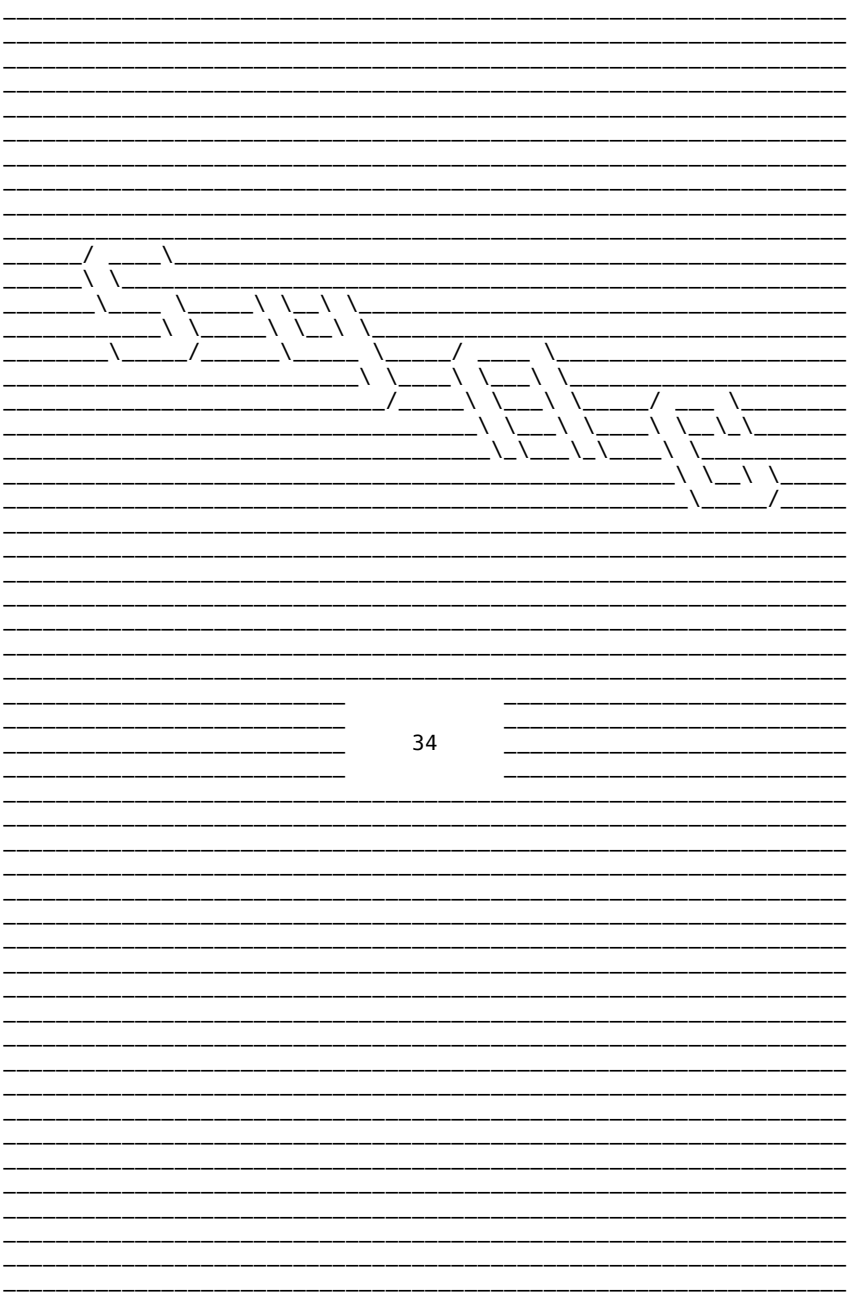


\\\ an ongoing artistic journal in digitally published zines \\\

source

Document.m

```
/*
     File: Document.m
 Abstract: Document object for TextEdit.

  Version: 1.9

 Disclaimer: IMPORTANT:  This Apple software is supplied to you by Apple
 Inc. ("Apple") in consideration of your agreement to the following
 terms, and your use, installation, modification or redistribution of
 this Apple software constitutes acceptance of these terms.  If you do
 not agree with these terms, please do not use, install, modify or
 redistribute this Apple software.

 In consideration of your agreement to abide by the following terms, and
 subject to these terms, Apple grants you a personal, non-exclusive
 license, under Apple's copyrights in this original Apple software (the
 "Apple Software"), to use, reproduce, modify and redistribute the Apple
 Software, with or without modifications, in source and/or binary forms;
 provided that if you redistribute the Apple Software in its entirety and
 without modifications, you must retain this notice and the following
 text and disclaimers in all such redistributions of the Apple Software.
 Neither the name, trademarks, service marks or logos of Apple Inc. may
 be used to endorse or promote products derived from the Apple Software
 without specific prior written permission from Apple.  Except as
 expressly stated in this notice, no other rights or licenses, express or
 implied, are granted by Apple herein, including but not limited to any
 patent rights that may be infringed by your derivative works or by other
 works in which the Apple Software may be incorporated.

 The Apple Software is provided by Apple on an "AS IS" basis.  APPLE
 MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION
 THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS
 FOR A PARTICULAR PURPOSE, REGARDING THE APPLE SOFTWARE OR ITS USE AND
 OPERATION ALONE OR IN COMBINATION WITH YOUR PRODUCTS.

 IN NO EVENT SHALL APPLE BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL
 OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 INTERRUPTION) ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION,
 MODIFICATION AND/OR DISTRIBUTION OF THE APPLE SOFTWARE, HOWEVER CAUSED
 AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE),
 STRICT LIABILITY OR OTHERWISE, EVEN IF APPLE HAS BEEN ADVISED OF THE
 POSSIBILITY OF SUCH DAMAGE.

 Copyright (C) 2013 Apple Inc. All Rights Reserved.

 */



#import <Cocoa/Cocoa.h>
#import "EncodingManager.h"
#import "Document.h"
#import "DocumentController.h"
#import "DocumentWindowController.h"
#import "PrintPanelAccessoryController.h"
#import "PrintingTextView.h"
#import "TextEditDefaultsKeys.h"
#import "TextEditErrors.h"
#import "TextEditMisc.h"
#import <objc/message.h> // objc_msgSend

#define oldEditPaddingCompensation 12.0

NSString *SimpleTextType = @"com.apple.traditional-mac-plain-text";
NSString *Word97Type = @"com.microsoft.word.doc";
NSString *Word2007Type = @"org.openxmlformats.wordprocessingml.document";
NSString *Word2003XMLType = @"com.microsoft.word.wordml";
```

```
NSString *OpenDocumentTextType = @"org.oasis-open.opendocument.text";


@implementation Document

+ (BOOL)isRichTextType:(NSString *)typeName {
    /* We map all plain text documents to public.text.  Therefore a document is rich iff
its type is not public.text. */
    return ![typeName isEqualToString:(NSString *)kUTTypeText];
}

+ (NSString *)readableTypeForType:(NSString *)type {
    // There is a partial order on readableTypes given by UTTypeConformsTo. We linearly
extend the partial order to a total order using <.
    // Therefore we can compute the ancestor with greatest level (furthest from root) by
linear search in the resulting array.
    // Why do we have to do this?  Because type might conform to multiple readable types,
such as "public.rtf" and "public.text" and "public.data"
    // and we want to find the most specialized such type.te
    static NSArray *topologicallySortedReadableTypes;
    static dispatch_once_t pred;
    dispatch_once(&pred, ^{
        topologicallySortedReadableTypes = [self readableTypes];
        topologicallySortedReadableTypes = [topologicallySortedReadableTypes
sortedArrayUsingComparator:^NSComparisonResult(id type1, id type2) {
            if (type1 == type2) return NSOrderedSame;
            if (UTTypeConformsTo((CFStringRef)type1, (CFStringRef)type2)) return
NSOrderedAscending;
            if (UTTypeConformsTo((CFStringRef)type2, (CFStringRef)type1)) return
NSOrderedDescending;
            return (((NSUInteger)type1 < (NSUInteger)type2) ? NSOrderedAscending :
NSOrderedDescending);
        }];
        [topologicallySortedReadableTypes retain];
    });
    for (NSString *readableType in topologicallySortedReadableTypes) {
        if (UTTypeConformsTo((CFStringRef)type, (CFStringRef)readableType)) return
readableType;
    }
    return nil;
}

- (id)init {
    if ((self = [super init])) {
        [[self undoManager] disableUndoRegistration];

           textStorage = [[NSTextStorage allocWithZone:[self zone]] init];

           [self setBackgroundColor:[NSColor whiteColor]];
           [self setEncoding:NoStringEncoding];
           [self setEncodingForSaving:NoStringEncoding];
           [self setScaleFactor:1.0];
           [self setDocumentPropertiesToDefaults];
           inDuplicate = NO;

           // Assume the default file type for now, since -initWithType:error: does not
currently get called when creating documents using AppleScript. (4165700)
           [self setFileType:[[NSDocumentController sharedDocumentController]
defaultType]];

           [self setPrintInfo:[self printInfo]];

           hasMultiplePages = [[NSUserDefaults standardUserDefaults]
boolForKey:ShowPageBreaks];

           [self setUsesScreenFonts:[self isRichText] ? [[NSUserDefaults
standardUserDefaults] boolForKey:UseScreenFonts] : YES];

           [[self undoManager] enableUndoRegistration];
```

```
    }
    return self;
}

/* Return an NSDictionary which maps Cocoa text system document identifiers (as declared
in AppKit/NSAttributedString.h) to document types declared in TextEdit's Info.plist.
*/
- (NSDictionary *)textDocumentTypeToTextEditDocumentTypeMappingTable {
    static NSDictionary *documentMappings = nil;
    // Use of dispatch_once() makes the initialization thread-safe, and it needs to be,
since multiple documents can be opened concurrently
    static dispatch_once_t once = 0;
    dispatch_once(&once, ^{
           documentMappings = [[NSDictionary alloc] initWithObjectsAndKeys:
            (NSString *)kUTTypeText, NSPlainTextDocumentType,
            (NSString *)kUTTypeRTF, NSRTFTextDocumentType,
            (NSString *)kUTTypeRTFD, NSRTFDTextDocumentType,
            SimpleTextType, NSMacSimpleTextDocumentType,
            (NSString *)kUTTypeHTML, NSHTMLTextDocumentType,
               Word97Type, NSDocFormatTextDocumentType,
               Word2007Type, NSOfficeOpenXMLTextDocumentType,
               Word2003XMLType, NSWordMLTextDocumentType,
               OpenDocumentTextType, NSOpenDocumentTextDocumentType,
            (NSString *)kUTTypeWebArchive, NSWebArchiveTextDocumentType,
               nil];
        });
    return documentMappings;
}

/* This method is called by the document controller. The message is passed on after
information about the selected encoding (from our controller subclass) and preference
regarding HTML and RTF formatting has been added. -lastSelectedEncodingForURL: returns the
encoding specified in the Open panel, or the default encoding if the document was opened
without an open panel.
*/
- (BOOL)readFromURL:(NSURL *)absoluteURL ofType:(NSString *)typeName error:(NSError
**)outError {
    DocumentController *docController = [DocumentController sharedDocumentController];
    return [self readFromURL:absoluteURL ofType:typeName encoding:[docController
lastSelectedEncodingForURL:absoluteURL] ignoreRTF:[docController
lastSelectedIgnoreRichForURL:absoluteURL] ignoreHTML:[docController
lastSelectedIgnoreHTMLForURL:absoluteURL] error:outError];
}

- (BOOL)readFromURL:(NSURL *)absoluteURL ofType:(NSString *)typeName encoding:
(NSStringEncoding)encoding ignoreRTF:(BOOL)ignoreRTF ignoreHTML:(BOOL)ignoreHTML error:
(NSError **)outError {
    NSMutableDictionary *options = [NSMutableDictionary dictionaryWithCapacity:5];
    NSDictionary *docAttrs;
    id val, paperSizeVal, viewSizeVal;
    NSTextStorage *text = [self textStorage];

    /* generalize the passed-in type to a type we support.  for instance, generalize
"public.xml" to "public.txt" */
    typeName = [[self class] readableTypeForType:typeName];

    [fileTypeToSet release];
    fileTypeToSet = nil;

    [[self undoManager] disableUndoRegistration];

    [options setObject:absoluteURL forKey:NSBaseURLDocumentOption];
    if (encoding != NoStringEncoding) {
        [options setObject:[NSNumber numberWithUnsignedInteger:encoding]
forKey:NSCharacterEncodingDocumentOption];
    }
    [self setEncoding:encoding];

    // Check type to see if we should load the document as plain. Note that this check
```

```
isn't always conclusive, which is why we do another check below, after the document has
been loaded (and correctly categorized).
    NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
    if ((ignoreRTF && ([workspace type:typeName conformsToType:(NSString *)kUTTypeRTF] ||
[workspace type:typeName conformsToType:Word2003XMLType])) || (ignoreHTML && [workspace
type:typeName conformsToType:(NSString *)kUTTypeHTML]) || [self isOpenedIgnoringRichText])
{
        [options setObject:NSPlainTextDocumentType
forKey:NSDocumentTypeDocumentOption]; // Force plain
        typeName = (NSString *)kUTTypeText;
        [self setOpenedIgnoringRichText:YES];
    }

    [[text mutableString] setString:@""];
    // Remove the layout managers while loading the text; mutableCopy retains the array so
the layout managers aren't released
    NSMutableArray *layoutMgrs = [[text layoutManagers] mutableCopy];
    NSEnumerator *layoutMgrEnum = [layoutMgrs objectEnumerator];
    NSLayoutManager *layoutMgr = nil;
    while ((layoutMgr = [layoutMgrEnum nextObject])) [text removeLayoutManager:layoutMgr];

    // We can do this loop twice, if the document is loaded as rich text although the user
requested plain
    BOOL retry;
    do {
           BOOL success;
           NSString *docType;

           retry = NO;

           [text beginEditing];
           success = [text readFromURL:absoluteURL options:options
documentAttributes:&docAttrs error:outError];

        if (!success) {
              [text endEditing];
              layoutMgrEnum = [layoutMgrs objectEnumerator]; // rewind
              while ((layoutMgr = [layoutMgrEnum nextObject])) [text
addLayoutManager:layoutMgr];   // Add the layout managers back
              [layoutMgrs release];
              return NO;        // return NO on error; outError has already been set
           }

           docType = [docAttrs objectForKey:NSDocumentTypeDocumentAttribute];

           // First check to see if the document was rich and should have been loaded as
plain
           if (![[options objectForKey:NSDocumentTypeDocumentOption]
isEqualToString:NSPlainTextDocumentType] && ((ignoreHTML && [docType
isEqual:NSHTMLTextDocumentType]) || (ignoreRTF && ([docType isEqual:NSRTFTextDocumentType]
|| [docType isEqual:NSWordMLTextDocumentType])))) {
              [text endEditing];
              [[text mutableString] setString:@""];
              [options setObject:NSPlainTextDocumentType
forKey:NSDocumentTypeDocumentOption];
              typeName = (NSString *)kUTTypeText;
              [self setOpenedIgnoringRichText:YES];
              retry = YES;
           } else {
              NSString *newFileType = [[self
textDocumentTypeToTextEditDocumentTypeMappingTable] objectForKey:docType];
              if (newFileType) {
            typeName = newFileType;
              } else {
            typeName = (NSString *)kUTTypeRTF; // Hmm, a new type in the Cocoa text
system. Treat it as rich. ??? Should set the converted flag too?
              }
        if (![[self class] isRichTextType:typeName]) [self applyDefaultTextAttributes:NO];
              [text endEditing];
```

```
            }
    } while(retry);

    [self setFileType:typeName];
    // If we're reverting, NSDocument will set the file type behind out backs. This
enables restoring that type.
    fileTypeToSet = [typeName copy];

    layoutMgrEnum = [layoutMgrs objectEnumerator]; // rewind
    while ((layoutMgr = [layoutMgrEnum nextObject])) [text
addLayoutManager:layoutMgr];   // Add the layout managers back
    [layoutMgrs release];

    val = [docAttrs objectForKey:NSCharacterEncodingDocumentAttribute];
    [self setEncoding:(val ? [val unsignedIntegerValue] : NoStringEncoding)];

    if ((val = [docAttrs objectForKey:NSConvertedDocumentAttribute])) {
        [self setConverted:([val integerValue] > 0)];   // Indicates filtered
        [self setLossy:([val integerValue] < 0)];       // Indicates lossily loaded
    }

    /* If the document has a stored value for view mode, use it. Otherwise wrap to window.
*/
    if ((val = [docAttrs objectForKey:NSViewModeDocumentAttribute])) {
        [self setHasMultiplePages:([val integerValue] == 1)];
        if ((val = [docAttrs objectForKey:NSViewZoomDocumentAttribute])) {
            [self setScaleFactor:([val doubleValue] / 100.0)];
        }
    } else [self setHasMultiplePages:NO];

    [self willChangeValueForKey:@"printInfo"];
    if ((val = [docAttrs objectForKey:NSLeftMarginDocumentAttribute])) [[self printInfo]
setLeftMargin:[val doubleValue]];
    if ((val = [docAttrs objectForKey:NSRightMarginDocumentAttribute])) [[self printInfo]
setRightMargin:[val doubleValue]];
    if ((val = [docAttrs objectForKey:NSBottomMarginDocumentAttribute])) [[self printInfo]
setBottomMargin:[val doubleValue]];
    if ((val = [docAttrs objectForKey:NSTopMarginDocumentAttribute])) [[self printInfo]
setTopMargin:[val doubleValue]];
    [self didChangeValueForKey:@"printInfo"];

    /* Pre MacOSX versions of TextEdit wrote out the view (window) size in PaperSize.
           If we encounter a non-MacOSX RTF file, and it's written by TextEdit, use
PaperSize as ViewSize */
    viewSizeVal = [docAttrs objectForKey:NSViewSizeDocumentAttribute];
    paperSizeVal = [docAttrs objectForKey:NSPaperSizeDocumentAttribute];
    if (paperSizeVal && NSEqualSizes([paperSizeVal sizeValue], NSZeroSize)) paperSizeVal =
nil;       // Protect against some old documents with 0 paper size

    if (viewSizeVal) {
        [self setViewSize:[viewSizeVal sizeValue]];
        if (paperSizeVal) [self setPaperSize:[paperSizeVal sizeValue]];
    } else {          // No ViewSize...
        if (paperSizeVal) {      // See if PaperSize should be used as ViewSize; if so,
we also have some tweaking to do on it
            val = [docAttrs objectForKey:NSCocoaVersionDocumentAttribute];
            if (val && ([val integerValue] < 100)) {    // Indicates old RTF file; value
described in AppKit/NSAttributedString.h
                NSSize size = [paperSizeVal sizeValue];
                if (size.width > 0 && size.height > 0 && ![self hasMultiplePages]) {
                    size.width = size.width - oldEditPaddingCompensation;
                    [self setViewSize:size];
                }
            } else {
                    [self setPaperSize:[paperSizeVal sizeValue]];
            }
        }
    }
```

```
    [self setHyphenationFactor:(val = [docAttrs
objectForKey:NSHyphenationFactorDocumentAttribute]) ? [val floatValue] : 0];
    [self setBackgroundColor:(val = [docAttrs
objectForKey:NSBackgroundColorDocumentAttribute]) ? val : [NSColor whiteColor]];

    // Set the document properties, generically, going through key value coding
    NSDictionary *map = [self documentPropertyToAttributeNameMappings];
    for (NSString *property in [self knownDocumentProperties]) [self setValue:[docAttrs
objectForKey:[map objectForKey:property]] forKey:property];        // OK to set nil to
clear

    [self setReadOnly:((val = [docAttrs objectForKey:NSReadOnlyDocumentAttribute]) &&
([val integerValue] > 0))];

    [self setOriginalOrientationSections:[docAttrs
objectForKey:NSTextLayoutSectionsAttribute]];

    [self setUsesScreenFonts:[self isRichText] ? [[docAttrs
objectForKey:NSUsesScreenFontsDocumentAttribute] boolValue] : YES];

    [[self undoManager] enableUndoRegistration];

    return YES;
}

- (NSDictionary *)defaultTextAttributes:(BOOL)forRichText {
    static NSParagraphStyle *defaultRichParaStyle = nil;
    NSMutableDictionary *textAttributes = [[[NSMutableDictionary alloc] initWithCapacity:
2] autorelease];
    if (forRichText) {
            [textAttributes setObject:[NSFont userFontOfSize:0.0]
forKey:NSFontAttributeName];
            if (defaultRichParaStyle == nil) {           // We do this once...
                NSInteger cnt;
                NSString *measurementUnits = [[NSUserDefaults standardUserDefaults]
objectForKey:@"AppleMeasurementUnits"];
                CGFloat tabInterval = ([@"Centimeters" isEqual:measurementUnits]) ? (72.0 /
2.54) : (72.0 / 2.0);  // Every cm or half inch
                NSMutableParagraphStyle *paraStyle = [[[NSMutableParagraphStyle alloc]
init] autorelease];
                NSTextTabType type = ((NSWritingDirectionRightToLeft == [NSParagraphStyle
defaultWritingDirectionForLanguage:nil]) ? NSRightTabStopType : NSLeftTabStopType);
                [paraStyle setTabStops:[NSArray array]]; // This first clears all tab stops
                for (cnt = 0; cnt < 12; cnt++) {         // Add 12 tab stops, at desired
intervals...
                    NSTextTab *tabStop = [[NSTextTab alloc] initWithType:type
location:tabInterval * (cnt + 1)];
                        [paraStyle addTabStop:tabStop];
                        [tabStop release];
                }
                defaultRichParaStyle = [paraStyle copy];
            }
            [textAttributes setObject:defaultRichParaStyle
forKey:NSParagraphStyleAttributeName];
    } else {
            NSFont *plainFont = [NSFont userFixedPitchFontOfSize:0.0];
        NSFont *charWidthFont = [plainFont
screenFontWithRenderingMode:NSFontDefaultRenderingMode];
            NSInteger tabWidth = [[NSUserDefaults standardUserDefaults]
integerForKey:TabWidth];
            CGFloat charWidth = [@" " sizeWithAttributes:[NSDictionary
dictionaryWithObject:charWidthFont forKey:NSFontAttributeName]].width;
        if (charWidth == 0) charWidth = [charWidthFont maximumAdvancement].width;

            // Now use a default paragraph style, but with the tab width adjusted
            NSMutableParagraphStyle *mStyle = [[[NSParagraphStyle defaultParagraphStyle]
mutableCopy] autorelease];
            [mStyle setTabStops:[NSArray array]];
            [mStyle setDefaultTabInterval:(charWidth * tabWidth)];
```

```
        [textAttributes setObject:[[mStyle copy] autorelease]
forKey:NSParagraphStyleAttributeName];

           // Also set the font
           [textAttributes setObject:plainFont forKey:NSFontAttributeName];
    }
    return textAttributes;
}

- (void)applyDefaultTextAttributes:(BOOL)forRichText {
    NSDictionary *textAttributes = [self defaultTextAttributes:forRichText];
    NSTextStorage *text = [self textStorage];
    // We now preserve base writing direction even for plain text, using the 10.6-
introduced attribute enumeration API
    [text enumerateAttribute:NSParagraphStyleAttributeName inRange:NSMakeRange(0, [text
length]) options:0 usingBlock:^(id paragraphStyle, NSRange paragraphStyleRange, BOOL
*stop){
        NSWritingDirection writingDirection = paragraphStyle ? [(NSParagraphStyle
*)paragraphStyle baseWritingDirection] : NSWritingDirectionNatural;
        // We also preserve NSWritingDirectionAttributeName (new in 10.6)
        [text enumerateAttribute:NSWritingDirectionAttributeName
inRange:paragraphStyleRange options:0 usingBlock:^(id value, NSRange attributeRange, BOOL
*stop){
            [value retain];
            [text setAttributes:textAttributes range:attributeRange];
            if (value) [text addAttribute:NSWritingDirectionAttributeName value:value
range:attributeRange];
            [value release];
        }];
        if (writingDirection != NSWritingDirectionNatural) [text
setBaseWritingDirection:writingDirection range:paragraphStyleRange];
    }];
}


/* This method will return a suggested encoding for the document. Since Mac OS X Leopard,
unless the user has specified a favorite encoding for saving that applies to the document,
we use UTF-8.
*/
- (NSStringEncoding)suggestedDocumentEncoding {
    NSUInteger enc = NoStringEncoding;
    NSNumber *val = [[NSUserDefaults standardUserDefaults]
objectForKey:PlainTextEncodingForWrite];
    if (val) {
        NSStringEncoding chosenEncoding = [val unsignedIntegerValue];
        if ((chosenEncoding != NoStringEncoding)  && (chosenEncoding !=
NSUnicodeStringEncoding) && (chosenEncoding != NSUTF8StringEncoding)) {
            if ([[[self textStorage] string] canBeConvertedToEncoding:chosenEncoding])
enc = chosenEncoding;
        }
    }
    if (enc == NoStringEncoding) enc = NSUTF8StringEncoding;       // Default to UTF-8
    return enc;
}

/* Clear the delegates of the text views and window, then release all resources and go
away...
*/
- (void)dealloc {
    [textStorage release];
    [backgroundColor release];

    [author release];
    [comment release];
    [subject release];
    [title release];
    [keywords release];
    [copyright release];
    [company release];
```

```
    [fileTypeToSet release];

    [originalOrientationSections release];
    [super dealloc];
}

- (CGFloat)scaleFactor {
    return scaleFactor;
}

- (void)setScaleFactor:(CGFloat)newScaleFactor {
    scaleFactor = newScaleFactor;
}

- (NSSize)viewSize {
    return viewSize;
}

- (void)setViewSize:(NSSize)size {
    viewSize = size;
}

- (void)setReadOnly:(BOOL)flag {
    isReadOnly = flag;
}

- (BOOL)isReadOnly {
    return isReadOnly;
}

- (void)setBackgroundColor:(NSColor *)color {
    id oldCol = backgroundColor;
    backgroundColor = [color copy];
    [oldCol release];
}

- (NSColor *)backgroundColor {
    return backgroundColor;
}

- (NSTextStorage *)textStorage {
    return textStorage;
}

- (NSSize)paperSize {
    return [[self printInfo] paperSize];
}

- (void)setPaperSize:(NSSize)size {
    NSPrintInfo *oldPrintInfo = [self printInfo];
    if (!NSEqualSizes(size, [oldPrintInfo paperSize])) {
           NSPrintInfo *newPrintInfo = [oldPrintInfo copy];
           [newPrintInfo setPaperSize:size];
           [self setPrintInfo:newPrintInfo];
           [newPrintInfo release];
    }
}

/* Layout orientation sections */
- (void)setOriginalOrientationSections:(NSArray *)array {
    [originalOrientationSections release];
    originalOrientationSections = [array copy];
}

- (NSArray *)originalOrientationSections {
    return originalOrientationSections;
}
```

```
/* Screen fonts property */
- (void)setUsesScreenFonts:(BOOL)aFlag {
    usesScreenFonts = aFlag;
}

- (BOOL)usesScreenFonts {
    return usesScreenFonts;
}

/* Hyphenation related methods.
*/
- (void)setHyphenationFactor:(float)factor {
    hyphenationFactor = factor;
}

- (float)hyphenationFactor {
    return hyphenationFactor;
}

/* Encoding...
*/
- (NSUInteger)encoding {
    return documentEncoding;
}

- (void)setEncoding:(NSUInteger)encoding {
    documentEncoding = encoding;
}

/* This is the encoding used for saving; valid only during a save operation
*/
- (NSUInteger)encodingForSaving {
    return documentEncodingForSaving;
}

- (void)setEncodingForSaving:(NSUInteger)encoding {
    documentEncodingForSaving = encoding;
}


- (BOOL)isConverted {
    return convertedDocument;
}

- (void)setConverted:(BOOL)flag {
    convertedDocument = flag;
}

- (BOOL)isLossy {
    return lossyDocument;
}

- (void)setLossy:(BOOL)flag {
    lossyDocument = flag;
}

- (BOOL)isOpenedIgnoringRichText {
    return openedIgnoringRichText;
}

- (void)setOpenedIgnoringRichText:(BOOL)flag {
    openedIgnoringRichText = flag;
}

/* A transient document is an untitled document that was opened automatically. If a real
document is opened before the transient document is edited, the real document should
replace the transient. If a transient document is edited, it ceases to be transient.
*/
- (BOOL)isTransient {
```

```
    return transient;
}

- (void)setTransient:(BOOL)flag {
    transient = flag;
}

/* We can't replace transient document that have sheets on them.
*/
- (BOOL)isTransientAndCanBeReplaced {
    if (![self isTransient]) return NO;
    for (NSWindowController *controller in [self windowControllers]) if ([[controller
window] attachedSheet]) return NO;
    return YES;
}

- (void)setFileType:(NSString *)type {
    /* Due to sandboxing, we cannot (usefully) directly change the document's file URL
except for changing it to nil.  This means that when we convert a document from rich text
to plain text, our only way of updating the file URL is to have NSDocument do it for us in
response to a change in our file type.  However, it is not as simple as setting our type
to kUTTypeText, as would be accurate, because if we have a rtf document, public.rtf
inherits from public.text and so NSDocument wouldn't change our extension (which is
correct, BTW, since it's also perfectly valid to open a rtf and not interpret the rtf
commands, in which case a save of a rtf document as kUTTypeText should not change the
extension).  Therefore we need to save using a subtype of kUTTypeText that isn't in the
path from kUTTypeText to kUTTypeRTF.  The obvious candidate is kUTTypePlainText.
Therefore, we need to save using kUTTypePlainText when we convert a rtf to plain text and
then map the file type from kTTypePlainText to kUTTypeText.  The inverse of the mapping
occurs here. */
    if ([type isEqualToString:(NSString *)kUTTypePlainText]) type = (NSString
*)kUTTypeText;
    [super setFileType:type];
}

- (BOOL)isRichText {
    return [[self class] isRichTextType:[self fileType]];
}


/* Document properties management */

/* Table mapping document property keys "company", etc, to text system document attribute
keys (NSCompanyDocumentAttribute, etc)
*/
- (NSDictionary *)documentPropertyToAttributeNameMappings {
    static NSDictionary *dict = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        dict = [[NSDictionary alloc] initWithObjectsAndKeys:
                NSCompanyDocumentAttribute, @"company",
                NSAuthorDocumentAttribute, @"author",
                NSKeywordsDocumentAttribute, @"keywords",
                NSCopyrightDocumentAttribute, @"copyright",
                NSTitleDocumentAttribute, @"title",
                NSSubjectDocumentAttribute, @"subject",
                NSCommentDocumentAttribute, @"comment", nil];
    });
    return dict;
}

- (NSArray *)knownDocumentProperties {
    return [[self documentPropertyToAttributeNameMappings] allKeys];
}

/* If there are document properties and they are not the same as the defaults established
in preferences, return YES
*/
- (BOOL)hasDocumentProperties {
```

```
    for (NSString *key in [self knownDocumentProperties]) {
           id value = [self valueForKey:key];
           if (value && ![value isEqual:[[NSUserDefaults standardUserDefaults]
objectForKey:key]]) return YES;
    }
    return NO;
}

/* This actually clears all properties (rather than setting them to default values
established in preferences)
*/
- (void)clearDocumentProperties {
    for (NSString *key in [self knownDocumentProperties]) [self setValue:nil forKey:key];
}

/* This sets document properties to values established in defaults
*/
- (void)setDocumentPropertiesToDefaults {
    for (NSString *key in [self knownDocumentProperties]) [self setValue:[[NSUserDefaults
standardUserDefaults] objectForKey:key] forKey:key];
}

/* We implement a setValue:forDocumentProperty: to work around NSUndoManager bug where
prepareWithInvocationTarget: fails to freeze-dry invocations with "known" methods such as
setValue:forKey:.
*/
- (void)setValue:(id)value forDocumentProperty:(NSString *)property {
    id oldValue = [self valueForKey:property];
    [[[self undoManager] prepareWithInvocationTarget:self] setValue:oldValue
forDocumentProperty:property];
    [[self undoManager] setActionName:NSLocalizedString(property, "")];       //
Potential strings for action names are listed below (for genstrings to pick up)

    // Call the regular KVC mechanism to get the value to be properly set
    [super setValue:value forKey:property];
}

- (void)setValue:(id)value forKey:(NSString *)key {
    if ([[self knownDocumentProperties] containsObject:key]) {
           [self setValue:value forDocumentProperty:key];          // We take a side-trip
to this method to register for undo
    } else {
           [super setValue:value forKey:key];  // In case some other KVC call is sent to
Document, we treat it normally
    }
}

/* For genstrings:
    NSLocalizedStringWithDefaultValue(@"author", @"", @"", @"Change Author", @"Undo menu
change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"copyright", @"", @"", @"Change Copyright", @"Undo
menu change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"subject", @"", @"", @"Change Subject", @"Undo menu
change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"title", @"", @"", @"Change Title", @"Undo menu
change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"company", @"", @"", @"Change Company", @"Undo menu
change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"comment", @"", @"", @"Change Comment", @"Undo menu
change string, without the 'Undo'");
    NSLocalizedStringWithDefaultValue(@"keywords", @"", @"", @"Change Keywords", @"Undo
menu change string, without the 'Undo'");
*/



- (NSPrintOperation *)printOperationWithSettings:(NSDictionary *)printSettings error:
(NSError **)outError {
    NSPrintInfo *tempPrintInfo = [[[self printInfo] copy] autorelease];
```

```
    [[tempPrintInfo dictionary] addEntriesFromDictionary:printSettings];

    BOOL multiPage = [self hasMultiplePages];

    if ([[self windowControllers] count] == 0) [self makeWindowControllers];
    NSView *documentView = [[[self windowControllers] objectAtIndex:0] documentView];

    id printingView;
    if (multiPage) {    // If already in multiple-page ("wrap-to-page") mode, we simply
use the display view for printing
        printingView = documentView;
        [[[[self windowControllers] objectAtIndex:0] firstTextView]
textEditDoForegroundLayoutToCharacterIndex:NSIntegerMax];                       // Make
sure the whole document is laid out before printing
    } else {            // Otherwise we create a new text view (along with a text
container and layout manager)
        printingView = [[[PrintingTextView alloc] init] autorelease];   //
PrintingTextView is a simple subclass of NSTextView. Creating the view this way creates
rest of the text system, which it will release when dealloc'ed (since the print panel will
be releasing this, we want to hand off the responsibility of release everything)
        NSLayoutManager *layoutManager = [[[[printingView textContainer] layoutManager]
retain] autorelease];
        NSTextStorage *unnecessaryTextStorage = [layoutManager textStorage];  // We don't
want the text storage, since we will use the one we have
        [unnecessaryTextStorage removeLayoutManager:layoutManager];
        [unnecessaryTextStorage release];
        [textStorage addLayoutManager:layoutManager];
        [textStorage retain];   // Since later release of the printingView will release
the textStorage as well
        [printingView setLayoutOrientation:[[[[self windowControllers] objectAtIndex:0]
firstTextView] layoutOrientation]];
    }

    PrintPanelAccessoryController *accessoryController = [[[PrintPanelAccessoryController
alloc] init] autorelease];
    NSPrintOperation *op = [NSPrintOperation printOperationWithView:printingView
printInfo:tempPrintInfo];
    [op setShowsPrintPanel:YES];
    [op setShowsProgressPanel:YES];
    // Since this printing view may not be embedded in a window, we have to manually set
the print job title.
    [op setJobTitle:[documentView printJobTitle]];

    // If we're in wrap-to-page mode, no need to let the user tweak wrap-to-page mode
printing
    [accessoryController setShowsWrappingToFit:!multiPage];

    NSPrintPanel *printPanel = [op printPanel];
    if (!multiPage) {
        [printingView setOriginalSize:[[[[self windowControllers] objectAtIndex:0]
firstTextView] frame].size];
        [printingView setPrintPanelAccessoryController:accessoryController];
        // We allow changing print parameters if not in "Wrap to Page" mode, where the
page setup settings are used
        [printPanel setOptions:[printPanel options] | NSPrintPanelShowsPaperSize |
NSPrintPanelShowsOrientation];
    }
    [printPanel addAccessoryController:accessoryController];

    return op;
}

- (NSPrintInfo *)printInfo {
    NSPrintInfo *printInfo = [super printInfo];
    if (!setUpPrintInfoDefaults) {
           setUpPrintInfoDefaults = YES;
           [printInfo setHorizontalPagination:NSFitPagination];
           [printInfo setHorizontallyCentered:NO];
           [printInfo setVerticallyCentered:NO];
```

```
            [printInfo setLeftMargin:72.0];
            [printInfo setRightMargin:72.0];
            [printInfo setTopMargin:72.0];
            [printInfo setBottomMargin:72.0];
    }
    return printInfo;
}

/* Toggles read-only state of the document
*/
- (IBAction)toggleReadOnly:(id)sender {
    [[self undoManager] registerUndoWithTarget:self selector:@selector(toggleReadOnly:)
object:nil];
    [[self undoManager] setActionName:[self isReadOnly] ?
        NSLocalizedString(@"Allow Editing", @"Menu item to make the current document
editable (not read-only)") :
        NSLocalizedString(@"Prevent Editing", @"Menu item to make the current document
read-only")];
    [self setReadOnly:![self isReadOnly]];
}

- (BOOL)toggleRichWillLoseInformation {
    NSInteger length = [textStorage length];
    NSRange range;
    NSDictionary *attrs;
    return ( ([self isRichText] // Only rich -> plain can lose information.
                && ((length > 0) // If the document contains characters and...
                        && (attrs = [textStorage attributesAtIndex:0
effectiveRange:&range])  // ...they have attributes...
                        && ((range.length < length) // ...which either are not the same
for the whole document...
                                || ![[self defaultTextAttributes:YES] isEqual:attrs]) // ...or
differ from the default, then...
                        )) // ...we will lose styling information.
                || [self hasDocumentProperties]); // We will also lose information if the
document has properties.
}

- (BOOL)hasMultiplePages {
    return hasMultiplePages;
}

- (void)setHasMultiplePages:(BOOL)flag {
    hasMultiplePages = flag;
}

- (IBAction)togglePageBreaks:(id)sender {
    [self setHasMultiplePages:![self hasMultiplePages]];
}

- (void)toggleHyphenation:(id)sender {
    float currentHyphenation = [self hyphenationFactor];
    [[[self undoManager] prepareWithInvocationTarget:self]
setHyphenationFactor:currentHyphenation];
    [self setHyphenationFactor:(currentHyphenation > 0.0) ? 0.0 : 0.9];       /* Toggle
between 0.0 and 0.9 */
}

/* Action method for the "Append '.txt' extension" button
*/
- (void)appendPlainTextExtensionChanged:(id)sender {
    NSSavePanel *panel = (NSSavePanel *)[sender window];
    [panel setAllowsOtherFileTypes:[sender state]];
    [panel setAllowedFileTypes:[sender state] ? [NSArray arrayWithObject:(NSString
*)kUTTypePlainText] : nil];
}

- (void)encodingPopupChanged:(NSPopUpButton *)popup {
    [self setEncodingForSaving:[[[popup selectedItem] representedObject]
```

```
unsignedIntegerValue]];
}

/* Menu validation: Arbitrary numbers to determine the state of the menu items whose
titles change. Speeds up the validation... Not zero. */
#define TagForFirst 42
#define TagForSecond 43

void validateToggleItem(NSMenuItem *aCell, BOOL useFirst, NSString *first, NSString
*second) {
    if (useFirst) {
        if ([aCell tag] != TagForFirst) {
            [aCell setTitleWithMnemonic:first];
            [aCell setTag:TagForFirst];
        }
    } else {
        if ([aCell tag] != TagForSecond) {
            [aCell setTitleWithMnemonic:second];
            [aCell setTag:TagForSecond];
        }
    }
}

/* Menu validation
*/
- (BOOL)validateMenuItem:(NSMenuItem *)aCell {
    SEL action = [aCell action];

    if (action == @selector(toggleReadOnly:)) {
           validateToggleItem(aCell, [self isReadOnly], NSLocalizedString(@"Allow
Editing", @"Menu item to make the current document editable (not read-only)"),
NSLocalizedString(@"Prevent Editing", @"Menu item to make the current document read-
only"));
           return YES;
    } else if (action == @selector(togglePageBreaks:)) {
        validateToggleItem(aCell, [self hasMultiplePages], NSLocalizedString(@"&Wrap to
Window", @"Menu item to cause text to be laid out to size of the window"),
NSLocalizedString(@"&Wrap to Page", @"Menu item to cause text to be laid out to the size
of the currently selected page type"));
           return YES;
    } else if (action == @selector(toggleHyphenation:)) {
        validateToggleItem(aCell, ([self hyphenationFactor] > 0.0), NSLocalizedString(@"Do
not Allow Hyphenation", @"Menu item to disallow hyphenation in the document"),
NSLocalizedString(@"Allow Hyphenation", @"Menu item to allow hyphenation in the
document"));
        if ([self isReadOnly]) return NO;
           return YES;
    }

    return [super validateMenuItem:aCell];
}

// For scripting. We already have a -textStorage method implemented above.
- (void)setTextStorage:(id)ts {
    // Warning, undo support can eat a lot of memory if a long text is changed frequently
    NSAttributedString *textStorageCopy = [[self textStorage] copy];
    [[self undoManager] registerUndoWithTarget:self selector:@selector(setTextStorage:)
object:textStorageCopy];
    [textStorageCopy release];

    // ts can actually be a string or an attributed string.
    if ([ts isKindOfClass:[NSAttributedString class]]) {
        [[self textStorage] replaceCharactersInRange:NSMakeRange(0, [[self textStorage]
length]) withAttributedString:ts];
    } else {
        [[self textStorage] replaceCharactersInRange:NSMakeRange(0, [[self textStorage]
length]) withString:ts];
    }
}
```

```
- (BOOL)revertToContentsOfURL:(NSURL *)url ofType:(NSString *)type error:(NSError
**)outError {
    BOOL success = [super revertToContentsOfURL:url ofType:type error:outError];
    if (success) {
        if (fileTypeToSet) {     // If we're reverting, NSDocument will set the file type
behind out backs. This enables restoring that type.
            [self setFileType:fileTypeToSet];
            [fileTypeToSet release];
            fileTypeToSet = nil;
        }
        [self setHasMultiplePages:hasMultiplePages];
        [[self windowControllers]
makeObjectsPerformSelector:@selector(setupTextViewForDocument)];
    }
    return success;
}

@end


/* Returns the default padding on the left/right edges of text views
*/
CGFloat defaultTextPadding(void) {
    static CGFloat padding = -1;
    if (padding < 0.0) {
        NSTextContainer *container = [[NSTextContainer alloc] init];
        padding = [container lineFragmentPadding];
        [container release];
    }
    return padding;
}

@implementation Document (TextEditNSDocumentOverrides)

+ (BOOL)autosavesInPlace {
    return YES;
}

+ (BOOL)canConcurrentlyReadDocumentsOfType:(NSString *)typeName {
    NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
    return !([workspace type:typeName conformsToType:(NSString *)kUTTypeHTML] ||
[workspace type:typeName conformsToType:(NSString *)kUTTypeWebArchive]);
}

- (void)makeWindowControllers {
    NSArray *myControllers = [self windowControllers];

    // If this document displaced a transient document, it will already have been assigned
a window controller. If that is not the case, create one.
    if ([myControllers count] == 0) {
        [self addWindowController:[[[DocumentWindowController allocWithZone:[self zone]]
init] autorelease]];
    }
}

/* One of the determinants of whether a file is locked is whether its type is one of our
writable types. However, the writable types are a function of whether the document
contains attachments. But whether we are locked cannot be a function of whether the
document contains attachments, because we won't be asked to redetermine autosaving safety
after an undo operation resulting from a cancel, so the document would continue to appear
locked if an image were dragged in and then cancel was pressed.  Therefore, we must use an
"ignoreTemporary" boolean to treat RTF as temporarily writable despite the attachments.
That's fine since -checkAutosavingSafetyAfterChangeAndReturnError: will perform this check
again with ignoreTemporary set to NO, and that method will be called when the operation is
done and undone, so no inconsistency results.
*/
- (NSArray *)writableTypesForSaveOperation:(NSSaveOperationType)saveOperation
ignoreTemporaryState:(BOOL)ignoreTemporary {
```

```
    NSMutableArray *outArray = [[[[self class] writableTypes] mutableCopy] autorelease];
    if (saveOperation == NSSaveAsOperation) {
           // Rich-text documents cannot be saved as plain text.
           if ([self isRichText]) {
               [outArray removeObject:(NSString *)kUTTypeText];
               [outArray removeObject:(NSString *)kUTTypePlainText];
           }

           // Documents that contain attachments can only be saved in formats that support
embedded graphics.
           if (!ignoreTemporary && [textStorage containsAttachments]) {
               [outArray setArray:[NSArray arrayWithObjects:(NSString *)kUTTypeRTFD,
(NSString *)kUTTypeWebArchive, nil]];
           }
    }
    return outArray;
}

- (NSArray *)writableTypesForSaveOperation:(NSSaveOperationType)saveOperation {
           return [self writableTypesForSaveOperation:saveOperation
ignoreTemporaryState:NO];
}

- (NSString *)fileNameExtensionForType:(NSString *)inTypeName saveOperation:
(NSSaveOperationType)inSaveOperation {
    /* We use kUTTypeText as our plain text type.  However, kUTTypeText is really a class
of types and therefore contains no preferred extension.  Therefore we must specify a
preferred extension, that of kUTTypePlainText. */
    if ([inTypeName isEqualToString:(NSString *)kUTTypeText]) return @"txt";
    return [super fileNameExtensionForType:inTypeName saveOperation:inSaveOperation];
}

/* When we save, we send a notification so that views that are currently coalescing undo
actions can break that. This is done for two reasons, one technical and the other HI
oriented.

Firstly, since the dirty state tracking is based on undo, for a coalesced set of changes
that span over a save operation, the changes that occur between the save and the next time
the undo coalescing stops will not mark the document as dirty. Secondly, allowing the user
to undo back to the precise point of a save is good UI.

In addition we overwrite this method as a way to tell that the document has been saved
successfully. If so, we set the save time parameters in the document.
*/
- (void)saveToURL:(NSURL *)absoluteURL ofType:(NSString *)typeName forSaveOperation:
(NSSaveOperationType)saveOperation completionHandler:(void (^)(NSError *error))handler {
    // Note that we do the breakUndoCoalescing call even during autosave, which means the
user's undo of long typing will take them back to the last spot an autosave occured. This
might seem confusing, and a more elaborate solution may be possible (cause an autosave
without having to breakUndoCoalescing), but since this change is coming late in Leopard,
we decided to go with the lower risk fix.
    [[self windowControllers] makeObjectsPerformSelector:@selector(breakUndoCoalescing)];
    [self performAsynchronousFileAccessUsingBlock:^(void (^fileAccessCompletionHandler)
(void) ) {
        currentSaveOperation = saveOperation;
        [super saveToURL:absoluteURL ofType:typeName forSaveOperation:saveOperation
completionHandler:^(NSError *error) {
            [self setEncodingForSaving:NoStringEncoding];   // This is set during
prepareSavePanel:, but should be cleared for future save operation without save panel
            fileAccessCompletionHandler();
            handler(error);
        }];
    }];

}

/* Indicate the types we know we can save safely asynchronously.
*/
- (BOOL)canAsynchronouslyWriteToURL:(NSURL *)url ofType:(NSString *)typeName
```

```
forSaveOperation:(NSSaveOperationType)saveOperation {
    return [[self class] canConcurrentlyReadDocumentsOfType:typeName];
}

- (NSError *)errorInTextEditDomainWithCode:(NSInteger)errorCode {
    switch (errorCode) {
        case TextEditSaveErrorWritableTypeRequired: {
            NSString *description, *recoverySuggestion;
            /* the document can't be saved in its original format, either because TextEdit
cannot write to the format, or TextEdit cannot write documents containing
                attachments to the format. */
            if ([textStorage containsAttachments]) {
                description = NSLocalizedString(@"Convert this document to RTFD format?",
                                                    @"Title of alert panel prompting the
user to convert to RTFD.");
                recoverySuggestion = NSLocalizedString(
                                                  @"Documents with graphics and attachments
will be saved using RTFD (RTF with graphics) format. RTFD documents are not compatible
with some applications. Convert anyway?",
                                                  @"Contents of alert panel prompting the
user to convert to RTFD.");
            } else {
                description = NSLocalizedString(@"Convert this document to RTF format?",
                                    @"Title of alert panel prompting the user to convert to
RTF.");
                recoverySuggestion = NSLocalizedString(@"This document must be converted
to RTF before it can be modified.",
                                                @"Contents of alert panel prompting the user to
convert to RTF.");
            }
            return [NSError errorWithDomain:TextEditErrorDomain
code:TextEditSaveErrorWritableTypeRequired userInfo:[NSDictionary
dictionaryWithObjectsAndKeys:
                description, NSLocalizedDescriptionKey,
                recoverySuggestion, NSLocalizedRecoverySuggestionErrorKey,
                [NSArray arrayWithObjects:
                    NSLocalizedString(@"Convert", @"Button choice that allows the user to
convert the document."),
                    NSLocalizedString(@"Cancel", @"Button choice that allows the user to
cancel."),
                    NSLocalizedString(@"Duplicate", @"Button choice that allows the user
to duplicate the document.")
                , nil], NSLocalizedRecoveryOptionsErrorKey,
                self, NSRecoveryAttempterErrorKey,
                nil]];
        }
        case TextEditSaveErrorConvertedDocument: {
            NSString *newFormatName = [textStorage containsAttachments] ?
NSLocalizedString(@"rich text with graphics (RTFD)", @"Rich text with graphics file format
name, displayed in alert")
                                        : NSLocalizedString(@"rich text", @"Rich text file
format name, displayed in alert");
            return [NSError errorWithDomain:TextEditErrorDomain
code:TextEditSaveErrorConvertedDocument userInfo:[NSDictionary
dictionaryWithObjectsAndKeys:
                        NSLocalizedString(
                            @"Are you sure you want to edit this document?",
                            @"Title of alert panel asking the user whether he wants to
edit a converted document."), NSLocalizedDescriptionKey,
                [NSString stringWithFormat:NSLocalizedString(@"This document was converted
from a format that TextEdit cannot save. It will be saved in %@ format.",
                            @"Contents of alert panel informing user that the document is
converted and cannot be written in its original format."), newFormatName],
NSLocalizedRecoverySuggestionErrorKey,
                [NSArray arrayWithObjects:
                    NSLocalizedString(@"Edit", @"Button choice that allows the user to
save the document."),
                    NSLocalizedString(@"Cancel", @"Button choice that allows the user to
cancel."),
```

```
                    NSLocalizedString(@"Duplicate", @"Button choice that allows the user
to duplicate the document.")
                    , nil], NSLocalizedRecoveryOptionsErrorKey,
                self, NSRecoveryAttempterErrorKey,
                nil]];
        }
        case TextEditSaveErrorLossyDocument: {
            return [NSError errorWithDomain:TextEditErrorDomain
code:TextEditSaveErrorLossyDocument userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
                NSLocalizedString(@"Are you sure you want to modify the document in
place?", @"Title of alert panel which brings up a warning about saving over the same
document"), NSLocalizedDescriptionKey,
                NSLocalizedString(@"Modifying the document in place might cause you to
lose some of the original formatting.  Would you like to duplicate the document first?",
@"Contents of alert panel informing user that they need to supply a new file name because
the save might be lossy"), NSLocalizedRecoverySuggestionErrorKey,
                [NSArray arrayWithObjects:
                    NSLocalizedString(@"Duplicate", @"Button choice that allows the user
to duplicate the document."),
                    NSLocalizedString(@"Cancel", @"Button choice that allows the user to
cancel."),
                    NSLocalizedString(@"Overwrite", @"Button choice allowing user to
overwrite the document."), nil], NSLocalizedRecoveryOptionsErrorKey,
                self, NSRecoveryAttempterErrorKey,
                nil]];
        }
        case TextEditSaveErrorEncodingInapplicable: {
            NSUInteger enc = [self encodingForSaving];
            if (enc == NoStringEncoding) enc = [self encoding];
            return [NSError errorWithDomain:TextEditErrorDomain
code:TextEditSaveErrorEncodingInapplicable userInfo:[NSDictionary
dictionaryWithObjectsAndKeys:
                        [NSString stringWithFormat:NSLocalizedString(@"This document can
no longer be saved using its original %@ encoding.", @"Title of alert panel informing user
that the file's string encoding needs to be changed."), [NSString
localizedNameOfStringEncoding:enc]], NSLocalizedDescriptionKey,
                    NSLocalizedString(@"Please choose another encoding (such as
UTF-8).", @"Subtitle of alert panel informing user that the file's string encoding needs
to be changed"), NSLocalizedRecoverySuggestionErrorKey,
                    NSLocalizedString(@"The specified text encoding isn\\U2019t
applicable.",
                        @"Failure reason stating that the text encoding is not
applicable."), NSLocalizedFailureReasonErrorKey,
                    [NSArray arrayWithObjects:
                        NSLocalizedString(@"OK", @"OK"),
                        NSLocalizedString(@"Cancel", @"Button choice that allows the
user to cancel."), nil], NSLocalizedRecoveryOptionsErrorKey,
                    self, NSRecoveryAttempterErrorKey,
                    nil]];
        }

    }
    return nil;
}

- (BOOL)checkAutosavingSafetyAfterChangeAndReturnError:(NSError **)outError {
    BOOL safe = YES;
          // it the document isn't saved, don't complain about limitations of its
supposed backing store.
          if ([self fileURL]) {
          if (![[self writableTypesForSaveOperation:NSSaveAsOperation] containsObject:
[self fileType]]) {
                    if (outError) *outError = [self
errorInTextEditDomainWithCode:TextEditSaveErrorWritableTypeRequired];
                    safe = NO;
                } else if (![self isRichText]) {
                    NSUInteger encoding = [self encoding];
                    if (encoding != NoStringEncoding && ![[textStorage string]
canBeConvertedToEncoding:encoding]) {
```

```
                        if (outError) *outError = [self
errorInTextEditDomainWithCode:TextEditSaveErrorEncodingInapplicable];
                        safe = NO;
                    }
                }
            }
    return safe;
}

- (void)updateChangeCount:(NSDocumentChangeType)change {
    NSError *error;

    // When a document is changed, it ceases to be transient.
    [self setTransient:NO];

            [super updateChangeCount:change];

    if (change == NSChangeDone || change == NSChangeRedone) {
        // If we don't have a file URL, we can change our backing store type without
consulting the user.
        // NSDocument will update the extension of our autosaving location.
        // If we don't do this, we won't be able to store images in autosaved untitled
documents.
        if (![self fileURL]) {
            if (![[self writableTypesForSaveOperation:NSSaveAsOperation] containsObject:
[self fileType]]) {
                [self setFileType:(NSString *)([textStorage containsAttachments] ?
kUTTypeRTFD : kUTTypeRTF)];
            }
        } else if (![self checkAutosavingSafetyAfterChangeAndReturnError:&error]) {
            void (^didRecoverBlock)(BOOL) = ^(BOOL didRecover) {
                if (!didRecover) {
                    if (change == NSChangeDone || change == NSChangeRedone) {
                        [[self undoManager] undo];
                    } else if (change == NSChangeUndone) {
                        [[self undoManager] redo];
                    }
                }
            };
            NSWindow *sheetWindow = [self windowForSheet];
            if (sheetWindow) {
                [self performActivityWithSynchronousWaiting:YES usingBlock:^(void
(^activityCompletionHandler)()) {
                [self presentError:error
                        modalForWindow:sheetWindow
                        delegate:self
                        didPresentSelector:@selector(didPresentErrorWithRecovery:block:)
                           contextInfo:Block_copy(^(BOOL didRecover) {
                               if (!didRecover) {
                                   if (change == NSChangeDone || change == NSChangeRedone)
{
                                       [[self undoManager] undo];
                                   } else if (change == NSChangeUndone) {
                                       [[self undoManager] redo];
                                   }
                               }
                               activityCompletionHandler();
                           })];
                }];
            } else {
                didRecoverBlock([self presentError:error]);
            }
        }
    }
}

- (NSString *)autosavingFileType {
    if (inDuplicate) {
        if (![[self writableTypesForSaveOperation:NSSaveAsOperation] containsObject:[self
```

```
fileType]])
            return (NSString *)([textStorage containsAttachments] ? kUTTypeRTFD :
kUTTypeRTF);
    }
    return [super autosavingFileType];
}

/* When we duplicate a document, we need to temporarily return the autosaving file type
for the
    resultant document.  Unfortunately, the only way to do this from a document subclass
appears
    to be to use a boolean indicator.
 */
- (NSDocument *)duplicateAndReturnError:(NSError **)outError {
    NSDocument *result;
    inDuplicate = YES;
    result = [super duplicateAndReturnError:outError];
    inDuplicate = NO;
    return result;
}

- (void)document:(NSDocument *)ignored didSave:(BOOL)didSave block:(void (^)(BOOL))block {
    block(didSave);
    Block_release(block);
}

- (void)didPresentErrorWithRecovery:(BOOL)didRecover block:(void (^)(BOOL))block {
    block(didRecover);
    Block_release(block);
}

- (void)attemptRecoveryFromError:(NSError *)error optionIndex:
(NSUInteger)recoveryOptionIndex delegate:(id)delegate didRecoverSelector:
(SEL)didRecoverSelector contextInfo:(void *)contextInfo {
    BOOL didRecover = NO;
    if ([[error domain] isEqualToString:TextEditErrorDomain]) {
        NSInteger errorCode = [error code];
        switch (errorCode) {
            case TextEditSaveErrorWritableTypeRequired:
            case TextEditSaveErrorConvertedDocument:
            case TextEditSaveErrorLossyDocument:
                // duplicate button - index 0 for lossy document errors and index 2
otherwise
                if (((errorCode == TextEditSaveErrorLossyDocument) && (recoveryOptionIndex
== 0)) ||
                    ((errorCode != TextEditSaveErrorLossyDocument) && (recoveryOptionIndex
== 2))) {
                    NSError *duplicateError;
                    NSDocument *duplicateDocument;
                    duplicateDocument = [self duplicateAndReturnError:&duplicateError];
                    if (!duplicateDocument) {
                        NSWindow *sheetWindow = [self windowForSheet];
                        if (sheetWindow) {
                            [delegate retain];
                            [self presentError:duplicateError
                                    modalForWindow:sheetWindow
                                    delegate:self

didPresentSelector:@selector(didPresentErrorWithRecovery:block:)
                                    contextInfo:Block_copy(^(BOOL didRecover) {
                                        [delegate release];
                                        ((void (*)(id, SEL, BOOL, void *))objc_msgSend)
(delegate, didRecoverSelector, NO, contextInfo);
                                    })];
                            return;
                        } else {
                            [self presentError:duplicateError];
                        }
                    }
```

```
                // save / overwrite button - index 2 for lossy document errors and 0
otherwise
                } else if (((errorCode == TextEditSaveErrorLossyDocument) &&
(recoveryOptionIndex == 2)) ||
                        ((errorCode != TextEditSaveErrorLossyDocument) && (recoveryOptionIndex
== 0))) {
                        if (![[self writableTypesForSaveOperation:NSSaveAsOperation]
containsObject:[self fileType]])
                            [self setFileType:(NSString *)([textStorage containsAttachments] ?
kUTTypeRTFD : kUTTypeRTF)];
                        [self setConverted:NO];
                        [self setLossy:NO];
                        didRecover = YES;
                    }
                    break;
                case TextEditSaveErrorEncodingInapplicable:
                    if (recoveryOptionIndex == 0) { // OK button
                        [delegate retain];
                        [self continueActivityUsingBlock:^(void) {
                        [self runModalSavePanelForSaveOperation:NSSaveOperation
                            delegate:self
                            didSaveSelector:@selector(document:didSave:block:)
                            contextInfo:Block_copy(^(BOOL didSave) {
                                [delegate release];
                                ((void (*)(id, SEL, BOOL, void *))objc_msgSend)(delegate,
didRecoverSelector, didSave, contextInfo);
                            })];
                        }];
                        return;
                    }
                    break;
            }
        }
        // call the delegate's didRecoverSelector
        ((void (*)(id, SEL, BOOL, void *))objc_msgSend)(delegate, didRecoverSelector,
didRecover, contextInfo);
    }

    /* Returns an object that represents the document to be written to file.
    */
    - (id)fileWrapperOfType:(NSString *)typeName error:(NSError **)outError {
        NSTextStorage *text = [self textStorage];
        NSRange range = NSMakeRange(0, [text length]);
        NSStringEncoding enc;

        NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithObjectsAndKeys:
                [NSValue valueWithSize:[self paperSize]], NSPaperSizeDocumentAttribute,
                [NSNumber numberWithInteger:[self isReadOnly] ? 1 : 0],
NSReadOnlyDocumentAttribute,
                [NSNumber numberWithFloat:[self hyphenationFactor]],
NSHyphenationFactorDocumentAttribute,
                [NSNumber numberWithDouble:[[self printInfo] leftMargin]],
NSLeftMarginDocumentAttribute,
                [NSNumber numberWithDouble:[[self printInfo] rightMargin]],
NSRightMarginDocumentAttribute,
                [NSNumber numberWithDouble:[[self printInfo] bottomMargin]],
NSBottomMarginDocumentAttribute,
                [NSNumber numberWithDouble:[[self printInfo] topMargin]],
NSTopMarginDocumentAttribute,
                [NSNumber numberWithInteger:[self hasMultiplePages] ? 1 : 0],
NSViewModeDocumentAttribute,
                [NSNumber numberWithBool:[self usesScreenFonts]],
NSUsesScreenFontsDocumentAttribute,
                nil];
        NSString *docType = nil;
        id val = nil; // temporary values

        NSSize size = [self viewSize];
        if (!NSEqualSizes(size, NSZeroSize)) {
```

```
            [dict setObject:[NSValue valueWithSize:size]
forKey:NSViewSizeDocumentAttribute];
    }

    // TextEdit knows how to save all these types, including their super-types. It does
not know how to save any of their potential subtypes. Hence, the conformance check is the
reverse of the usual pattern.
    NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
    // kUTTypePlainText also handles kUTTypeText and has to come before the other types so
we will use the least specialized type
    // For example, kUTTypeText is an ancestor of kUTTypeText and kUTTypeRTF but we should
use kUTTypeText because kUTTypeText is an ancestor of kUTTypeRTF.
    if ([workspace type:(NSString *)kUTTypePlainText conformsToType:typeName]) docType =
NSPlainTextDocumentType;
    else if ([workspace type:(NSString *)kUTTypeRTF conformsToType:typeName]) docType =
NSRTFTextDocumentType;
    else if ([workspace type:(NSString *)kUTTypeRTFD conformsToType:typeName]) docType =
NSRTFDTextDocumentType;
    else if ([workspace type:SimpleTextType conformsToType:typeName]) docType =
NSMacSimpleTextDocumentType;
    else if ([workspace type:Word97Type conformsToType:typeName]) docType =
NSDocFormatTextDocumentType;
    else if ([workspace type:Word2007Type conformsToType:typeName]) docType =
NSOfficeOpenXMLTextDocumentType;
    else if ([workspace type:Word2003XMLType conformsToType:typeName]) docType =
NSWordMLTextDocumentType;
    else if ([workspace type:OpenDocumentTextType conformsToType:typeName]) docType =
NSOpenDocumentTextDocumentType;
    else if ([workspace type:(NSString *)kUTTypeHTML conformsToType:typeName]) docType =
NSHTMLTextDocumentType;
    else if ([workspace type:(NSString *)kUTTypeWebArchive conformsToType:typeName])
docType = NSWebArchiveTextDocumentType;
    else [NSException raise:NSInvalidArgumentException format:@"%@ is not a recognized
document type.", typeName];

    if (docType) [dict setObject:docType forKey:NSDocumentTypeDocumentAttribute];
    if ([self hasMultiplePages] && ([self scaleFactor] != 1.0)) [dict setObject:[NSNumber
numberWithDouble:[self scaleFactor] * 100.0] forKey:NSViewZoomDocumentAttribute];
    if ((val = [self backgroundColor])) [dict setObject:val
forKey:NSBackgroundColorDocumentAttribute];

    if (docType == NSPlainTextDocumentType) {
        enc = [self encoding];
        if ((currentSaveOperation == NSSaveOperation || currentSaveOperation ==
NSSaveAsOperation) && (documentEncodingForSaving != NoStringEncoding)) {
            enc = documentEncodingForSaving;
        }
        if (enc == NoStringEncoding) enc = [self suggestedDocumentEncoding];
        [dict setObject:[NSNumber numberWithUnsignedInteger:enc]
forKey:NSCharacterEncodingDocumentAttribute];
    } else if (docType == NSHTMLTextDocumentType || docType ==
NSWebArchiveTextDocumentType) {
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        NSMutableArray *excludedElements = [NSMutableArray array];
        if (![defaults boolForKey:UseXHTMLDocType]) [excludedElements addObject:@"XML"];
        if (![defaults boolForKey:UseTransitionalDocType]) [excludedElements
addObjectsFromArray:[NSArray arrayWithObjects:@"APPLET", @"BASEFONT", @"CENTER", @"DIR",
@"FONT", @"ISINDEX", @"MENU", @"S", @"STRIKE", @"U", nil]];
        if (![defaults boolForKey:UseEmbeddedCSS]) {
            [excludedElements addObject:@"STYLE"];
            if (![defaults boolForKey:UseInlineCSS]) [excludedElements addObject:@"SPAN"];
        }
        if (![defaults boolForKey:PreserveWhitespace]) {
            [excludedElements addObject:@"Apple-converted-space"];
            [excludedElements addObject:@"Apple-converted-tab"];
            [excludedElements addObject:@"Apple-interchange-newline"];
        }
        [dict setObject:excludedElements forKey:NSExcludedElementsDocumentAttribute];
        [dict setObject:[defaults objectForKey:HTMLEncoding]
```

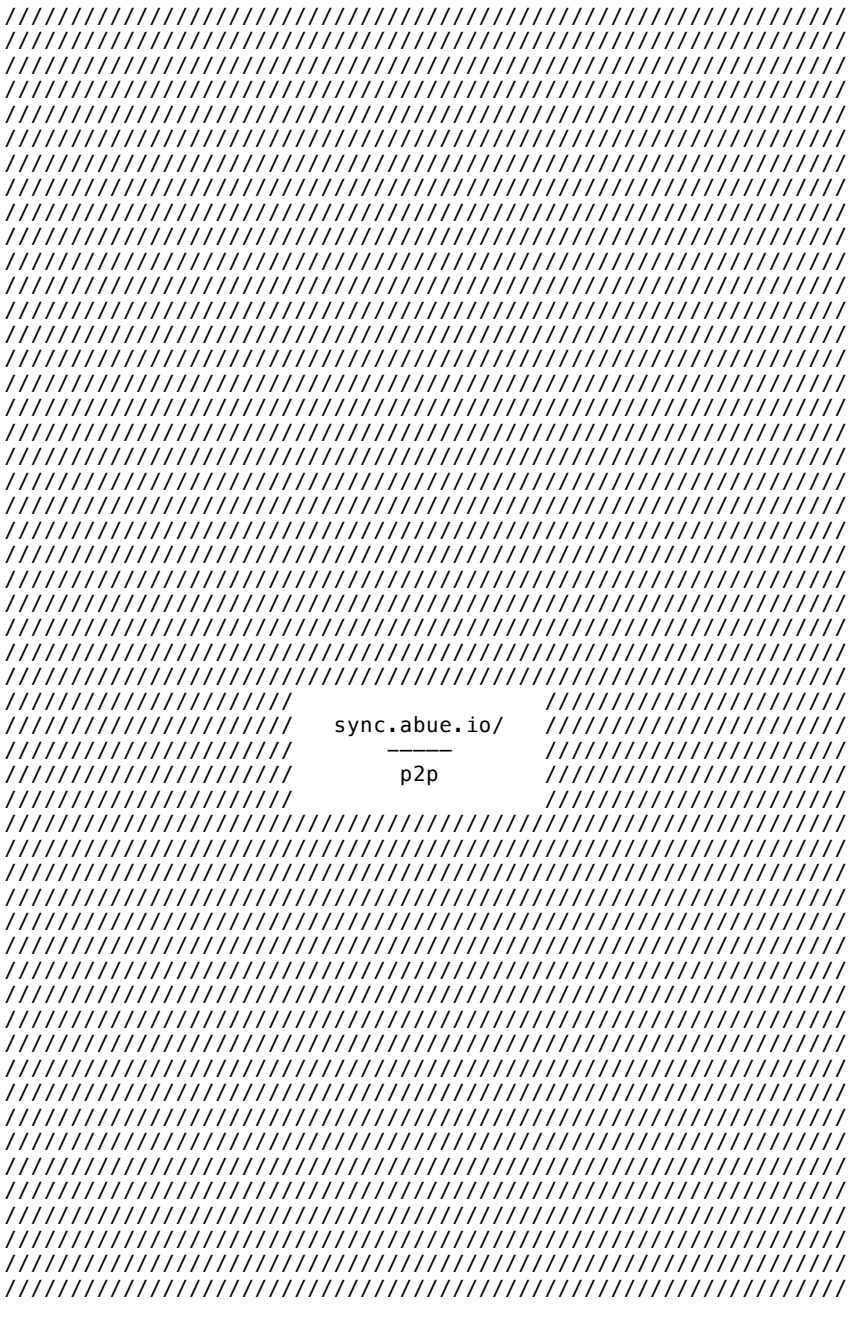
```
forKey:NSCharacterEncodingDocumentAttribute];
        [dict setObject:[NSNumber numberWithInteger:2]
forKey:NSPrefixSpacesDocumentAttribute];
    }

    // Set the text layout orientation for each page
    if ((val = [[[self windowControllers] objectAtIndex:0] layoutOrientationSections]))
[dict setObject:val forKey:NSTextLayoutSectionsAttribute];

    // Set the document properties, generically, going through key value coding
    for (NSString *property in [self knownDocumentProperties]) {
            id value = [self valueForKey:property];
            if (value && ![value isEqual:@""] && ![value isEqual:[NSArray array]]) [dict
setObject:value forKey:[[self documentPropertyToAttributeNameMappings]
objectForKey:property]];
    }

    NSFileWrapper *result = nil;
    if (docType == NSRTFDTextDocumentType || (docType == NSPlainTextDocumentType && ![self
isOpenedIgnoringRichText])) {     // We obtain a file wrapper from the text storage for
RTFD (to produce a directory), or for true plain-text documents (to write out encoding in
extended attributes)
        result = [text fileWrapperFromRange:range documentAttributes:dict
error:outError]; // returns NSFileWrapper
    } else {
            NSData *data = [text dataFromRange:range documentAttributes:dict
error:outError]; // returns NSData
            if (data) {
                result = [[[NSFileWrapper alloc] initRegularFileWithContents:data]
autorelease];
                if (!result && outError) *outError = [NSError
errorWithDomain:NSCocoaErrorDomain code:NSFileWriteUnknownError userInfo:nil];    //
Unlikely, but just in case we should generate an NSError
        }
    }
    if (result && docType == NSPlainTextDocumentType && (currentSaveOperation ==
NSSaveOperation || currentSaveOperation == NSSaveAsOperation)) {
        [self setEncoding:enc];
    }

    return result;
}

- (BOOL)checkAutosavingSafetyAndReturnError:(NSError **)outError {
    BOOL safe = YES;
    if (![super checkAutosavingSafetyAndReturnError:outError]) return NO;
    if ([self fileURL]) {
        // If the document is converted or lossy but can't be saved in its file type, we
will need to save it in a different location or duplicate it anyway.  Therefore, we should
tell the user that a writable type is required instead.
        if (![[self writableTypesForSaveOperation:NSSaveAsOperation
ignoreTemporaryState:YES] containsObject:[self fileType]]) {
            if (outError) *outError = [self
errorInTextEditDomainWithCode:TextEditSaveErrorWritableTypeRequired];
            safe = NO;
        } else if ([self isConverted]) {
            if (outError) *outError = [self
errorInTextEditDomainWithCode:TextEditSaveErrorConvertedDocument];
            safe = NO;
        } else if ([self isLossy]) {
            if (outError) *outError = [self
errorInTextEditDomainWithCode:TextEditSaveErrorLossyDocument];
            safe = NO;
        }
    }
    return safe;
}

/* For plain-text documents, we add our own accessory view for selecting encodings. The
```

```
plain text case does not require a format popup.
*/
- (BOOL)shouldRunSavePanelWithAccessoryView {
    return [self isRichText];
}

/* If the document is a converted version of a document that existed on disk, set the
default directory to the directory in which the source file (converted file) resided at
the time the document was converted. If the document is plain text, we additionally add an
encoding popup.
*/
- (BOOL)prepareSavePanel:(NSSavePanel *)savePanel {
    NSPopUpButton *encodingPopup;
    NSButton *extCheckbox;
    NSUInteger cnt;
    NSString *string;

    if (![self isRichText]) {
           BOOL addExt = [[NSUserDefaults standardUserDefaults]
boolForKey:AddExtensionToNewPlainTextFiles];
           // If no encoding, figure out which encoding should be default in encoding
popup, set as document encoding.
           string = [textStorage string];
           NSStringEncoding enc = [self encoding];
           [self setEncodingForSaving:(enc == NoStringEncoding || ![string
canBeConvertedToEncoding:enc]) ? [self suggestedDocumentEncoding] : enc];
        NSView *accessoryView = [[[NSDocumentController sharedDocumentController] class]
encodingAccessory:[self encodingForSaving] includeDefaultEntry:NO
encodingPopUp:&encodingPopup checkBox:&extCheckbox];
        accessoryView.translatesAutoresizingMaskIntoConstraints = NO;
           [savePanel setAccessoryView:accessoryView];

           // Set up the checkbox
           [extCheckbox setTitle:NSLocalizedString(@"If no extension is provided, use \
\U201c.txt\\U201d.", @"Checkbox indicating that if the user does not specify an extension
when saving a plain text file, .txt will be used")];
           [extCheckbox setToolTip:NSLocalizedString(@"Automatically append \\U201c.txt\
\U201d to the file name if no known file name extension is provided.", @"Tooltip for
checkbox indicating that if the user does not specify an extension when saving a plain
text file, .txt will be used")];
           [extCheckbox setState:addExt];
           [extCheckbox setAction:@selector(appendPlainTextExtensionChanged:)];
           [extCheckbox setTarget:self];
           if (addExt) {
               [savePanel setAllowedFileTypes:[NSArray arrayWithObject:(NSString
*)kUTTypePlainText]];
               [savePanel setAllowsOtherFileTypes:YES];
           } else {
            // NSDocument defaults to setting the allowedFileType to kUTTypePlainText,
which gives the fileName a ".txt" extension. We want don't want to append the extension
for Untitled documents.
           // First we clear out the allowedFileType that NSDocument set. We want to
allow anything, so we pass 'nil'. This will prevent NSSavePanel from appending an
extension.
            [savePanel setAllowedFileTypes:nil];
            // If this document was previously saved, use the URL's name.
            NSString *fileName;
            BOOL gotFileName = [[self fileURL] getResourceValue:&fileName
forKey:NSURLNameKey error:nil];
            // If the document has not yet been seaved, or we couldn't find the fileName,
then use the displayName.
            if (!gotFileName || fileName == nil) {
                fileName = [self displayName];
            }
            [savePanel setNameFieldStringValue:fileName];
        }

           // Further set up the encoding popup
           cnt = [encodingPopup numberOfItems];
```

```
            if (cnt * [string length] < 5000000) {       // Otherwise it's just too slow;
would be nice to make this more dynamic. With large docs and many encodings, the items
just won't be validated.
                while (cnt--) {   // No reason go backwards except to use one variable
instead of two
                 NSStringEncoding encoding = (NSStringEncoding)[[[encodingPopup
itemAtIndex:cnt] representedObject] unsignedIntegerValue];
                        // Hardwire some encodings known to allow any content
                        if ((encoding != NoStringEncoding) && (encoding !=
NSUnicodeStringEncoding) && (encoding != NSUTF8StringEncoding) && (encoding !=
NSNonLossyASCIIStringEncoding) && ![string canBeConvertedToEncoding:encoding]) {
                            [[encodingPopup itemAtIndex:cnt] setEnabled:NO];
                        }
                    }
            }
            [encodingPopup setAction:@selector(encodingPopupChanged:)];
            [encodingPopup setTarget:self];
    }

    return YES;
}

@end


/* Truncate string to no longer than truncationLength; should be > 10
*/
NSString *truncatedString(NSString *str, NSUInteger truncationLength) {
    NSUInteger len = [str length];
    if (len < truncationLength) return str;
    return [[str substringToIndex:truncationLength - 10]
stringByAppendingString:@"\u2026"];          // Unicode character 2026 is ellipsis
}
```

This text is part of the source code of macOS' build in word processor TextEdit, the very program every issue of sync is composed with. Document.m is one of its major source files, containing specifications for handling each documents general layout, as well as its scaling, view size, margins etc. Also a lot of the plain text character encoding and the whole print specifications are handled here, two of the most important factors for syncs visual appearance.

Reading this code even with minor knowledge one can notice the circuity, sometimes even messiness of its commentary (the lines starting with "/*" or "//"), pointing to its collective fabrication and layered writing process. Also, in search of the code for certain functions it becomes apparent that TextEdits source code is embedded into a software framework with code being implemented from various other sources – even the smallest specifications branching out into lines upon lines of text in multiple documents.


sync.abue.io/
p2p

180831ab_sync_34_source.pdf