Undo Architecture

Data Management

2009-07-27

Contents

Introduction to Undo Architecture

This topic describes how to record operations with `NSUndoManager`, so the user can reverse an operation's effect. Undo and redo operations and the `NSUndoManager` class are available on both iOS and Mac OS X.

Note: The `NSUndoManager` class was added to iOS in version 3.0.

You should read this document to learn how to use an undo manager in your application.

Organization of This Document

This document contains the following articles:

- "Undo Manager" (page 7) provides a conceptual overview of the object which records operations for undo and redo.
- "Registering Undo Operations" (page 9) describes how to add operations to the undo stack.
- "Performing Undo and Redo" (page 13) describes how to perform undo and redo operations.
- "Clearing the Undo Stack" (page 15) describes how to remove operations from the undo stack.
- "Setting Action Names" (page 17) describes how to provide custom names for the Undo and Redo menu items.
- "Using Undo in AppKit-Based Applications" (page 21) describes the Application Kit's additions to undo.
- "Using Undo Notifications" (page 19) describes synchronizing using notifications.

Undo Manager

This article provides a *conceptual* understanding of the basic properties and behavior of the undo manager. Practical, code-based examples are provided in later articles.

Overview

`NSUndoManager` is a general-purpose undo stack where clients can register callbacks to be invoked should an undo be requested. When you perform an action that changes the property values of an object (for example, by invoking a set accessor method), you can also register with an undo manager an operation that can reverse the action.

An undo manager collects all undo operations that occur within a single cycle of the run loop,so that performing an undo reverts all changes that occurred during the cycle. Also, when performing undo an undo manager saves the operations that were reverted so that you can redo the undos.

`NSUndoManager` is implemented as a class of the Foundation framework because executables other than applications might want to revert changes to their states. For example, you might have an interactive command-line tool with undo and redo commands; or there could be Distributed Object implementations that can revert operations "over the wire." However, users typically see undo and redo as application-level features. The Application Kit implements undo and redo in its `NSTextView` object and makes it easy to implement it in objects along the responder chain. For more on the role of the Application Kit in undo and redo, see "Using Undo in AppKit-Based Applications" (page 21).

Undo Operations and Groups

An **undo operation** is a method for reverting a change to an object, along with the arguments needed to revert the change. The operation specifies:

- The object to receive a message if an undo is requested.

  This may be the object that changed, or an object that owns the object that changed.
- The message to send.
- The arguments to pass with the message.

  Typically you need to pass at least one argument—the original value.

Because `NSUndoManager` also supports redo, these operations should typically be reversible. The method that's invoked during an undo operation should itself register an undo operation that will then serve as the redo action.

Undo operations are typically collected in **undo groups**, which represent whole revertible actions, and are stored on a stack. When an undo manager performs undo or redo, it is actually undoing or redoing an entire group of operations. For example, a user could change the type face and the font size of some text. An application might package both attribute-setting operations as a group, so when the user chooses Undo, both type face and font size are reverted. To undo a single operation, it must still be packaged in a group.

Redo operations and groups are simply undo operations stored on a separate stack (described below).

`NSUndoManager` normally creates undo groups automatically during the run loop. The first time it is asked to record an undo operation in the run loop, it creates a new group. Then, at the end of the loop, it closes the group. You can create additional, nested undo groups within these default groups using the `beginUndoGrouping` and `enableUndoRegistration` methods. You can also turn off the default grouping behavior using `setGroupsByEvent:`.

The Undo and Redo Stacks

Undo groups are stored on a stack, with the oldest groups at the bottom and the newest at the top. The undo stack is unlimited by default, but you can restrict it to a maximum number of groups using the `setLevelsOfUndo:` method. When the stack exceeds the maximum, the oldest undo groups are dropped from the bottom.

Initially, both stacks are empty. Recording undo operations adds to the undo stack, but the redo stack remains empty until undo is performed. Performing undo causes the reverting operations in the latest group to be applied to their objects. Since these operations cause changes to the objects' states, the objects presumably register new operations with the undo manager, this time in the reverse direction from the original operations. Since the undo manager is in the process of performing undo, it records these operations as redo operations on the redo stack. Consecutive undos add to the redo stack. Subsequent redo operations pull the operations off the redo stack, apply them to the objects, and push them back onto the undo stack.

The redo stack's contents last as long as undo and redo are performed successively. However, because applying a new change to an object invalidates the previous changes, as soon as a new undo operation is registered, any existing redo stack is cleared. This prevents redo from returning objects to an inappropriate prior state. You can check for the ability to undo and redo with the `canUndo` and `canRedo` methods.

Registering Undo Operations

This article describes the two ways you register an undo operation with an undo manager.

Overview

To add an undo operation to the undo stack, you must register it with the object that performs the undo operation. `NSUndoManager` supports two ways to register undo operations:

- "Simple undo" based on a simple selector with a single object argument.

  Using this approach, when an object changes, the object itself (or another object acting on its behalf) registers the change with the undo manager, passing an argument that holds the attributes of the object prior to the change. (This may often be an `NSDictionary` object, but it can be any object.) Performing the undo then involves resetting the object with these attributes.
- "Invocation-based undo" which uses an `NSInvocation` object.

  Because this approach uses an invocation object, it can use a method that takes any number and type of arguments. Invocation-based undo is useful for registering specific state-changing methods, such as a `setWidth:height:` method.

In most applications a single instance of `NSUndoManager` belongs to an object that contains or manages other objects. This is particularly the case with desktop document-based applications, where each `NSDocument` object is responsible for all undo and redo operations for a document. An object such as this is often called the undo manager's client. Each client object has its own `NSUndoManager`. The client claims exclusive right to alter its undoable objects so that it can record undo operations for all changes. In the specific case of documents, this scheme keeps each pair of undo and redo stacks separate so that when an undo is performed, it applies to the focal document in the application (typically the one displayed in the key window). It also relieves the individual objects in a document from having to know the identity of their undo manager or from having to track changes to themselves.

However, an object that is changed can have its own undo manager and perform its own undo and redo operations. For example, you could have a custom view that displays images dragged into it; with each successful drag operation, it registers a new undo group. If the view is then selected (that is, made first responder) and the Undo command applied, the previously displayed image would be redisplayed.

Many of the following code examples register the same method for undo and redo operations. Although that approach is convenient when undo and redo toggle a simple data value between two states, you do not have to register the same method for undo and redo operations. When what is being undone and redone is more complex—for example, the insertion and deletion of objects in an array—you could call a pair of methods in alternation, once which knows how to construct a state and the other which knows how to deconstruct it. When you register a selector as an undo action, it causes the method identified by that selector to be called when the user requests that what occurred in a given context be undone. The method does not have to be the same one in which registration occurred. And the method itself could register its own undo selector. The important point is that, when registering undo and redo operations, both the data state and the selector can be varied.

Simple Undo

To record a simple undo operation, you need only invoke `registerUndoWithTarget:selector:object:`, giving the object to be sent the undo operation selector, the selector to invoke, and an argument to pass with that message. The target object may not be the actual object whose state is changing; instead, it may be the client object, a document or container that holds many undoable objects. The argument is an object that captures the state of the object before the change is made, as illustrated in the following example:

```
- (void)setMyObjectTitle:(NSString *)newTitle {

    NSString *currentTitle = [myObject title];
    if (newTitle != currentTitle) {
        [undoManager registerUndoWithTarget:self
                selector:@selector(setMyObjectTitle:)
                object:currentTitle];
        [undoManager setActionName:NSLocalizedString(@"Title Change", @"title
undo")];
        [myObject setTitle:newTitle];
    }
}
```

In an undo operation, `setMyObjectTitle:` is invoked with the previous value. Notice that this will again invoke the `registerUndoWithTarget:selector:object:` method—in this case with the "new" value of `myObject`'s title. Since the undo manager is in the process of undoing, it is recorded as a *redo* operation.

Invocation-Based Undo

For other changes involving specific methods or arguments that are not objects, you can use invocation-based undo, which records an actual message to revert the target object's state. As with simple undo, you record a message that reverts the object to its state before the change. However, in this case you do so by sending the message directly to the undo manager, after preparing it with a special message (`prepareWithInvocationTarget:`) to note the target, as in this example:

```
- (void)setMyObjectWidth:(CGFloat)newWidth height:(CGFloat)newHeight{

    float currentWidth = [myObject size].width;
    float currentHeight = [myObject size].height;
    if ((newWidth != currentWidth) && (newHeight != currentHeight) {
        [[undoManager prepareWithInvocationTarget:self]
                setMyObjectWidth:currentWidth height:currentHeight];
        [undoManager setActionName:NSLocalizedString(@"Size Change", @"size
undo")];
        [myObject setSize:NSMakeSize(newWidth, newHeight)];
    }
}
```

The `prepareWithInvocationTarget:` method records the argument as the target of the undo operation about to be established. Following this, you send the message that reverts the target's state—in this case, `setMyObjectWidth:height:`. Because `NSUndoManager` does not respond to this method, `forwardInvocation:` is invoked, which `NSUndoManager` implements to record the `NSInvocation` object containing the target, selector, and all arguments. Performing undo thus results in *`self`* being sent a `setMyObjectWidth:height:` message with the original values.

Performing Undo and Redo

Performing undo and redo is usually as simple as sending `undo` and `redo` messages to an `NSUndoManager` object. The `undo` message closes the last open undo group and then applies all the undo operations in that group (recording any undo operations as redo operations instead). The `redo` message likewise applies all the redo operations on the top redo group.

The `undo` method is intended for undoing top-level groups, and should not be used for nested undo groups. If any unclosed, nested undo groups are on the stack when `undo` is invoked, it raises an exception. To undo nested groups, you must explicitly close the group with an `endUndoGrouping` message, then use `undoNestedGroup` to undo it. Note also that if you turn off automatic grouping by event with `setGroupsByEvent:`, you must explicitly close the current undo group with `endUndoGrouping` before invoking either undo method.

Clearing the Undo Stack

If you are using undo managers in a reference-counted environment, you have to be careful about issues related to memory management. An `NSUndoManager` object does not retain the targets of undo operations. The client—the object performing undo operations—typically owns the undo manager, so if the undo manager in turn retained its target this would frequently create a retain cycle. This means, though, that an undo manager may potentially hold a reference to an object that has been deallocated. If a target object has been deallocated and an undo message is sent to it, this results in a runtime exception.

To guard against this, you must take care to clear undo operations for targets that are being deallocated. You typically do this in one of three ways, depending on the configuration of the client:

- The client is the exclusive owner of the undo manager and the target of all undo operations.

  In this case the client can simply release the undo manager in its `dealloc` method.

- The client shares the undo manager with other clients.

  To handle this the client should send `removeAllActionsWithTarget:` (passing `self` as the argument) to the undo manager before releasing it in its `dealloc` method.

- The client registers objects other than itself for undo operations.

  Here either the client must watch for the other objects being deallocated in order to send `removeAllActionsWithTarget:`, or the other objects must do so themselves when deallocated (which requires that they have a reference to the undo manager). This is likely to be needed with invocation-based undo.

In a more general sense, it sometimes makes sense to clear all undo and redo operations. Some applications might want to do this when saving a document, for example. To this end, `NSUndoManager` defines the `removeAllActions` method, which clears both stacks.

Setting Action Names

You can use the `NSUndoManager` method `setActionName:` to qualify the Undo and Redo command titles in the Edit menu. You pass the string you want appended to "Undo" and "Redo" in the menu items when the *current* undo group is at the top of the undo and redo stacks. Because the name is applied to the current operation, you should typically set the name at the same time as registering the operation to ensure that the two are kept in sync.

```
- (void)setBookTitle:(NSString *)newTitle {
    [undoManager registerUndoWithTarget:self
                 selector:@selector(setBookTitle:)
                 object:[book title]];
    [book setTitle:newTitle];
    [undoManager setActionName:@"Title Change"];
}
```

Consider, for example, a graphics application that allows users to add a circle, fill it with a color, and delete it. With `setActionName:`, you could set the name of each action to "Add Circle," "Fill," and "Delete." After each action, the Undo menu item title is set to "Undo Add Circle," "Undo Fill," and "Undo Delete" respectively.

`NSUndoManager` automatically localizes the "Undo" and "Redo" portion of the command titles, but merely appends the action name to them. You should localize the action names yourself. If you want to further customize how these titles are localized, you can create a subclass of `NSUndoManager` and override `undoMenuTitleForUndoActionName:` and `redoMenuTitleForUndoActionName:`.

Using Undo Notifications

An `NSUndoManager` regularly posts checkpoint notifications to synchronize the inclusion of undo operations in undo groups. Objects sometimes delay performing changes, for various reasons. This means they may also delay registering undo operations for those changes. Because `NSUndoManager` collects individual operations into groups, it must be sure to synchronize its client with the creation of these groups so that operations are entered into the proper undo groups. To this end, whenever an undo manager opens or closes a new undo group (except when it opens a top-level group), it posts an `NSUndoManagerCheckpointNotification` so observers can apply their pending undo operations to the group in effect. The undo manager's client should register itself as an observer for this notification and record undo operations for all pending changes upon receiving it.

`NSUndoManager` also posts a number of other notifications at specific intervals: when a group is created, when a group is closed, and just before and just after both undo and redo operations.

Using Undo in AppKit-Based Applications

The Application Kit supplements the behavior of `NSUndoManager` in several ways:

- It offers default undo and redo behavior in text.
- It includes APIs for managing the action names that appear with "Undo" and "Redo" in an application's menu.
- It establishes a framework for the distribution and selection of undo managers in an application

Undo and the Responder Chain

As application can have one or more undo clients—objects that register and perform undo operations in their local contexts. Each of these objects has its own `NSUndoManager` object and the associated undo and redo stacks. One example of this scenario involves custom views, each a client of an undo manager. For example, you could have a window with two custom views; each view can display text in changeable attributes (such as font, color, and size) and users can undo (or redo) each change to any attribute in either of the views. `NSResponder` and `NSWindow` define methods to help you control the context of undo operations within the view hierarchy.

`NSResponder` declares the `undoManager` method for most objects that inherit from it (namely, windows and views). When the first responder of an application receives an `undo` or `redo` message, `NSResponder` goes up the responder chain looking for a next responder that returns an `NSUndoManager` object from `undoManager`. Any returned undo manager is used for the undo or redo operation.

If the `undoManager` message wends its way up the responder chain to the window, the `NSWindow` object queries its delegates with `windowWillReturnUndoManager:` to see if the delegate has an undo manager. If the delegate does not implement this method, the window creates an `NSUndoManager` object for the window and all its views.

Document-based applications often make their `NSDocument` objects the delegates of their windows and have them respond to the `windowWillReturnUndoManager:` message by returning the undo manager used for the document. These applications can also make each `NSWindowController` object the delegate of its window—the window controller implements `windowWillReturnUndoManager:` to get the undo manager from its document and return it:

```
return [[self document] undoManager];
```

NSTextView

Instances of `NSTextView` provide undo and redo behavior. This is an optional feature, and you must make sure that when you create the text view either you select the appropriate check box in Interface Builder, or send it `setAllowsUndo:` with an argument of `YES`. If you want a text view to use its own undo manager (and not the window's), you provide a delegate for the text view; the delegate can then return an instance of `NSUndoManager` from the `undoManagerForTextView:` delegate method.

The default undo and redo behavior applies to text fields and text in cells as long as the field or cell is the first responder (that is, the focus of keyboard actions). Once the insertion point leaves the field or cell, prior operations cannot be undone.

Using Undo on iPhone

UIKit supplements the behavior of the `NSUndoManager` class by establishing a framework for the distribution and selection of undo managers in an application.

Enabling Undo

Applications don't support undo by default. You need to tell the application that you want it to respond to shake events as an edit request:

```
application.applicationSupportsShakeToEdit = YES;
```

You typically set this property when the application starts up, in the application delegate's `applicationDidFinishLaunching:` or `application:didFinishLaunchingWithOptions:` methods.

Undo and the Responder Chain

As application can have one or more undo clients—objects that register and perform undo operations in their local contexts. Each of these objects has its own `NSUndoManager` object and the associated undo and redo stacks.

One example of this scenario involves custom views, each a client of an undo manager. For example, you could have a window with two custom views; each view can display text in changeable attributes (such as font, color, and size) and users can undo (or redo) each change to any attribute in either of the views. `UIResponder` helps you control the context of undo operations within the view hierarchy.

The `UIResponder` class declares the `undoManager` property through which objects that inherit from it (in particular views, and view controllers) can provide an undo manager to the framework. When the application receives an undo event, `UIResponder` goes up the responder chain (starting with the first responder) looking for a responder that returns an `NSUndoManager` object from `undoManager`. The first undo manager that is found is used for the undo or redo operation.

Important: The `UIResponder` class declares the `undoManager` property as read-only. Thus in your subclass of `UIView` or `UIViewController` (or subclasses of these classes), you must override this declaration to make the property writable. For example:

```
@property (retain) NSUndoManager *undoManager; // readwrite is the default
```

Then synthesize the redeclared property in the implementation file. If you redeclare the property, you should refrain from calling the superclass implementation of `setUndoManager:`. Alternatively, you can override the getter accessor for `undoManager` and return your own `NSUndoManager` object.

UIKit automatically creates a suitable alert panel for you based on the existence and state of the undo manger. If the user chooses to perform an undo or redo operation, the undo manager is sent an `undo` or `redo` message as appropriate.

You typically provide the undo manager in a view controller (see "Design Patterns" (page 24)).If you want a view controller to provide an undo manager, the view controller must be willing to become first responder, and must become first responder when its view appears. Conversely, it should resign first responder when its view disappears. It does this by implementing the following code:

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}
```

Design Patterns

In an iPhone application there are a number of patterns, conventions and constraints that come into play when considering undo support:

- Users interact with a screenful of information at a time.

  Users navigate from one screen to another. Each screen contains a different set of information, and is managed by a different view controller.

- There are clearly-defined editing modes, and saving state is implicit.

  Users often enter an editing mode by tapping an Edit button. In edit mode, they are frequently given the option to cancel the changes they've made. When the user taps Done, any changes they have made are considered to be committed—the user doesn't have to select a Save menu item, for example.

- Undo and redo messages are sent directly to the undo manager.

  You are not provided with undo and redo methods that you can override.

- Memory on devices is constrained.

You need to avoid consuming too much memory, otherwise your application may be terminated.

When supporting undo, you should heed these design patterns, conventions, and constraints.

It doesn't necessarily make sense to support undo pervasively. The user's expectation is that once an action has been taken, it is irreversible. Moreover, the context is important. Consider an application that displays a list of books, and allows you to navigate to a detail view that in turn allows you to edit individual properties of the book (such as its title, author, and copyright date). You might create a new book from the list screen, navigate between two other screens to edit its properties, then navigate back to the original list. It might seem peculiar if an undo operation in the list view undid a change to the author's name that was made two screens away rather than deleting the entire book.

There's also the issue of memory management. Each undo action requires that the data to perform the undo be kept in memory. In some applications, this may become a significant overhead. In general, it's best if you try to keep memory footprint to a minimum. If your application supports editing modes, it may be appropriate to create an undo manager only when the user enters editing mode, and to dispose of the undo manager when the user leaves the mode.

Because undo and redo messages are sent directly to the undo manager, you should register as an observer of the undo manager's change notifications. In the notification callbacks, you can propagate the change to the user interface as necessary.

Because the user may navigate between different screens during an editing operation, you may need to remind them what it is they're undoing. It is useful to provide undo action titles so that it's clear to the user what effect the undo or redo operation will have.

The following example illustrates how you might respond to change in editing state in a subclass of `UITableView`. If editing begins, you create an undo manager to track changes. You also register as an observer of undo manager change notifications, so that if an undo or redo operation is performed, the table view can be reloaded. When editing ends, de-register from the notification center and remove the undo manager.

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {

    [super setEditing:editing animated:animated];

    NSNotificationCenter *dnc = [NSNotificationCenter defaultCenter];

    if (editing) {
        NSUndoManager *anUndoManager = [[NSUndoManager alloc] init];
        self.undoManager = anUndoManager;
        [undoManager setLevelsOfUndo:3];
        [anUndoManager release];

        [dnc addObserver:self selector:@selector(undoManagerDidUndo:)
                    name:NSUndoManagerDidUndoChangeNotification
object:undoManager];
        [dnc addObserver:self selector:@selector(undoManagerDidRedo:)
                    name:NSUndoManagerDidRedoChangeNotification
object:undoManager];
    }
    else {
        [dnc removeObserver:self];
        self.undoManager = nil;
    }
}
```

Document Revision History

This table describes the changes to *Undo Architecture*.

Date	Notes
2009-07-27	Added supplementary information plus links to Cocoa Core Competencies.
2009-05-16	Corrected typographical errors.
2009-03-13	First release for iOS.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.