりません。

また、デフォルトの NSDocumentControllerがなんらかの意味でアプリケーションの要求仕様を満足できない場合……、例えばユーザ初期設定のハンドリングとか特殊なアプリケーションのデリゲート・メッセージに応答する必要があるとか、そうした場合、NSDocumentControllerをサブクラスするのではなく、それらを処理する別のコントローラ・オブジェクトを作成してください。 NSDocumentControllerおよび、NSWindowControllerのサブクラスに関する詳細な情報については、それぞれのクラスのドキュメントを参照してください。また、「NSDocumentControllerのサブクラスを作成する」（19ページ）、「よくある質問」（42ページ）も参照してください。

NSDocumentのサブクラスを作成する

ドキュメント・アーキテクチャを利用するあらゆるアプリケーションが、少なくとも１つはNSDocumentのサブクラスを作成しなければなりません。このアーキテクチャはそのサブクラスによっていくつかのメソッド（必須のものも必要に応じてのものもありますが）がオーバーライドされることを前提としており、状況によっては他のいくつかのメソッドのオーバーライドも推奨されます。このセクションではNSDocument のサブクラスがしなければならないオーバーライドに加えて、init や displayNameなど、状況によってオーバーライドされることもあるメソッドについても解説します。

オーバーライドが必須のメソッド

以下の項目で説明されている機能は、NSDocument のサブクラスによってオーバーライドされることによって実現されます。プログラマはまず、リードとライトに関わるメソッドを１つずつオーバーライドしなければなりません。最も簡単な場合、オーバーライドしなければならないメソッドは２つだけです。もしアプリケーションがファイル・ロケーションをケアする必要があるなら、URLを使ってファイル位置を特定するタイプのリード／ライト・メソッドをオーバーライドすることになります。また、アプリケーションがファイル・パッケージになっているドキュメントをサポートするような場合には、ファイル・ラッパーを通してリード／ライトが行えるメソッドをオーバーライドしなければなりません。

なお、これらのメソッドはみんな、ファイル全体を一度に読み書きすることにも注意しなければなりません。アプリケーションが巨大なデータセットを扱う場合、リード／ライト・メソッドはその読み書きを数回に分けて行うようにしなければならないかもしれません。

データを介したリード／ライト・メソッド

dataOfType:error: メソッドはドキュメントがサポートするタイプのデータを作成し、NSData オブジェクトにパッケージして返します。このメソッドは通常ドキュメント・データをファイルに書き出すために使われます。 readFromData:ofType:error: メソッドは NSData オブジェクトの形で読み込まれたドキュメント・データをドキュメントの内部形式に変換し、ドキュメント・ウインドウに表示するためにオーバーライドされます。

Mac OS X 10.3とそれ以前のシステムで稼働しなければならないアプリケーションでは、これらの代わりにdataRepresentationOfType: と loadDataRepresentation:ofType: がオーバーライドされる必要があります。

ロケーションを介したリード／ライト・メソッド

デフォルトでは、 dataOfType:error: メソッドによって用意されたデータを fileWrapperOfType:error: メソッドがラップし、writeToURL:ofType:error: メソッドが ファイルに書き出します。そして読み込みに当たっては、 readFromURL:ofType:error: メソッドがまずデータをファイルから読み出してNSFileWrapper オブジェクトを作成。このオブジェクトがパッケージでなく単なるファイルから読み出されたものだった場合には、 readFromData:ofType:error: に渡され、そうでない場合にはreadFromFileWrapper:ofType:error: に渡されます。 NSDocument のサブクラスがデフォルトのデータを介したリード／ライト・メソッドで要求される仕様を満足できない場合には、これらのメソッドのいずれでもオーバーライドすることができます。しかしながらそのオーバーライドはデータを介したリード／ライト・メソッドのローディングを引き受けなければなりません。

Mac OS X 10.3とそれ以前のシステムで稼働しなければならないアプリケーションにおいては、これらの代わりに writeToFile:ofType:、 fileWrapperRepresentationOfType:、 readFromFile:ofType:、そして loadFileWrapperRepresentation:ofType: をオーバーライドすることになります。

（訳注：この項、原題が「Location-based reading and writing method」なので上のように訳しておきましたが、内容的には「ファイル・パッケージを介した〜」としたほうが正しいようです）

リード／ライト・メソッドのオーバーライドに関する注意

まず第一にオーバーライドしたメソッドから fileURL（または fileName……このメソッドはMac OS X 10.4では既に推奨されません）、fileType、 fileModificationDate などのメソッドを呼び出さないでください。読み込みの間、特にオブジェクトの初期化に伴う読み込みの間は、たとえばファイルのロケーションやタイプなどのNSDocumentのプロパティが設定され終わっているという保証がありません。オーバーライド・メソッドは、読み込みに必要な全ての要素を渡されたパラメータから知ることができるべきです。また、書き出しに際しても、ドキュメントは異なった位置に異なったタイプでその中身を書き込めと命じられるかもしれません。繰り返しになりますが、オーバーライド・メソッドは、書き出しに必要な全ての要素も渡されたパラメータから知ることができるべきです。

もしオーバーライドしたメソッドが必要とする情報をどうしてもパラメータから得られない場合には、別のメソッドをオーバーライドすることを考えてください。例えば readFromData:ofType:error: のオーバーライド・メソッドから fileURL を呼び出す必要が生じてしまった場合は、 readFromURL:ofType:error: をオーバーライドするべきなのです。もう一つ、 writeToURL:ofType:error: のオーバーライド・メソッドから同じく fileURLを呼び出したくなったら、迷わず writeToURL:ofType:forSaveOperation:originalContentsURL:error: をオーバーライドしてください。

必要に応じてオーバーライドされるメソッド

以下に挙げた機能に関しては、いくつかの状況でオーバーライドが必要となります。

＊ウインドウ・コントローラの生成

NSDocument のサブクラスはウインドウ・コントローラを生成しなければなりません。これには間接、直接２つのやり方があります。ドキュメントが１つのnibファイルと１つのウインドウしか持たないのであれば、そのnibファイルの名称を返すよう、 windowNibName メソッドをオーバーライドできます。結果的にデフォルトの NSWindowController インスタンスはここで指定されたnibファイルの所有者として生成されます。ドキュメントが複数のウインドウを持っていたり、サブクラスしたNSWindowControllerのカスタマイズ版を使用したりする場合には、NSWindowController のサブクラスを生成するためにNSWindowController をオーバーライドする必要があります。

＊プリントとページ設定

通常、ドキュメント＝ベース・アプリケーションではドキュメント・データをどのように印刷するかという情報を変更できるようになっています。この情報は NSPrintInfo というオブジェクトにカプセル化されていて、アプリケーションがこの情報を参照してプリントを行うために、NSDocument のサブクラスは printOperationWithSettings:error: をオーバーライドしなければなりません。もしアプリケーションがMac OS X 10.3、あるいはそれ以前のバージョンで稼働しなければならない場合には、代わりに printShowingPrintPanel: をオーバーライドしてください。なお、アプリケーションがプリントをサポートしないのであれば、Xcode がテンプレートに用意したnibファイルの、印刷関係のメニュー・コマンドからの接続を必ず解除しておいてください。

＊バックアップファイルの作成

ドキュメントを保存するとき、新しいデータを書き出す前に、NSDocumentは置き換えられる元のファイルのバックアップを作成します。このバックアップファイルは、新しいファイルと同じ名前を持っていますが、拡張子の直前にティルド記号が付け加えられています。書き出し処理が正常に終了した時点でこのファイルは削除されますが、サブクラスは keepBackupFile メソッドをオーバーライドしてYESを返すようにし、最新のバックアップファイルを残すように変更することができます。

＊セーブ・パネルのアクセサリービューを変更する

NSDocument がセーブ・パネルを表示するとき、もしそのドキュメントが複数のドキュメント・タイプを書き込むことが可能であれば、デフォルトでパネルの下部にアクセサリービューが表示されます。このビューは書き込み可能なタイプのポップアップ・メニューを持ち、ユーザに書き込むタイプの選択を許します。アプリケーションがこのポップアップ・メニューの表示を望まない場合には、shouldRunSavePanelWithAccessoryView をNOを返すようにオーバーライドしてください。また、prepareSavePanel: をオーバーライドすれば、他の点に関してもセーブ・パネルをカスタマイズすることが可能です。

＊メニュー項目の有効化

NSDocument は「最後に保存した状態に戻す」と「別名で保存」という２つのメニューについて、その有効・無効を設定するために validateUserInterfaceItem: を実装しています。アプリケーションがこれ以外のメニュー項目に関しても状況によって有効・無効を設定したいときは、このメソッドをオーバーライドします。なお、その際にはくれぐれもスーパークラスのインプリメンテーションをコールするのを忘れないでください。この機構について詳しくは、 NSUserInterfaceValidations プロトコルのドキュメントを参照してください。

初期化に関して

NSDocument サブクラスの初期化に関してはもうひとつ知っておくべき問題があります。デフォルトの初期化メソッドとして定義されている init メソッドは、 initWithType:error: や initWithContentsOfURL:ofType:error: といった他の初期化ルーチンから呼び出されることを前提としています。もしドキュメントが新規作成されるときだけ必要で、既存のファイルを読み込んで生成される場合には行う必要がないような初期化処理があれば、 initWithType:error: をオーバーライドしてそれを行います。また逆に、既存のファイルをオープンする場合にのみ行いたい初期化処理がある場合には initWithContentsOfURL:ofType:error: をオーバーライドして行ってください。もちろんそうした状況に関わらず必要な初期化に関しては init をオーバーライドします。なおどちらのケースでも最初にスーパークラスのインプリメンテーションを呼び出すのを忘れないでください。

アプリケーションが Mac OS X 10.3以前のシステムで稼働することを求められる場合には、上の２つのメソッドの代わりに initWithContentsOfFile: あるいは initWithContentsOfURL: をオーバーライドします。

なお、init をオーバーライドした場合、絶対に nil をリターンしないようにしてください。nil を返すと Application Kit のバージョンによってプログラムがクラッシュすることがありますし、そこまで行かなくてもあんまり役に立たないメッセージが表示されることになります。もし、例えばそのアプリケーション特有の理由によってドキュメントの作成を中止したい場合には、init の代わりに特定の NSDocumentController のメソッドをオーバーライドして対処してください。

ドキュメント・ウインドウのタイトルのカスタマイズ

NSDocument のサブクラスでは、displayName メソッドをオーバーライドしてドキュメント・データを表示しているウインドウのタイトルをカスタマイズするケースがあります。 displayName が返すドキュメントのディスプレイ・ネームは単にウインドウのタイトルに使われるだけのものではないので、この方法はあまり正しいやり方とは言えません。例えばディスプレイ・ネームは以下のようなところで使われています。

・ドキュメントの印刷、保存、あるいは復帰などの際に表示されるアラート・ボックス

・ドキュメントの保存の際、ファイルが移動されていたり、リネームされていたり、ごみ箱に入れられていたりした場合に表示されるアラート・ボックス

・ユーザがその内容を保存しないでウインドウを閉じようとした際に表示されるアラート・ボックス

・別名で保存する際にセーブ・パネルに表示するデフォルトのファイル名として

ドキュメント・ウインドウのタイトルをカスタマイズする正しい方法は、NSWindowControllerをサブクラスして windowTitleForDocumentDisplayName: メソッドをオーバーライドすることです。もしアプリケーションがさらに深いカスタマイズを必要とする場合には、synchronizeWindowTitleWithDocumentName をオーバーライドしてください。

NSDocumentControllerのサブクラスを作成する

NSDocumentController クラスは、アプリケーションのセッション中、 sharedDocumentController クラスメソッドによって作成された NSDocumentController （あるいはそのサブクラス）の最初のインスタンスを、保持し続けます。アプリケーションが NSDocumentControllerのカスタム・サブクラスを使うようにするためには、起動時に作成される NSDocumentController のインスタンスがそのサブクラスのものであることを保証する必要があります。その方法は２つあります。

1. サブクラスをメインのnibファイルに作成します。

   メイン nibファイルはアプリケーションの起動時にロードされます。このファイルの中にNSDocumentControllerのサブクラスのインスタンスが定義されていれば、アプリケーションはそれを共有されたドキュメント・コントローラとしてロードします。アプリケーションのデリゲートとしてデフォルトの NSDocumentController オブジェクトが接続されていたら、これをサブクラスのインスタンスに変更するのをお忘れなく。

2. applicationWillFinishLaunching: メソッドでサブクラスのインスタンスを生成します。

   アプリケーションは自身のデリゲートにapplicationWillFinishLaunching: メッセージを送るまでNSDocumentControllerのクラスメソッド、 sharedDocumentController を呼び出しません。したがって、デリゲートにapplicationWillFinishLaunching: が送られてきたタイミングでサブクラスを生成すれば、このインスタンスがドキュメント・コントローラとして設定されることになります。

ドキュメント・アーキテクチャのメッセージ・フロー

ドキュメント・アーキテクチャを構成するオブジェクトは、ドキュメント＝ベース・アプリケーションの要件を満たすため相互に働きかけを行います。その相互作用は主としてオブジェクト同士のパブリック・APIによるメッセージのやりとりという形をとります。この仕組みはプログラマに、NSDocument やその他のオブジェクトをサブクラスしてメソッドをオーバーライドすることで、アプリケーションの動作をカスタマイズする機会を保証しています。

この章ではMac OS X 10.4以降のシステムにおけるドキュメント・アーキテクチャでの主要なオブジェクト間のメッセージ（自身に送るメッセージも含みます）のやりとりを追い、そのメカニズムについて解説します。

なお、ここでの説明は、 過去のドキュメント＝ベース・アプリケーションがオーバライドしていて、Mac OS X 10.4で推奨されなくなったメソッドと同様、 ドキュメント・アーキテクチャがJavaをサポートするためのコード・パスをカバーしていません。なお、推奨されなくなったメソッドのオーバーライドはいまのところきちんと動作します。ただ、Mac OS X 10.4で導入された強固なエラー処理などのアドバンテージを教授できません。しかし最終的には、これらはくだんのメソッドのデフォルト実装によって送られるようになり、サブクラスの動作とは違ってしまう可能性があります。

ドキュメントを新規に作成する

ユーザがドキュメント＝ベース・アプリケーションのファイル・メニューから「新規」を選択すると、ドキュメント・アーキテクチャは新しいドキュメントを作成します。この動作は、NSDocumentControllerオブジェクト、作成されたNSDocumentオブジェクト、そして NSWindowControllerオブジェクトの間でのメッセージのやりとりで実現されます。

新しいドキュメントの作成は以下のステップで進行します。

1. ユーザがファイル・メニューから「新規」を選択するとドキュメント・コントローラに newDocument: メッセージが送られる（アップルイベントからも同じメッセージが来る）。

2. ドキュメント・コントローラは自身の openUntitledDocumentAndDisplay:error: メソッドを呼び出してデフォルトのドキュメント・タイプ（アプリケーションの info.plist に定義されている）を調べ、それをパラメータにして再び自身に makeUntitledDocumentOfType:error: メッセージを送る。

3. makeUntitledDocumentOfType:error: メソッドは受け取ったドキュメント・タイプから対応する NSDocument のサブクラスを決定し、そのインスタンスを作成して初期化メッセージを送る。

4. ドキュメント・コントローラは自身の保持するドキュメント・リストに新しいドキュメントを加え、そのドキュメントにウインドウ（これはnibファイルで定義されている）を表示するためのウインドウ・コントローラを作るようメッセージを送る。なお、このドキュメントが複数のウインドウを持つような場合には NSDocumentサブクラス の makeWindowControllers はオーバーライドされているはず。

5. ドキュメントは自身に addWindowController: メッセージを送り、新規に作成したウインドウ・コントローラをウインドウ・コントローラ・リストに加える。

6. ドキュメント・コントローラはドキュメントにウインドウを表示するようメッセージを送る。それに応えてドキュメントはウインドウ・コントローラに showWindow: を送り、そのウインドウをメイン、かつキー・ウインドウに設定する。

ドキュメントを開く

ユーザがファイル・メニューから「開く」を選択すると、ドキュメント・アーキテクチャはドキュメントをオープンし、ファイルからその内容を読み込みます。この動作は、NSDocumentController、NSOpenPanel、NSDocument、および NSWindowControllerオブジェクトによるメッセージ交換によって実現されます。

ドキュメントを開く処理とそれを新規に作成するメカニズムには多くの類似点があります。どちらの場合もドキュメント・コントローラは 適切なNSDocument サブクラスを決定して ドキュメント・オブジェクトを作りそれを初期化しなければなりませんし、また作ったドキュメントをドキュメント・リストに追加しなければなりません。そしてドキュメントがウインドウ・コントローラを作成してそれにウインドウの表示を命じなければならないのも同じです。

ドキュメント・オープンのメッセージ・フロー

既存のドキュメントをオープンする処理はいくつかの点で新規に作成する処理と異なります。ファイル・メニューの「開く」の選択によりドキュメント・オープンが呼び出されたら、ドキュメント・コントローラはまずオープン・パネルを表示してユーザにドキュメントの内容を読み込むファイルを選択させなければなりません。ドキュメント・オープンはアップルイベントによっても呼び出されますが、その場合には対象となるファイルの情報がイベントと共に送られてくるので、オープン・パネルのステップを必要としません。どちらのケースでも、ドキュメントはファイルからそのデータを読み込み、URLやドキュメント・タイプ、更新日時などのメタ情報をケアしなければなりません。

ドキュメント・オープンは以下のステップで進行します。

1. ユーザがファイル・メニューから「開く」を選択すると、ドキュメント・コントローラに openDocument: メッセージが送られる。

2. NSDocumentController オブジェクトは、ユーザにドキュメント・ファイルの場所を指示してもらうため、自身に URLsFromRunningOpenPanelメッセージを送る。このメッセージがオープン・パネルを適切にセットアップしたら、ドキュメント・コントローラは続いて自身に runModalOpenPanel:forTypes: メッセージを送り、その中で NSOpenPanel オブジェクトに runModalForTypes: を送ってユーザにオープン・パネルを提示する。

3. NSDocumentController オブジェクトは、オープン・パネルでユーザが指定したURLをパラメータとして再び自身に openDocumentWithContentsOfURL:display:error: を送る。

4. NSDocumentController オブジェクトは自身に makeDocumentWithContentsOfURL:ofType:error: を送って NSDocument オブジェクトを作成、このドキュメントに initWithContentsOfURL:ofType:error: を送って初期化を行う。このメッセージを受け取ったドキュメントは指定されたURLにあるファイルからその内容を読み込んで自身を初期化する（この初期化の過程については次項で詳述します）。

5. makeDocumentWithContentsOfURL:ofType:error: が初期化済みのNSDocument オブジェクトを返してきたら NSDocumentController オブジェクトは自身に addDocument: メッセージを送ってこれをドキュメント・リストに追加する。

6. ドキュメント・コントローラは NSDocument オブジェクトに makeWindowControllers メッセージを送ってドキュメントのユーザ・インタフェースの表示を指示。これを受けた NSDocumentオブジェクトは NSWindowControllerのインスタンスを作成し、addWindowController: を使ってこれをそのリストに加える。

7. 最後に、ドキュメント・コントローラはNSDocument オブジェクトに、これを受けた ドキュメントは NSWindowController オブジェクトに showWindows メッセージを送ってウインドウを表示し、それをメインかつキー・ウインドウに設定する。

8. もし URLsFromRunningOpenPanel メソッドが複数のURLを配列で返した場合には、１つのURLごとに3から7までを繰り返す。

ドキュメント初期化のメッセージ・フロー

NSDocument サブクラス・オブジェクトの初期化はある意味でドキュメント・アーキテクチャに関わる処理の典型といえます。既存のドキュメント・データを使って行われるドキュメントの初期化には、ドキュメントのロケーション、またはその内容であるデータを介したリード／ライト・メソッドが使われます。プログラマは少なくともそれらのうちの片方をオーバーライドしなくてはなりません。

ドキュメントの作成は以下のステップで進行します。

1. NSDocumentController オブジェクトが作成したばかりのNSDocument オブジェクトに対して initWithType:error: メッセージを送る。

2. NSDocument オブジェクトは自身に init メッセージを送り、指定された初期化メソッドを呼び出す。それから setFileType: を使ってファイル・タイプを設定する。

ファイルのオープンを伴うドキュメントの初期化以下のステップで進行します。

1. NSDocumentController オブジェクトが作成したばかりのNSDocument オブジェクトに対して initWithContentsOfURL:ofType:error: メッセージを送る。

2. NSDocumentオブジェクトは自身に init メッセージを送り、指定された初期化メソッドを呼び出す。それから 自身のメソッド、 setFileURL:、 setFileType:、 setFileModificationDate: を使ってこれから開こうとしているファイルのメタデータを設定する。

3. NSDocumentオブジェクトはファイルの内容を読み込むために、自身に readFromURL:ofType:error: を送る。このメソッドはディスクからファイル・ラッパーをゲットし、自身に readFromFileWrapper:ofType:error: メッセージを送ってそれを読む。最終的に NSDocument オブジェクトはファイルの内容をNSDataオブジェクト入れ、これを自身の readFromData:ofType:error: メソッドに渡す。NSDocument のサブクラスは、この過程に関わる３つのメソッド（readFromURL:ofType:error:,、readFromData:ofType:error:、readFromFileWrapper:ofType:error:）の少なくとも１つをオーバーライドするか、readFromURL:ofType:error: を呼び出す可能性のあるメソッド全てをオーバーライドしなければならない。

ドキュメントのセーブ

ユーザがファイル・メニューから「保存」かそれに類するコマンド、あるいは「書き出し」を選ぶと、ドキュメント・アーキテクチャはドキュメントを保存ーその内容をファイルに書くことーします。保存は基本的にはドキュメントそれ自身によって処理されます。

ドキュメントの保存は以下のステップで進行します。

1. ユーザがファイル・メニューから「保存」を選ぶと、NSDocumentオブジェクトに saveDocument: メッセージが送られる。

2. NSDocument オブジェクトは自身に saveDocumentWithDelegate:didSaveSelector:contextInfo: メッセージを送る。もしドキュメントが一度も保存されたことがないか、あるいはユーザによってファイルの場所や名前を変更されたりしていた場合、NSDocumentはそのドキュメントの保存位置をユーザに尋ねるためにモーダル・セーブ・パネルを表示させる。これは「別名で保存」あるいは「書き出し」などを選ばれたときに無条件で行うのと同じプロセス。

3. NSDocument オブジェクトはセーブ・パネルを出すために、自身に runModalSavePanelForSaveOperation:delegate:didSaveSelector:contextInfo: メッセージを送る。このメッセージはサブクラスにパネルをカスタマイズする機会を与えるために prepareSavePanel:を自身に送り、それからセーブ・パネルに runModalForDirectory:file: を送る。

4. NSDocument オブジェクトが自身に対して、２つのメッセージ saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo: と saveToURL:ofType:forSaveOperation:error: を順番に送る。

5. NSDocument オブジェクトが自身に writeSafelyToURL:ofType:forSaveOperation:error:メッセージを送る。デフォルトのインプリメンテーションでは、テンポラリ・ディレクトリを用意するか、置き換えられるファイルの名前を変更したりして書き込みが行われるときに同じ名前のファイルが存在しないようにする。次に writeToURL:ofType:forSaveOperation:originalContentsURL:error: メッセージをドキュメントに送る。

6. NSDocument オブジェクトは、自身に writeToURL:ofType:error: メッセージを送ってドキュメントの内容をファイルに書く出す。このメソッドはデフォルトで fileWrapperOfType:error: メッセージを、 fileWrapperOfType:error: は、ドキュメントの内容からNSDataオブジェクトを作成するために dataOfType:error: メッセージを自身に送る（バックワード・コンパチビリティを保つため、もし 推奨されなくなった dataRepresentationOfType: がオーバーライドされている場合にはそちらが使われる）。NSDocumentのサブクラスはそのドキュメント書き出しメソッド、 dataOfType:error:、 writeToURL:ofType:error:、 fileWrapperOfType:error:、 writeToURL:ofType:forSaveOperation:originalContentsURL:error: のうちの少なくとも１つをオーバーライドしなければならない。

7. NSDocument オブジェクトはファイルの属性などのファイルに書き込む情報を得るため自身に fileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error: メッセージを送る。それからメソッドは最終的な保存位置にファイルを移動するか、またはファイルの古いレビジョンを削除、テンポラリ・ディレクトリも削除する。その際、自身に keepBackupFile メッセージを送る。サブクラスは古いレビジョンのドキュメントを保持するため、このメソッドがYESを返すようにオーバーライドできる。

8. NSDocument オブジェクトはその位置、ファイル・タイプ、更新日時をアップデートするために setFileURL:、setFileType:、 setFileModificationDate: を自身に送る。

ウインドウを閉じる振る舞い

ドキュメント・アーキテクチャはドキュメントとそのウインドウ、およびウインドウ・コントローラに関わるメモリ管理を自動的に行います。その振る舞いは：

- ドキュメントの最後のウインドウが閉じられるとき、ドキュメントも閉じられます。すなわち、ウインドウ、ウインドウ・コントローラ、そしてドキュメントは全て解放されます。

- ドキュメントのプライマリ・ウインドウが閉じられるとき、セカンダリ・ウインドウの状態いかんに関わらず、ドキュメントも閉じられます。すなわち全てのウインドウ、ウインドウ・コントローラ、そしてドキュメント自体も解放されます。

- ドキュメントの セカンダリ・ウインドウが閉じられても、ドキュメントは閉じられません。ドキュメントは自身が保持するウインドウ・コントローラのリストから閉じられたウインドウに対応するコントローラを取り除きます。 セカンダリ・ウインドウとそのウインドウ・コントローラは解放されます。

ウインドウとウインドウ・コントローラに関する一つの考え方は、ドキュメント・コントローラはドキュメントがオープンされ、ユーザ・インタフェース（つまりは開かれているウインドウですが）を表示するためにメモリを使い、またそのメモリが確保されるのはウインドウが開かれている間だけである、というものです。ユーザがセカンダリ・ウインドウを閉じた場合、そのウインドウはもう二度と開かれない可能性もあります。よってそのウインドウ上のユーザ・インタフェースを更新するためのオーバーヘッドを被り続ける必要はありません。

対してプライマリ・ウインドウはドキュメントがオープンされている限り開かれている必要があります。プログラマはそのカスタム・ウインドウ・コントローラに setShouldCloseDocument:メッセージを送り、それがドキュメントのクローズと共に閉じられるタイプのウインドウであることを教えてプライマリ・ウインドウを定めます。

ウインドウは各々のウインドウ・コントローラと、ウインドウ・コントローラはドキュメントと、そしてドキュメントは共有のドキュメント・コントローラと協調してこの動作を実現しています。例えばあるウインドウが閉じられると、ウインドウはそのことをウインドウ・コントローラに伝えます。ウインドウ・コントローラは同じことをドキュメントに伝え、ドキュメントはウインドウ・コントローラが閉じているのを確認してそれをウインドウ・コントローラのリストから外します。この時、このリストが唯一コントローラをリテインしているので、結果としてコントローラはリリースされ、そのメモリはディアロケートされるわけです。

ウインドウ・コントローラがどのドキュメントにも管理されていない場合、そのデフォルトの振る舞いは異なります。このコントローラを管理しているドキュメントがない以上、何か他のオブジェクトがそれをリテインしていなければなりません。そしてウインドウが閉じられてもそのオブジェクトはコントローラをリテインし続け、結果としてウインドウ・コントローラはディアロケートされません。ウインドウ・コントローラがディアロケートされなければそれが管理しているウインドウも（閉じられているにも関わらず）ディアロケートされません。そしてこれがアバウト・ボックスなどのウインドウを管理するコントローラに要求される振る舞いです。

ドキュメントの管理下にないウインドウが閉じられるとき、ウインドウとそのコントローラをディアロケートしたいのであれば、そのウインドウ・コントローラをリテインしているオブジェクトにNSWindow の通知、 NSWindowWillCloseNotificationを監視するコードを付け加えるか、またはそのオブジェクトをウインドウのデリゲートにして、 windowWillClose: をインプリメントしてください。そうすればウインドウが閉じられるのを察知し、そのタイミングでウインドウとそのコントローラに autorelease メッセージを送ることができます。これはまた、ウインドウ・コントローラのnibファイルからロードされたウインドウとその他トップレベルのオブジェクトを、アプリケーションのメイン・イベント・ループが一周するタイミングでリリースするという効果を持ちます。

ドキュメントの管理下におかれるウインドウ・コントローラとそうでないものを区別するためのもう一つの考え方は、アバウト・ボックスや情報ウインドウなどのウインドウ・コントローラは通常アプリケーションの中でユニークであるのに対し、 ドキュメントに管理されるウインドウ・コントローラは複写された量産品だというものです。アプリケーションがドキュメント・オブジェクトを生成するときにはいつも（新規、あるいは既存のファイルからロードされるかに関係なく）、そのドキュメントは全く新しいウインドウ・コントローラの一そろいを要します。ドキュメントがなくなるとウインドウ・コントローラも必要なくなります。したがってアプリケーションは生成するドキュメントの数分だけウインドウ・コントローラを生成できる必要があります。対してアバウト・ボックスはアプリケーションに一個しかなく、そのためのウインドウ・コントローラも一個しか必要ありません。そのウインドウが再度使われると予想できるなら、プログラマはパフォーマンスのためメモリ上にウインドウ・コントローラを確保しておくことができます。逆にもうそのウインドウが使われないと予想できれば、メモリを節約するべくウインドウ・コントローラを削除してしまえばいいのです。ドキュメントに関わるウインドウと違い、その辺は自由です。なお、パフォーマンスに関する情報は、「Memory Usage Performance Guidelines」および、「Code Size Performance Guidelines」を参照してください。

ウインドウ・コントローラとnibファイル

プログラマがnibファイルによってウインドウ・コントローラを定義するとき、ウインドウ・コントローラはnibファイルの管理について全面的責任を負います。

＊nibファイルのロード

ウインドウ・コントローラが何かするように指示されたとき、まず最初に行うのはウインドウ・コントローラnibファイルからウインドウをロードすることです。その際、コントローラは自身をnibファイルの所有者として設定します。

＊ウインドウ・コントローラのディアロケートに伴うトップレベル・オブジェクトの解放

nibファイルの所有者として、ウインドウ・コントローラはnibファイルでインスタンシエイトされたあらゆるトップレベルのオブジェクトについて解放の責任を負います。これにはプログラマがnibファイルに追加したウインドウ以外の全てのオブジェクトが含まれます。ウインドウ・コントローラはロードの際自動的にこれらの追跡を行い、自身がディアロケートを要求されるとそれらを解放します。ウインドウ・コントローラがnibファイルからウインドウをロードして使用することができるように、nibファイルにはある種のオブジェクトが適切なアウトレット・オブジェクトと接続されて構成されている必要があります。

＊ファイルの所有者

ウインドウ・コントローラがnibファイルをロードするとき、自身をそのnibファイルの所有者として設定します。nibファイル中でウインドウ・コントローラから他のオブジェクトへの接続を可能にするために、プログラマはファイル所有者のクラスを、作成したカスタム・ウインドウ・コントローラのクラスに設定しておく必要があります（NSWindowControllerをサブクラスしない場合はこのかぎりではありません）。

＊ウインドウ・アウトレット

ウインドウ・コントローラは管理するウインドウをアウトレットとして捕捉しています。nibファイルの所有者として、ウインドウ・コントローラのウインドウ・アウトレットはそれが管理するウインドウのインスタンスに接続されていなければなりません。

＊デリゲート・アウトレット

ことさら必要ないと思われても、管理するウインドウのデリゲートとしてウインドウ・コントローラを接続しておくと便利なことがあります。nibファイルでは、管理するウインドウのデリゲート・アウトレットにファイル所有者を接続しておきましょう。

ここで説明している内容に関連して、「よくある質問」の「自動的に特定のnibファイルを使うNSWindowControllerのサブクラスを作るには？」の項も参照してください。

アプリケーションのプロパティ・リストにドキュメント・タイプを格納する

アプリケーションは情報プロパティ・リストファイルを使用します。これはindo.plistという名称でアプリケーションのバンドルに保存され、ランタイムに使用する様々な情報を指定しています。ドキュメント＝ベース・アプリケーションは、アプリケーションが編集または参照できるドキュメントのタイプをここに指定しておきます。この情報はファインダやラウンチ・サービス（Mac OS Xでアプリケーションの起動に関わるAPI）にも利用されます。

例えば、NSDocumentController オブジェクトが新規にドキュメントを作成したり、既存のドキュメントを開いたりするとき、それはまずプロパティ・リストからそのアプリケーションが編集したり開いたりすることができるドキュメント・タイプの拡張子を探します（同様にラウンチ・サービスはタイプごとのアイコンに関する情報を使うかもしれません）。プロパティ・リストのこの情報はまた、アプリケーションによるオープンやセーブ・パネルのサポートも簡便にします。

ドキュメント・タイプの情報は、 CFBundleDocumentTypes をキーとするディクショナリ・オブジェクトのアレイとして格納されています。それらプロパティ・リスト内のキーと値に関しては「Property List Reference」を参照してください。

プログラマは、Property List Editorアプリケーション（/Developer/Application/Utilities/ にあります）などを使用して、直接プロパティ・リストを作成したり編集したりすることができますが、Xcodeはドキュメント・タイプに関わる情報を（そしてその他の情報も）編集できるインタフェースを提供しています。以下の図はサンプル・プロジェクト「TextEdit」におけるターゲット情報のプロパティ・ペーンを表示したところです。ここでプログラマは ドキュメント・タイプを追加したり、登録されたドキュメント・タイプを削除、あるいは編集することができます。

注意：図のようなプロパティ・ペーンは、info.plistを使ってプロダクトを作成するターゲットにおいてのみ表示されます。JamベースのProject Builder ターゲットなどの旧来のターゲットではinfo.plistのエントリを構成することはできませんのでご注意ください。これらのターゲットをXcodeで編集するには、グループ＆ファイルのリストの中からターゲットをダブルクリックしてターゲット・エディタを起動してください。

TextEdit アプリケーションは NSDocument をサブクラスしていないので、前ページの図のクラスの列にはエントリがありません。もしアプリケーションが特定のタイプを扱う NSDocument のサブクラスを持つ場合には、同じサンプルの Sketch アプリケーション（/Developer/Examples/AppKit/Sketch）が SKTDrawDocument をそうしているようにサブクラス名をエントリしてください。

プロパティ・ペーンの上の方では、 実行可能ファイルの名称（そのままアプリケーションの名前になります）や識別子、タイプやクリエータ、バージョン など、プロダクトに関する基本情報を設定できます。なお、ここで指定するアイコンファイルの名前は、バンドルの Resources フォルダに格納されるアイコンファイル（拡張子は .icns）と一致している必要があります。

主要クラスと主要Nibファイルは、Cocoa アプリケーション、あるいはAitomatorアクションのための項目です。主要クラスはプロパティ・リストの NSPrincipalClass キーに対応し、主要Nibファイルはアプリケーションの起動時に自動的にロードされるNibファイルの指定です。こちらはプロパティ・リストの NSMainNibFile キーに対応しています。

ドキュメント・タイプ（図では「書類のタイプ」）のテーブルでは、完成したプログラムがどんなドキュメントを扱うことができるかを指定します。リストの左下にある「＋」「ー」のボタンでドキュメント・タイプの追加と削除が行えます。ユーザが新しいドキュメントを作ろうとしたとき、ドキュメント・コントローラはこのテーブルの先頭のタイプを使用します。ですからプログラマは、アプリケーションの主たるドキュメント・タイプを先頭にエントリするようにしないければなりません。以下、ドキュメント・タイプごとの各種情報について説明します。

＊名前：ドキュメント・タイプの名前です。

＊UTI：ドキュメントのタイプを特定する Uniform Type Indentifier 文字列。UTI はドキュメントのタイプを識別するユニークな文字列です。ファイル・フォーマットやデータ・タイプを特定できるだけでなく、ディクショナリやボリウム、パッケージといった他のカテゴリの情報も含めることが可能です。UTI に関する詳しい情報は、UTTypes.h を参照してください。このファイルは Mac OS X 10.3以降のシステムで、 LaunchServices.framework の一部として提供されています。

＊拡張子：ドキュメント・タイプを示すためのファイル名の拡張子。この指定に「.」を含めてはいけません。

＊MIMEタイプ：ドキュメントに対応する Multipurpose Internet Mail Extensions のタイプ。このデータはインターネットアプリケーションのためにドキュメントの中身のタイプを特定します。

＊OSタイプ：ドキュメント・タイプを示すための4レター・コード。このコードはドキュメントのリソース、あるいはプロパティ・リストファイルに格納されます。

＊クラス：ドキュメントが使用する NSDocument のサブクラス。

＊アイコンファイル：ドキュメント・タイプを表すアイコンデータを格納したファイル名

＊タイプを保存：ドキュメントの保存に使われるファイルの形式（2進、SQL、XML、メモリ内）。

＊役割：アプリケーションがこのタイプのドキュメントをどう使用するか（以下の4つのうち１つ）。

□エディタ：アプリケーションはこのドキュメントを開き、編集し、結果を保存できる。
□ビューア：アプリケーションはこのドキュメントを開ける、が変更はできない。
□シェル：アプリケーションは他のプロセス、例えばJavaアプレットなどにランタイム・サービスを提供する。
□なし：アプリケーションはこのドキュメントを表示も編集もしないが、他の方法で使用する。例えばSketchは、書き出し（Export）はできるが読み込めないタイプにこの役割を設定している。

＊パッケージ：そのドキュメントが単一のファイルなのかパッケージ（実はディレクトリだがファインダのようなアプリケーションには単一のファイルと扱われるもの）なのかの別。

ドキュメント・タイプに関する情報を編集するには、テーブルの中のタイプをクリックして選択し、編集したい項目をダブルクリックします。

以下のリストはプロパティ・リストファイルのこのテーブルにあたる部分をテキスト・エディタで開いたものです。なお、このリストはアップル・アプリケーションでのみ使われるいくつかの項目を省略しています。また LSTypeIsPackage の項目は、True であればこのドキュメントがパッケージであることを示します。

```
<key>CFBundleDocumentTypes</key>
	<array>
		<dict>
			<key>CFBundleTypeExtensions</key>
			<array>
				<string>rtf</string>
			</array>
			<key>CFBundleTypeIconFile</key>
			<string>rtf.icns</string>
			<key>CFBundleTypeMIMETypes</key>
			<array>
				<string>text/rtf</string>
			</array>
			<key>CFBundleTypeName</key>
			<string>NSRTFPboardType</string>
			<key>CFBundleTypeOSTypes</key>
			<array>
				<string>RTF </string>
			</array>
			<key>CFBundleTypeRole</key>
			<string>Editor</string>
			<key>LSIsAppleDefaultForType</key>
			<true/>
		</dict>
		<dict>
			<key>CFBundleTypeExtensions</key>
			<array>
				<string>doc</string>
```

```
        </array>
        <key>CFBundleTypeName</key>
        <string>Microsoft Word Document</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>W8BN</string>
            <string>W6BN</string>
        </array>
        <key>CFBundleTypeRole</key>
        <string>Viewer</string>
    </dict>
    <dict>
        <key>CFBundleTypeExtensions</key>
        <array>
            <string>rtfd</string>
        </array>
        <key>CFBundleTypeIconFile</key>
        <string>rtfd.icns</string>
        <key>CFBundleTypeName</key>
        <string>NSRTFDPboardType</string>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>LSIsAppleDefaultForType</key>
        <true/>
        <key>LSTypeIsPackage</key>
        <true/>
    </dict>
    <dict>
        <key>CFBundleTypeExtensions</key>
        <array>
            <string>txt</string>
            <string>text</string>
            <string>*</string>
        </array>
        <key>CFBundleTypeIconFile</key>
        <string>txt.icns</string>
        <key>CFBundleTypeMIMETypes</key>
        <array>
            <string>text/plain</string>
        </array>
        <key>CFBundleTypeName</key>
        <string>NSStringPboardType</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>****</string>
        </array>
        <key>CFBundleTypeRole</key>
```

```
        <string>Editor</string>
        <key>LSIsAppleDefaultForType</key>
        <true/>
</dict>
<dict>
        <key>CFBundleTypeIconFile</key>
        <string>txt.icns</string>
        <key>CFBundleTypeName</key>
        <string>Apple SimpleText document</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
                <string>TEXT</string>
                <string>sEXT</string>
                <string>ttro</string>
        </array>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>LSIsAppleDefaultForType</key>
        <true/>
</dict>
<dict>
        <key>CFBundleTypeExtensions</key>
        <array>
                <string>html</string>
                <string>htm</string>
        </array>
        <key>CFBundleTypeIconFile</key>
        <string>html.icns</string>
        <key>CFBundleTypeName</key>
        <string>Apple HTML document</string>
        <key>CFBundleTypeRole</key>
        <string>Viewer</string>
</dict>
<dict>
        <key>CFBundleTypeExtensions</key>
        <array>
                <string>webarchive</string>
        </array>
        <key>CFBundleTypeName</key>
        <string>Apple Web archive</string>
        <key>CFBundleTypeRole</key>
        <string>Viewer</string>
</dict>
```

このプロパティ・リストはTextEdit アプリケーションが多くのドキュメント・タイプ（rtf、rtfd、textの他、Apple SimpleText、MicrosoftWord、Apple HTML、AppleWeb archiveなど）をサポートしていることを示しています。またオープン・パネル、セーブ・パネルにおいてフィルタとして使われる拡張子を指定します（各タイプの最初の拡張子は自動的にセーブ・パネルの「名前」の欄に表示されるファイル名に付加されます）。なお、TextEditは それらを編集することができますが、MicrosoftWord、HTML、web archive タイプに対して CFBundleTypeRole をビューアに指定しています。これはTextEdit がこれらのタイプのプライマリなアプリケーションではないことを表すため、つまりラウンチ・サービスが「.doc」ファイルのダブルクリックに対して MicrosoftWordアプリケーションではなくTextEditを起動してしまわないようにするためです。

TextEdit アプリケーション はドキュメント＝ベース・アプリケーションでありながらNSDocument のサブクラスを持たない珍しい例です。そのため、前掲のディクショナリにはドキュメント・クラス名のエントリがありません。が、通常のドキュメント＝ベース・アプリケーションはNSDocumentをサブクラスして特定のドキュメント・タイプを開いたり編集したりするので、そのクラス名をプロパティ・リストにエントリしなければなりません。以下にそれを行っている例として Sketch アプリケーションのプロパティ・リストファイルの該当部分を示します。最初のエントリ、 SKTDrawDocument はこのアプリケーションの主要なドキュメント・タイプです。

ドキュメントのオープン時、NSDocumentControllerオブジェクトはデータ・タイプに対応した NSDocument サブクラスのインスタンスを生成するためにプロパティ・リストの情報を使います。したがってプロパティ・リストでドキュメント・クラスの指定をしておけば、プログラム中で明示的にサブクラスを割り当てて初期化しなくてもドキュメント・コントローラが自動的にそれを行ってくれます。

```
<key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>sketch</string>
                <string>draw2</string>
            </array>
            <key>CFBundleTypeIconFile</key>
            <string>Draw2File</string>
            <key>CFBundleTypeName</key>
            <string>Apple Sketch Graphic Format</string>
            <key>CFBundleTypeRole</key>
            <string>Editor</string>
            <key>NSDocumentClass</key>
            <string>SKTDrawDocument</string>
            <key>NSExportableAs</key>
            <array>
                <string>NSPDFPboardType</string>
                <string>NSTIFFPboardType</string>
            </array>
        </dict>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
```

```
                <string>pdf</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSPDFPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>tiff</string>
                <string>tif</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSTIFFPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
    </array>
```

ドキュメント・タイプを定義するためには、CFBundleTypeExtensions、CFBundleTypeName、そしてCFBundleTypeRole の3つのキーが最低限必要です。上のリストの2番目、3番目のエントリがこれら必須のキーのみでドキュメント・タイプを定義しています。役割に対する「なし」の指定は、通常アプリケーションがそのドキュメント・データを理解しないという意味ですが、ファインダがフォントにアイコンを付与しているようにそのタイプに対する情報を定義するだけ、という場合にも使われます。上のケースでは、Sketch が書き出すことはできるが読み込めないタイプの拡張子を役割「なし」で定義しています。ここで、Sketch のプライマリ・タイプ（テーブルの最初のエントリ）がこれらのタイプを NSExportableAs キーの元に宣言しているのは、Sketch がプライマリ・タイプのドキュメントの内容をこれらのフォーマットに書き出すことができることを示します。ユーザは「別名で保存」の操作の際に、これらのフォーマットへの書き出しを選択することができます。

HFSタイプとクリエータ・コードの保存

HFS（Hierarchical file system）タイプとクリエータ・コードはファインダやラウンチ・サービスによって、ドキュメント・ファイルをアプリケーションやアイコンなどと関連付けるために使われます。けれどもNSDocument ベースのアプリケーションは、デフォルトではHFSタイプもクリエータ・コードもドキュメント・ファイルに保存しません。これらをセットするためには、NSDocumentのサブクラスで、fileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error: をオーバーライドします。そして NSFileHFSTypeCode、 NSFileHFSCreatorCode 属性の値として各々のコードを設定します。

NSDocument に関係ないところでファイルにHFSタイプやクリエータ・コードを設定したい場合には、NSFileManager のメソッド、 changeFileAttributes:atPath: を使います。

以下のサンプルコードは、NSDocument サブクラスにおけるfileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error: メソッドのオーバーライドの例です。この実装は、このコード以前に以下のように kMyAppCreatorCode が定義されていることを前提に書かれています。

```
const OSType kMyAppCreatorCode = 'Blah';
```

なお、このコードはHFSタイプとクリエータ・コードを書き換えるだけでそのまま使用することができます。

```
- (NSDictionary *)fileAttributesToWriteToURL:(NSURL *)absoluteURL
        ofType:(NSString *)typeName
        forSaveOperation:(NSSaveOperationType)saveOperation
        originalContentsURL:(NSURL *)absoluteOriginalContentsURL
        error:(NSError **)outError {
    NSMutableDictionary *fileAttributes =
        [[super fileAttributesToWriteToURL:absoluteURL
            ofType:typeName forSaveOperation:saveOperation
            originalContentsURL:absoluteOriginalContentsURL
            error:outError] mutableCopy];
    [fileAttributes setObject:[NSNumber
        numberWithUnsignedInt:kMyAppCreatorCode]
            forKey:NSFileHFSCreatorCode];
    [fileAttributes setObject:[NSNumber
        numberWithUnsignedInt:kMyDocumentTypeCode]
            forKey:NSFileHFSTypeCode];
    return [fileAttributes autorelease];
}
```

マルチ・ドキュメント・タイプのアプリケーションを作る

ドキュメント・アーキテクチャは複数のNSDocumentのサブクラスで、異なるドキュメント・タイプを操作できるアプリケーションのためのサポートを提供しています。例えば、AppleWorks では一つのアプリケーションでテキスト・ドキュメント、スプレッドシートその他のタイプのドキュメントを作成することができます。それら異なったドキュメント・タイプは、それぞれユニークなNSDocumentサブクラスにカプセル化された異なるユーザ・インタフェースを求められます。１つのドキュメント＝ベース・アプリケーションにこのような複数のサブクラスを統合するためには、以下に記載するように、nibファイル、info.plistファイル、そしてドキュメント・コントローラを設定します。

まず、Xcode でドキュメント＝ベース・アプリケーションを作成するためにXcode の新規プロジェクト・パネルから、Cocoa Document-based Application のテンプレートを選びます。Xcodeはドキュメントを表示するためのウインドウおよびNSDocumentサブクラスが定義されたnibファイルをひとつ用意します。これに新たなサブクラスのnibファイルを追加するには、それを手動で行うことになります。

新規にNSDocumentサブクラスを作成するには、15ページの「NSDocumentのサブクラスを作成する」で説明されているように、まずプリミティブなリード／ライト・メソッドを用意しなければなりません。

ドキュメント・ウインドウのユーザ・インタフェースをデザインし、それをnibファイルに収めるには Interface Builder を使います。次にNSDocumentサブクラスの windowNibName メソッドをオーバーライドし、このnibファイルの名前を返すようにします。

Interface Builder で新規に作成されたnibファイルではファイルの所有者が自動的にNSDocumentのサブクラスにならず、単なるNSObjectに設定されています。したがってプログラマはnibファイルの所有者を自分でNSDocument サブクラス、またはNSWindowController をサブクラスするのであればそのサブクラスに設定し直す必要があります。そして、ファイル所有者のアウトレットに作成したウインドウを接続します。これを行わないと、NSDocument サブクラスのインスタンスが生成されてもウインドウは表示されません。最後にファイル所有者をウインドウのデリゲートに設定してください。

またプログラマは、28ページ「 アプリケーションのプロパティ・リストにドキュメント・タイプを格納する 」で説明されている通り、info.plist の CFBundleDocumentTypesキーにこのサブクラスの情報を追加しなければなりません。

もしそのアプリケーションが既存のドキュメント開くだけなら、開かれるときにそのドキュメントのタイプが決定していることになるので、Application Kit によって自動的に作成されるデフォルトのドキュメント・コントローラを使用することができます。しかしながらアプリケーションが複数のタイプのファイルを新規に作成できるとすれば、そのタイプを選ぶコードを追加するために NSDocumentController をサブクラスしなければなりません。

NSDocumentController のアクションメソッド、newDocument: は Info.plist に定義されているアプリケーションが扱えるドキュメント・タイプのアレイの先頭のタイプを使って新しいドキュメントを生成します。これではこのアプリケーションがサポートする他のタイプのドキュメントを新規に作成することはできないことになります。何らかの方法で作成するドキュメントのタイプを決定するか、その決定をユーザにゆだねるインタフェースを実装しなければなりません。

プログラマはアプリケーションのデリゲート、あるいはNSDocumentControllerのサブクラスに独自のアクションメソッドを作り込むことでこの問題を解決できます。例えば、ファイル・メニューの「新規」にサブメニューを設けて作成するドキュメントのタイプを限定する、または「新規」を選択されたあとでユーザに作成したいドキュメントのタイプを尋ねるようなインタフェースを表示するなどが考えられます。

ユーザがタイプを決定したら、アクションメソッドはNSDocumentCotroller のメソッド、makeUntitledDocumentOfType:error: にそのタイプを指定して新しいドキュメント・オブジェクトを生成します。無事にドキュメントが作成できたらそれをドキュメント・コントローラのリストに追加し、また20ページ「 ドキュメント・アーキテクチャのメッセージ・フロー 」で説明したように、そのドキュメントに makeWindowControllers と showWindows メッセージを送ります。

また、NSDocuemtnController サブクラスで defaultType メソッドをオーバーライドし、ユーザが「新規」を選んだときに作成されるデフォルトのドキュメント・タイプを操作するという方法もあります。

ドキュメント・アーキテクチャにおけるオートセーブ

Mac OS X 10.4 以降のシステムでは、ドキュメント＝ベース・アプリケーション・アーキテクチャはドキュメントのオートセーブをサポートしています。これは、NSDocumentおよび NSDocumentControllerの呼び出したり、カスタマイズするためにオーバーライドできるメソッドとして提供されます。

オートセーブの振る舞い

オートセーブは、ユーザによる編集中一定の時間間隔で、ドキュメント・アーキテクチャが自動的にドキュメントをディスクに保存するというメカニズムです。この機能はユーザのデータを停電やアプリケーション・クラッシュなどから守ります。デフォルトではオートセーブ機構は作動していませんが、アプリケーションが NSDocumentController にたった１つ、メッセージを送るだけで簡単に作動させることができます。

ドキュメント＝ベース・アプリケーションでオートセーブを作動するには、 NSDocumentController オブジェクトに 0より大きい数（オートセーブのインターバル）をパラーメータにして setAutosavingDelay: メッセージを送るだけでOKです。また、 NSDocumentとNSDocumentControllerのオートセーブ関連のメソッドをオーバーライドしてその振る舞いをカスタマイズすることもできます。

オートセーブが作動中のアプリケーションでは、新しいドキュメントがユーザによって保存されるまでドキュメント・アーキテクチャがその内容を、デフォルトでは /Library/Autosave information フォルダに保存し続けます。そしてユーザがドキュメントに名前を付けて保存を行うと、それ以降のオートセーブはユーザがそのドキュメントを保存したフォルダで行われ、アプリケーションの正常終了とともに削除されます。

オートセーブが作動中のアプリケーションがクラッシュするなどの異常終了に見舞われると、ドキュメント・アーキテクチャはその直前にオートセーブされたドキュメントをリテインします。また、アプリケーションが再起動されるとオートセーブも自動的に再開されます。

NSDocumentController のオートセーブ・メソッド

NSDocumentController にはオートセーブに関連する4つのメソッドがあります。その1つはそのインターバルを設定してオートセーブを作動させるものです。これらのメソッドを呼び出したりオーバーライドすることで、オートセーブの振る舞いをカスタマイズすることができます。2つのメソッドがドキュメントがいつオートセーブされるのかに関連し、他の2つがアプリケーションの起動時、オートセーブされたドキュメントが再びオープンされるときに行われる動作に関連しています。

ドキュメントがいつオートセーブされるのか（または全くされないのか）をコントロールするメソッドが保存のインターバル時間を秒単位で設定する setAutosavingDelay: メソッドです。ここで指定されるインターバルは、ドキュメント・コントローラがユーザによるドキュメントの改変を感知してからドキュメントに対して autosaveDocumentWithDelegate:didAutosaveSelector:contextInfo: メッセージを送るまでに待つ時間です。０の指定はオートセーブをしないことを意味します（デフォルト値が0なので、オートセーブはデフォルトでオフなわけです）。メソッド autosavingDelay は現在設定されているインターバル時間を返します。

オートセーブ・ドキュメントが再オープンされる際の振る舞いをカスタマイズするには、 その再オープンを行うために NSDocumentController が呼び出すメソッド、reopenDocumentForURL:withContentsOfURL:error: か、あるいは その際にインスタンスを作るクラスを決定し、ドキュメント・オブジェクトをアロケートし、initForURL:withContentsOfURL:ofType:error: メッセージをそのオブジェクトに送る、 makeDocumentForURL:withContentsOfURL:ofType:error: メソッドをオーバーライドします。

NSDocument のオートセーブ・メソッド

NSDocument にもオートセーブに関連するメソッドがあります。これらのメソッドはオートセーブを有効にしたりそのインターバルを設定したりするものではありませんが、呼び出したりあるいはオーバーライドしたりすることでオートセーブの振る舞いをカスタマイズできます。

initForURL:withContentsOfURL:ofType:error: メソッドは、オートセーブされたドキュメントが再オープンされるときに実際の初期化を行います。このメソッドは指定されたURLのドキュメントを初期化しますが、内容は別の場所、オートセーブされたファイルの位置から読み込みます。

プログラマは setAutosavedContentsFileURL: メソッドを使って、ドキュメントがオートセーブされる位置を任意に設定できます。また、 autosavedContentsFileURL メソッドを使えば現在のセーブ位置を参照することができます。

autosaveDocumentWithDelegate:didAutosaveSelector:contextInfo: メソッドはドキュメントをオートセーブします。デフォルトの実装ではドキュメントがオートセーブされる必要があるかどうかを判断し、必要があれば saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo: メッセージをドキュメントに送ってセーブを実行させます。プログラマがこのメソッドや、その他URLパラメータを受け取ってディスクに書き込みを行う NSDocument のメソッドをオーバーライドする場合には、必ずURLパラメータの指定している位置にファイル書くよう注意しなければなりません。そのパラメータのURLはオートセーブの機構によって変更されており、 メソッド fileURL が返す値とは違っている可能性があります。

プログラマはドキュメントに hasUnautosavedChanges メッセージを送ってそのドキュメントのオートセーブ状態を確認することができます。ドキュメントに加えられた変更がまだオートセーブされていない場合、このメッセージは YES を返します。

autosavingFileType メソッドは、オートセーブに使用されるべきドキュメントのタイプを返します。プログラマはこのメソッドを nil を返すようにオーバーライドすることで、個々のドキュメントのオートセーブを完全に無効にすることができます。またこのメソッドが特殊なタイプ、例えばそのアプリケーションがオートセーブのためだけに定義した特別なタイプなどを返すようにオーバーライドして、完全なドキュメントの代わりにその変更部分だけを効率的に記録するような仕組みをつくることも可能です。

ドキュメント・アーキテクチャのエラー処理

Mac OS X10.4 以降のシステムでは、ドキュメント・アーキテクチャのエラー処理が大幅に改善されました。

Mac OS X 10.4 でNSDocument および NSDocumentController に追加されたメソッドの多くが、最後のパラメータとして NSError オブジェクトへの間接参照を含んでいます。これらのメソッドはドキュメントを作成したりファイルを書いたりリソースにアクセスしたり、それらに類する働きをするものです。多くの場合、それらのメソッドは推奨されなくなったエラー・アーギュメントを持たないメソッドを置き換えるものです。

NSDocumentController のエラー・パラメータを持つメソッドの例を2つ挙げましょう。1つは新規に名称未設定のドキュメントを作成する openUntitledDocumentAndDisplay:error:、もう1つは指定されたURLのドキュメントをオープンする openDocumentWithContentsOfURL:display:error: です。オペレーションが失敗した場合、これらのメソッドは戻り値として nil を返し、最後のパラメータに 起きたエラーについての情報を含む NSError オブジェクトを返します。この情報が必要な場合、NSError オブジェクトを変数として宣言し、これらのメソッドの最後のパラメータにそのアドレスをパスします。必要がなければ、このパラメータにはNULLを渡します。

NSError オブジェクトを使うことで、Cocoa アプリケーションは従来よりはるかに詳しく役に立つエラーメッセージをユーザに提供できるようになります。そこにはエラーが起きた詳細な状況、回復に向けてのサジェスチョン、そしてプログラマティックな復帰に向けてのメカニズムさえ含まれます。なお、ユーザへのエラー表示は Application Kit が行います。

NSError を使ったエラー処理の詳細については「Error Handling Programming Guide For Cocoa」を参照してください。

ドキュメント・アーキテクチャで NSError によるエラー処理を利用するには readFromURL:ofType: のような推奨されなくなったメソッドの代わりに、エラー・パラメータを持つ readFromURL:ofType:error: のようなメソッドをオーバーライドしてください。また、プログラムコードの中にNSDocument、NSDocumentController のそうした古いメソッドをオーバーライドしたものがあればそれを削除します。ドキュメント・アーキテクチャはバックワード・コンパチビリティを保証するため、古いメソッドのエントリが存在する場合にはそちらを使ってしまうので、こういうコードが残っているとNSError のエラー処理が使用されません。

重要：Cocoa エラードメインに含まれ、エラー・パラメータを持つ Cocoaメソッドは NSError オブジェクトを返すことを保証されます。従ってそうしたメソッドをオーバーライドする際には必ず以下の要件を守ってください。すなわち「error:(NSError**)outError」というアーギュメントを取るメソッドはエラーが発生したとき（メソッド自体の戻り値が nil または NOのとき）にNSError オブジェクトを受け取るため、*outError（outErrorの内容）としてNSErrorオブジェクトへのポインタを渡さなければなりません。

これらのメソッドをオーバーライドしていて、ユーザに対してエラー・アラートを表示したくない場合には、ドメインにNSCocoaErrorDomain、コードに NSUserCancelledError をセットしたNSError オブジェクトを戻してください。Application kit はNSApplication の presentError: および NSResponder で宣言される presentError:modalForWindow:delegate:didPresentSelector:contextInfo:methods: を通してエラーを表示します。これらのインプリメンテーションは、ドメインがNSCocoaErrorDomain、かつコードが NSUserCancelledError のNSError オブジェクトを無視します。ですから例えば openDocumentWithContentsOfURL:display:error: のオーバーライドがユーザに向けてのエラー・アラートの表示を避けたい場合には、以下のコードのようにします。

```
if (outError) {
    *outError = [NSError errorWithDomain:NSCocoaErrorDomain
                                    code:NSUserCancelledError userInfo:nil];
}
```

Cocoa のメモリ管理ルールは、このメソッドの呼び出し側に NSError オブジェクトの解放を義務づけていません。上の errorWithDomain:code:userInfo: はオートリリースされるオブジェクトを返します。また、もしオーバーライドがこれらのメソッドのスーパクラスのインプリメンテーションを呼ぶ場合は、それをただパスしてください。自分で outError を設定したりする必要はありません。

よくある質問

このセクションでは、Application Kit のドキュメント・ハンドリング・クラスに関するよくある質問とその答えを紹介します。Application Kit のドキュメント・ハンドリング・クラスとは、主に NSDocument、そして、NSDocumentController、NSWindowControllerを含みます。

何から始めれば？

NSDocument ベースのアプリケーションの開発を始めるには、Xcode で新規プロジェクトを作成する際に「Cocoa Document-based Application」テンプレートを選択します。 こうすればXcodeは NSDocument のサブクラスとドキュメントnibファイルを含んだアプリケーション・プロジェクトを用意します。

NSDocument サブクラスはドキュメントnibファイルをロードするためにプリセットされています。このサブクラスにはプログラマが実装しなければならないロードとセーブのための空のメソッドのほか、nibファイルがロードされたあとで呼ばれるメソッドが定義されています。

この状態に何一つコードを書き加えずに一度これらのコードをコンパイルし、アプリケーションを実行してみてください。アプリケーションが起動すると空の、Untitled という名前のウインドウが表示されます。ファイル・メニューのコマンドは例えば「開く」を選ぶとオープン・パネルが、「保存」を選ぶとセーブ・パネルが表示されるといったように、全てなにかそれらしいことを行います。もっともまだ何のタイプも定義されていませんし、ロードとセーブに関わるプログラムも実装されていないので、実際には何も開くことはできないし、保存することもできません。

ここをスタートとして、まずはドキュメント・タイプを定義し（次項「タイプを定義するには？」参照）、ロードやセーブのためのメソッドを実装（次ページ「ファイルのロード、セーブを実装するには？」参照）します。

それらの実装が済んだら、そのドキュメント・サブクラスの他の機能の実装に入ります。ドキュメント・サブクラスはドキュメントの内容を保持し、管理できなければなりません。それを行うためのメソッドを実装します。

ドキュメント＝ベース・アプリケーションの作成に関わる詳細は8ページの「 ドキュメント＝ベース・アプリケーションの実装」を、NSDocument サブクラスについての詳細は15ページ「NSDocumentのサブクラスを作成する」を参照してください。またドキュメント＝ベース・アプリケーションのサンプル、Sketch が /Developer/Examples/AppKit/Sketch にありますからこれも参考にしてください。

タイプを定義するには？

タイプはXcodeによって管理される info.plist というファイルで定義されます。このファイルの内容であるプロパティ・リストについては28ページ「 アプリケーションのプロパティ・リストにドキュメント・タイプを格納する」および、「Property List Programming Guide for Cocoa」を参照してください。

プログラマはXcode のターゲット・インスペクタでアプリケーションで使用するタイプを定義することができます。ターゲット・インスペクタは「 アプリケーションのプロパティ・リストにドキュメント・タイプを格納する」で説明されているように、ドキュメント・タイプを編集できるユーザ・インタフェースを提供します。

プログラマはこれから作成するアプリケーションのために理解しやすい名称と拡張子でタイプを定義します。なお、タイプは複数定義することができます。

アプリケーションのメインの、あるいは最も重要なドキュメント・タイプをこのテーブルの先頭に記載してください。ユーザが新しいドキュメントを作成する際に、NSDocumentControllerはこのテーブルの先頭のタイプを使用します。

ファイルのロード、セーブを実装するには？

新規に作成されたドキュメント＝ベース・アプリケーション・プロジェクトには自動的に作成されるNSDocumentのサブクラスに、 dataRepresentationOfType: 、 loadDataRepresentation:ofType: というメソッドの空の実装が定義されています。もし作成するアプリケーションが Mac OS X 10.3、あるいはそれ以前のシステムで稼働しなければならない場合には、単純なファイルの読み書きをサポートするためにこれらのメソッドを実装してください。

Mac OS X 10.4以降を必要とするアプリケーションの場合は、 これらの代わりにdataOfType:error: 、readFromData:ofType:error: の両メソッドをオーバーライドします。これらのオーバーライドの実装については 8ページ「 ドキュメント＝ベース・アプリケーションの実装」や、サンプル・アプリケーション Sketch（/Developer/Examples/AppKit/Sketch）を参照してください。

dataOfType:error: メソッドは要求されたタイプのドキュメントの内容を NSData オブジェクトの形式で返さなければなりません。また、 readFromData:ofType:error: メソッドは NSData オブジェクトを与えられたタイプのドキュメントとして読み込みます。

もしアプリケーションがファイル・ラッパーとしてファイルを保存するなど、なにか特別なことをする必要がある場合には、次項「ドキュメント・パッケージを実装するには？」、あるいは次ページ「ファイル・ラッパー以外の特殊なロード／セーブ・メソッドを実装するには？」を参照してください。

ドキュメント・パッケージを実装するには？

ドキュメント・パッケージは、本当はフォルダなのにファインダ上では1個のドキュメント・ファイルとして表示されます。通常、扱うドキュメントがこのパッケージである必要があるアプリケーションでは、これを構築しアクセスするために NSFileWrapper のクラスを使用します。作成するアプリケーションで扱うドキュメントがこれに当たる場合、プログラマは dataOfType:error: と readFromData:ofType:error: をオーバーライドする代わりに fileWrapperOfType:error: と readFromFileWrapper:ofType:error: をオーバーライドする必要があります。

これらのメソッドのすることは NSDataベースのメソッドと対して違いませんが、NSData オブジェクトの代わりに NSFileWapperオブジェクトを使います。

また、この場合、プロパティ・リストのドキュメント・タイプのセクションに、そのタイプがファイル・ラッパーであることを記載しておく必要があります。詳しくは28ページ「 アプリケーションのプロパティ・リストにドキュメント・タイプを格納する」を参照してください。

ファイル・ラッパー以外の特殊なロード／セーブ・メソッドを実装するには？

なんらかの理由でNSData ベース、NSFileWrapper ベース、どちらのメソッドを使ってもアプリケーションに要求される仕様を満足できない場合、プログラマはURLによる位置指定を介してドキュメントのロード／セーブを行うNSDocumentのメソッドをオーバーライドすることができます。そのメソッドは以下の3つです。

readFromURL:ofType:error:
writeToURL:ofType:error:
writeToURL:ofType:forSaveOperation:originalContentsURL:error:

上の読み込みメソッドと書き出しメソッドのうちの片方をオーバーライドしてください。

オーバーライドの目的がロードやセーブの処理そのもののカスタマイズではなく、ロードやセーブの直前、あるいは直後になんらかの処理を実行することである場合があります。そのような場合、オーバーライド・メソッドはスーパークラスの実装を呼び出す直前や直後に目的の処理を行うように書かれます。46ページの「読み込んだタイプを自動的に変換するには？」でこのタイプのオーバーライドについて説明していますのでお読みください。

NSWindowControllerをサブクラスするべきなのは？

デフォルトのドキュメント＝ベース・アプリケーション・プロジェクトのテンプレートではNSWindowControllerをサブクラスしていません。作成するのが簡単なアプリケーション（テンプレートは最も簡単なアプリケーションとも言えます）であれば、NSWindowControllerをサブクラスする必要は生じないでしょう。けれども、より高度なアプリケーションを作るとなると話は違ってきます。プログラマはほとんどの場合、その必要を感じるはずです。以下にNSWindowControllerをサブクラスするのが望ましい状況をいくつか例示してみます。

＊ドキュメントを表示するのに複数のウインドウを必要とする場合：ドキュメントの内容が複雑で、それを表示するのに複数のウインドウを必要とするアプリケーション（例えば3方向からの図を表示するCADプログラムや3Dプログラムなど）では、異なる複数のウインドウを管理するためにNSWindowController のサブクラスが必要になります。

＊ドキュメントに対して複数のビューをサポートしたい場合：ユーザが一つのドキュメント上に複数のビューを作成できるようにしたい場合（例えば一つのドキュメント上に複数のビューを配置して別々の指定で描画が行えるドローイング・プログラムなど）には、NSWindowController のサブクラスが必要になります。

＊アプリケーションのコントローラ・レイヤーがそれをフロントエンド（NSWindowsControllerのサブクラス）とバックエンド（NSDocumentのサブクラス）に分けた方がいいと思えるほど複雑な場合：特に大規模なアプリケーションの場合、これら2つのクラスの役割を分割することは大きな意味があります。この方法は、なんらかの事情で使われない可能性のあるメモリその他を節約するため、開かれているけれどもスクリーンに表示されていないドキュメントという存在を許容します。

ドキュメント・アーキテクチャにおけるNSDocument と NSWindowControllerの役割については、4ページの「 ドキュメント＝ベース・アプリケーションにおけるキー・オブジェクトの役割」を参照してください。

NSWindowControllerをサブクラスする方法は？

いったんNSWindowControllerをサブクラスすると決めたら、ドキュメント＝ベース・アプリケーションのデフォルトの設定にいくつかの変更を加える必要があります。まず最初に、Interface Builderで定義されるアウトレットやアクションをNSDocument のサブクラスの代わりに NSWindowController のサブクラスに接続するようにします。これはNSWindowControllerサブクラスがnibファイルの所有者になるからです。ただしメニュー・アクションのいくつか、例えば「保存」や、「最後に保存した状態に戻す」などは、変わらず NSDocumentに実行させることになります。また、ドキュメントに新しいメニュー・アクション、例えばドキュメントに新しいビューを加えるなど、を追加することもできます。

次に、NSDocumentのサブクラスで windowNibName メソッドをオーバーライドする代わりにmakeWindowControllersをオーバーライドします。このメソッドで、NSWindowControllerサブクラスのインスタンスを少なくとも1つ作成し、 addWindowController: を使ってこれをドキュメントに追加します。またドキュメントが常に複数のウインドウ・コントローラを必要とする場合にはこのタイミングでその全てを作成します。ドキュメントに複数のビューをサポートしたい場合、デフォルトかつ最初の1つのためのコントローラをここで作成します。そして、新たなビューを追加するためのユーザ・アクションを用意します。

なお、makeWindowControllers メソッドの中で強制的にウインドウをビジブルにするべきではありません。その必要があればNSDocumentは自動的にそうします。

NSWindowController サブクラスに関するさらなる詳細は47ページの「特定のnibファイルを自動的に使うNSWIndowControllerサブクラスを作るには？」を参照してください。

ユーザ・インタフェース・オブジェクトをセットアップするのはいつ？

多くのアプリケーションが、アプリケーションのモデルデータがロードされた後で、ビューの内容を準備するなどユーザ・インタフェース・オブジェクトのセットアップを行う必要があります。このケースで、NSDocumentの readFromData:ofType:error: などのデータを読むためのメソッドは、ドキュメントのユーザ・インタフェース・オブジェクトがnibファイルからロードされる前にコールされるということを忘れてはいけません。当たり前ですがまだロードされていないユーザ・インタフェース・オブジェクトにメッセージを送って何かを実行させることはできません。

データのロードが済んだ後、プログラムはそれを一時棚上げしてドキュメントnibファイルからのユーザ・インタフェース・オブジェクトのロードを待ってそれをセットアップする必要があります。アプリケーションがNSWindowControllerをサブクラスしていない場合は、NSDocumentのメソッド windowControllerDidLoadNib: を代わりにオーバーライドして、またNSWindowControllerをサブクラスしている場合には、そのメソッド windowDidLoad をオーバーライドします。

これとは逆に、nibファルがロードされる直前になんらかの操作を行う必要がある場合には、NSDocumentの windowControllerWillLoadNib: メソッドをオーバーライドすることが可能です。こちらの場合もNSWindowControllerをサブクラスしていれば、 windowWillLoad を代わりに使うことになります。

nibファイルでインスタンシエイトされたオブジェクトに対しては、 awakeFromNib メソッドを実装することでセットアップを行えます。Application Kit は nibファイルから全てのオブジェクトがロードされ、それらの接続全てがセットアップされたあとで個々のオブジェクトに対してこのメッセージを送ります。ただし、送られる順番は決まっていません。

リード・オンリー・タイプをサポートするには？

アプリケーションに、読み込むことはできるけれども書き出すことのできないタイプがある場合、プロパティ・リストのドキュメント・タイプのテーブルの役割の項目に、「エディタ」の代わりに「ビューア」を設定します。詳しくは28ページの「 アプリケーションのプロパティ・リストにドキュメント・タイプを格納する 」を参照してください。

ライト・オンリー・タイプをサポートするには？

前項とは逆に、アプリケーションに書き出すことはできるけれども読み込むことのできないタイプがある場合、NSExportableAs キーを使ってそれを宣言することができます。プロパティ・リストのドキュメントのエントリ（普通はプライマリ・ドキュメントのエントリです）に、このキーをつかって書き出せるが読み込めないファイルのタイプのアレイをセットします。

サンプル・アプリケーション Sketch ではこのタイプとしてTIFFとEPSを宣言しています。参考にしてください。

これらライト・オンリー・タイプは「別名で保存」の処理の際に選択可能になります。「保存」の場合には選択できません。

読み込んだタイプを自動的に変換するには？

アプリケーションには、あるタイプの書き出し方は判らないけれども読み込むことはできるというケースがあります。そのような場合、ドキュメントを読み込むと同時に書き出すこともできるタイプに変換してしまうべきです。良い例が自身の古いバージョンで作成されたドキュメントや、競合製品のドキュメントを読むことができるアプリケーションです。こうしたアプリケーションではドキュメントを読み込むと同時に現在のバージョンのネイティブのタイプに変換を行います。

これを可能にする最初のステップは、読み込むタイプをリード・オンリー・タイプとして登録すること（このページの「 リード・オンリー・タイプをサポートするには？」を参照）です。こうするとアプリケーションはそのファイルをオープンでき、しかも読み込まれたドキュメントは「Untitled」としてやってきます。

次にこれを自動的にネイティブなタイプに変換するには、NSDocumentのサブクラスで readFromで始まる適当なメソッドをオーバーライドして、スーパークラスの実装を呼び、そのあとで setFileURL: と setFileType: を使用してファイル名とタイプをリセットします。なお。ファイル名の設定のときもし拡張子がついていれば、必ずそれを削除して新しいタイプの拡張子を付加してください。

セーブ・パネルをカスタマイズするには？

NSDocument クラスの shouldRunSavePanelWithAccessoryView メソッドをオーバーライドすると、セーブ・パネルに表れるデフォルトのアクセサリ・ビュー（ユーザにドキュメントを保存するタイプを選択させるポップアップメニューを表示する）を出ないようにすることができます。アプリケーションが複数のタイプの書き出しをサポートしていて、このメソッドがYESを返す場合（それがデフォルトなのですが）、セーブ・パネルはアクセサリ・ビューを含んで表示されます。

prepareSavePanel: メソッドをオーバーライドすると、もっと完全にセーブ・パネルをカスタマイズすることができます。例えばアクセサリ・ビューをまるごとほかのものと置き換えてしまうことも可能です。

プリントを実装するには？

NSDocument のサブクラスでプリントを実装する場合には、 printShowingPrintPanel: メソッドをオーバーライドしてください。通常このメソッドはドキュメントのプリント情報を使って NSPrintOperationオブジェクトを作成し、それを実行します。

なお、ウインドウ・コントローラの存在いかんに関わらず、ドキュメントはそれ自体のプリントを行えるようになっているべきです。

プリント情報に関して何かすることは？

理想を言えば、ドキュメントのプリント情報は他の内容と同様、セーブされロードされるドキュメントの一部として扱われるべきです。しかしながら、ドキュメントのフォーマットが既に定義済みでプリント情報をそれに含めるほど冗長性がないなどの理由で、これがいつも可能というわけにはいきません。

このプリント情報のセーブとロードを別にすれば、これに関して別段しなければならないことはありません。

特定のnibファイルを自動的に使うNSWIndowControllerサブクラスを作るには？

それが全てのウインドウ・コントローラのデフォルトの実装なので、NSWindowController オブジェクトは、ロードするべき nibファイルを、その initWithWindowNibdで始まるいくつかのメソッドを通して指示されるものと予想します。しかしながら NSWindowControllerをサブクラスした場合、そのサブクラスはそのために定義されたnibファイルのユーザ・インタフェースをコントロールするようデザインされます、というより、違うnibファイルは制御できません。これでは不便ですから、どのnibファイルをロードすべきかをサブクラスに設定しなければなりません。

この問題は単にスーパークラスの initWithWindowNibName: メソッドを呼ぶだけになっている初期化メソッドをオーバーライドして正しいnibファイル名を与えることで容易に解決が可能です。また、initWithWindowNibで始まる初期化メソッドをオーバーライドすることで他のクライアントから違うnibファイルのロードを指示されることを拒めます。これは特定のnibファイルを使うように設計されたNSWindowController サブクラスにとっていいアイディアです。NSWindowController サブクラスがその基本的な機能を拡張しており、特定のnibファイルの存在を前提としていない場合に限り、これ以外の方法をとるべきです。

共有パネル（インスペクタや検索パネルなど）にNSWindowControllerを使うには？

NSDocument オブジェクトに結びついていない NSWindowController オブジェクトは単独で活用されます。例えば、nibファイルをより効率的に管理するための補助パネルのコントローラの基底クラスとして使われています。

スタンドアロン NSWindowController サブクラスの一般的な用途に、検索パネル、インスペクタ、初期設定パネルなど共有パネルのコントロールがあります。この場合、プログラマはインスタンスを共有するためのメソッドを持つ NSWindowController サブクラスを作るわけです。例えば、 PreferencesController のサブクラスを作り、これに sharedPreferenceControllerというクラスメソッドをインプリメントします。このメソッドは、最初に呼ばれたときにひとつインスタンスを生成、リテインしておいて返し、2度目からはこのリテインされたインスタンスを返します。

このサブクラスは NSWindowController をスーパークラスとしていますから、Preferences nibファイルの名前を与えるだけで、自動的にそのnibファイルをロードしてウインドウを管理できます。あとはパネルにユーザ・インタフェース・オブジェクトを配置し、これらをマネージメントするためにアウトレットやアクションを接続したり、パネルそのものをコントロールするメソッドを加えたりすればOKです。

サンプル・アプリケーション Sketch ではいろいろなセカンダリ・パネルにこの手法を使っていますので参考にしてください。

1つのドキュメントのために複数のNSWindowControllerを使うには？

ウインドウ・コントローラを生成するために makeWindowControllers メソッドを使っていれば（45ページの「 NSWindowControllerをサブクラスする方法は？」を参照）、そのメソッドで最初から必要なだけ異なるタイプの NSWindowControllerサブクラスを生成しておけます。別の方法としては、アプリケーションが新たなコントローラを必要としたときにそれを作るのを許容するやり方があります。いずれにせよ、作成したコントローラ・オブジェクトは addWindowController: を使ってドキュメントに追加しなければなりません。

デフォルトでは、最後のウインドウ・コントローラが閉じられるとドキュメントは閉じられます。また特定のウインドウ・コントローラが閉じられたら他のコントローラが開いていてもドキュメントが閉じられるよう設定することもできます。身近な例がInterface Builderです。nibドキュメントのためのメインウインドウとそれに付随するnibの中身を編集するためのウインドウがあります。ユーザがメインウインドウを閉じると、ドキュメントそのものが閉じられ、結果的に他のウインドウも全て閉じられます。Interface Builderはこの動作を実装するために、メインウインドウに対して setShouldCloseDocument: メッセージでYESを送っています。

NSWindowController で個々のウインドウ・タイトルをカスタマイズするには？

NSWindowControllerのメソッド、 windowTitleForDocumentDisplayName: をオーバーライドすることで個々のビューのタイトルを変更することが可能です。例えばCADプログラムでは「飛行機」というドキュメントを表示する複数のウインドウに「飛行機」「飛行機・上面」「飛行機・側面」などと表示しているビューによって異なるタイトルをつける場合があります。

アンドゥを実装するには？

アンドゥはいつも容易に実装できるわけではありません。が、その実装のメカニズムそのものは簡単です。デフォルトで、NSDocumentオブジェクトはそれぞれの NSUndoManager オブジェクトを持っています。NSUndoManagerクラスを使えば、ドキュメントに加えられた変更の正反対の動作を簡単に構成できるのです。

鍵はドキュメントに変更を加えるプリミティブなメソッドがしっかり定義されていることです。それぞれのモデル・オブジェクト、そしてNSDocumentのサブクラスが、ドキュメントに変更を加えることのできるプリミティブなメソッドのセットを定義していなければなりません。それらのメソッドは、その動作の取り消しを行うためのアンドゥ・マネージャの使用に責任を持ちます。例えば setColor: というメソッドがそのモデル・オブジェクトを変更するプリミティブ・メソッドであるとするなら、setColor: の内部では以下のような処理が必要とされます。

```
[[[myDocument undoManager] prepareInvocationWithTarget:self] setColor:oldColor]
```

この呼び出しはアンドゥ・マネージャにその動作を取り消すシーケンスを記憶させます。ユーザがあとでアンドゥを呼ぶと、記憶されたシーケンスが呼び出され、モデル・オブジェクトに送られます。この場合なら、oldColorをパラメータにした setColor: メッセージです（このアンドゥの実行をトレースする必要があるのではないかお思いなら、その必要はありません。実際、リドゥはアンドゥが発生したときにレジスターされた動作を監視し、それをリドゥ・スタックに記録することで実現されます）。

良いアンドゥ実装に必要なもう1つの点は、アンドゥ、リドゥのメニュー・アイテムをより具体的なものにすることです。アンドゥのアクション名は、モデル・オブジェクト対してコールされたプリミティブ・メソッドの名称ではなく、より包括的な動作を表す言葉にするべきです。なぜなら、一つのユーザ・アクションによって複数の変更が引き起こされる場合もあるし、また異なるユーザ・アクションが同じプリミティブ・メソッドを違う目的で呼ぶかもしれないからです。サンプル・アプリケーション Sketch はこの方法でアンドゥを実装していますので参考にしてください。なお、アンドゥのサポートに関する詳細に関しては「Undo Architecture」というドキュメントを参照してください。

部分的アンドゥを実装するには？

NSUndoManager がマルチレベルのアンドゥをサポートしますから、ドキュメント・オブジェクトがそのサブセットのような部分的なアンドゥを実装するのはいいアイディアとは言えません。アンドゥ・マネージャは、繰り返されたアンドゥのヒストリーを通してドキュメントを元に戻すことが可能であるという前提の上に機能しています。もしいくつかの変更がスキップされてしまうと、アンドゥ・スタックとドキュメントの中身は同期しなくなってしまいます。それはユーザをいらいらさせるだけでなく、時には致命的な問題を引き起こします。

もしプログラムで元に戻すことのできないような変更があるとすれば、ユーザがその変更を行った場合の対処は２つ考えられます。その変更がドキュメントの他の部分と全く関係ないことが確信できる場合には、ただその変更をアンドゥ・マネージャに記録しないで済ませることができます（もちろんアンドゥは行われませんがアンドゥ・スタックの整合性は損なわれません）。逆に言えば、その変更がドキュメントの内容の他の部分になんらかの関係がある場合は、関連する全ての動作をアンドゥ・マネージャから取り除かなければなりません。また、そのような変更はユーザに「引き返し限界点」として知らされるべきですし、もっと言えば、アプリケーションとドキュメントの設計段階でそうしたものが必要にならないように努力をするべきです。

アンドゥをサポートしたくなかったら？

アプリケーションでアンドゥをサポートしたくない場合には、setHasUndoManager: メソッドにパラメータ NOを添えてコールします。こうするとドキュメントはアンドゥ・マネージャを持たなくなります。

ただし、アンドゥ・マネージャを持たなければドキュメントは変更状態を自動的に追跡することができなくなります。したがってアンドゥを実装しない場合には、ドキュメントが編集されるごとに updateChangeCount: を呼び出してそれを記録する必要があります。

チェンジ・カウントって何？

アンドゥをサポートするために、ドキュメントは自身が変更されたか（ダーティ）そうでないか（クリーン）という以上の情報を保持しなければなりません。ユーザがファイルを開いて5回の変更を行って、5回アンドゥを選んだら、ドキュメントはクリーンに戻ります。でもユーザが実行したアンドゥが4回だったら、ドキュメントはまだダーティでなくてはなりません。

NSDocument はこれを実現するためにチェンジ・カウントを保持しています。チェンジ・カウントは、変更タイプをパラメータに updateChangeCount: を呼ぶことで変更できます。サポートされている変更タイプは、NSChangeDone、NSChangeUndone、そして NSChangeClearedの３つです。チェンジ・カウントはユーザがドキュメントを保存するか、最後に保存した状態に戻すとクリアされます。ドキュメントがアンドゥ・マネージャを持っている場合、アンドゥ・マネージャはドキュメントの変更、アンドゥ。そしてリドゥを感知して自動的にチェンジ・カウントを更新してくれます。

ドキュメントのサブクラスがアンドゥをサポートしない場合には、 updateChangeCount: を呼んで自身でチェンジ・カウントを更新しなければなりません（前項「アンドゥをサポートしたくなかったら？」を参照）。

NSDocumentControllerをサブクラスするべきなのは？

通常、プログラマが NSDocumentController をサブクラスする必要はありません。このクラスをサブクラスしてできることのほとんどはアプリケーションのデリゲートで同じくらい容易に行えるはずです。が、どうしても必要であればもちろん NSDocumentController をサブクラスすることは可能です。

例えば、オープン・パネルをカスタマイズする必要がある場合には確かに NSDocumentController をサブクラスする必要があります。プログラマはそのメソッド、 runModalOpenPanel:forTypes: をオーバーライドして、パネルやそのアクセサリ・ビューをカスタマイズすることができます。

NSDocumentControllerをサブクラスする方法は？

NSDocumentController をサブクラスするには以下の２つの方法があります。

＊アプリケーションのメイン nib ファイルで NSDocumentController サブクラスのインスタンスを作成する。

＊アプリケーション・デリゲートの applicationWillFinishLaunching: メソッドで NSDocumentController サブクラスを作成する。

上のようにして作成された NSDocumentController オブジェクトは共有インスタンスになります。Application Kit は NSDocumentControllerの共有インスタンスをアプリケーション起動時の「finish launching」のタイミングで作成します。ですからサブクラスのインスタンスはそれより以前に作成されなければなりません。

ユーザ・アクション・メソッドを介さないで新規ドキュメントを作るには？

NSDocumentController のメソッド、 openUntitledDocumentAndDisplay:error: と、openDocumentWithContentsOfURL:display:error: を使えば、ドキュメントを生成することが可能です。またパラメータ display: にYESを指定すれば、ドキュメントのウインドウ・コントローラを作成し、ドキュメントを開かれているドキュメントのリストに加えることもできます。また後者のメソッドは指定されたパスをチェックして、既存のファイルがあればそのドキュメントを返します。

NSDocumentController の単にドキュメントを作るだけのメソッド、makeUntitledDocumentOfType:error:、 makeDocumentWithContentsOfURL:ofType:error:、そしてmakeDocumentForURL:withContentsOfURL:ofType:error: を使うこともできます。これらを使えばそのタイプを指定することで望みのNSDocumentサブクラスを初期化することができます。

どちらの場合にも、 NSDocumentController の addDocument: メソッドを使って作成したドキュメントをドキュメント・コントローラのドキュメント・リストに追加する必要があります。

アプリケーションが起動時に名称未設定ドキュメントを作るのを防ぐには？

アプリケーション・デリゲートに applicationShouldOpenUntitledFile: を実装してNOを返すようにすれば、アプリケーションは起動時に名称未設定ドキュメントを作りません。もし起動時に名称未設定ドキュメントを作るようにはしたいけれど、既に起動していてドックから呼び戻されるときにはそれを望まないという場合には、上のメソッドの代わりに applicationShouldHandleReopen:hasVisibleWindows: を実装してやはりNOを返すようにしてください。

ドキュメント更新履歴

オリジナル・ドキュメントは「Document-Based Application Overview」2006/03/08版。

要約は2006/06/12完成。