EOUndoManager

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOControl/EOUndoManager.h

Class Description

EOUndoManager is a general purpose recorder of operations for undo and redo. You register an undo operation by specifying the object that's changing (or the owner of that object), along with a method to invoke to revert its state, and the arguments for that method. EOUndoManager automatically groups all operations within a single cycle of the run loop, so that performing an undo reverts all changes that occurred during the loop. Also, when performing undo an EOUndoManager saves the operations reverted so that you can redo the undos.

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 (for example, its state before the change). Undo operations are typically collected into *undo groups*, which represent whole undoable actions. *Redo operations* and groups are simply undo operations stored on a separate stack (described below). When an EOUndoManager performs undo or redo, it's actually undoing or redoing an entire group of operations. To undo a single operation, it must be packaged in a group.

EOUndoManager normally creates undo groups automatically during the run loop. The first time it's 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 **endUndoGrouping** 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 undo 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 EOUndoManager, this time in the reverse direction from the original operations. Since the EOUndoManager 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 and apply them to the objects.

The redo stack's contents last as long as undo and redo is performed. However, because applying a new change to objects invalidates the previous changes, as soon as a new undo operation is registered, the 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

EOUndoManager supports two types of undo operations: one based on a simple selector with a single object argument, and one based on a general NSInvocation (which allows any number and type of arguments). The first method is commonly used by EOEditingContext for changes to enterprise objects. When an object changes, the EOEditingContext records a simple undo operation with an NSDictionary containing the old property values of the object. Performing undo then applies this object snapshot via the key-value coding protocol's **takeValues:forKeys:** method. Invocation-based undo is useful for undoing specific state-changing methods, such as a document object's **setFont:color:**. This more general undo operation is useful for already-defined methods, especially when their arguments aren't objects.

Regardless of the type of operations recorded, a single instance of EOUndoManager typically belongs to a single document or container of objects, called the EOUndoManager's *client*. Each EOEditingContext in an application, for example, has its own private EOUndoManager. This 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 from having to know the identity of their EOUndoManager.

In order to use undo effectively, either the client must claim exclusive right to alter its undoable objects—in order to record undo operations for all changes—or the objects themselves must participate in recording their changes. The first case is exemplified by a text document that holds a private NSTextView, handling all text operations by registering undo operations and forwarding the change to the NSTextView. For the second case, the **willChange** method defined by Enterprise Objects Framework allows any object to notify observers that it's about to change. EOEditingContexts, being containers for enterprise objects, receive these change notifications and record undo operations (among many other things). Even in this case, interaction with the EOUndoManager is handled exclusively by the container object.

Simple Undo

To record a simple undo operation, you need only invoke **registerUndoWithTarget:selector:arg:**, 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 is rarely the actual object whose state is changing; instead, it's the client object, a document or container that holds many undoable objects. An object like EOEditingContext, for example,

can record an undo operation for **insertObject:** by registering a **deleteObject:** message with the object inserted (**undoManager** is an instance variable):

```
[undoManager registerUndoWithTarget:self selector:@selector(deleteObject:)
    arg:anObject];
```

An update might be recorded for undo like so:

```
NSDictionary *updateDict = [NSDictionary dictionaryWithObjectsAndKeys:anObject,
    @"object", [anObject snapshot], @"snapshot"];

[undoManager registerUndoWithTarget:self
    selector:@selector(revertUpdate:)
    arg:updateDict];
```

This fragment is likely to be executed as a result of **anObject** invoking the standard **willChange** method, which announces that the object's state is going to change. Since it hasn't changed yet, the state can be recorded for later undo. This fragment, then, registers the client (**self**) to be sent a **revertUpdate:** message with the object and its old state when undo is performed. The old values are retrieved with a **snapshot** message. **revertUpdate:** can be implemented to pass the old state back to the object:

```
- (void)revertUpdate:(NSDictionary *)updateDict
{
    [[updateDict objectForKey:@"object"]
        udpateFromSnapshot:[updateDict objectForKey:@"snapshot"]];
    return;
}
```

Both **snapshot** and **updateFromSnapshot:** are methods added to NSObject by the Framework. See the NSObject Additions specification for more information.

Invocation-Based Undo

For other changes involving specific methods or arguments that aren't 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 EOUndoManager, after preparing it with a special message to note the target:

```
[[myUndoManager prepareWithInvocationTarget:textObject]
    setFont:[textObject font] color:[textObject textColor]];
[textObject setFont:newFont color:newColor];
```

prepareWithInvocationTarget: records the argument as the target of the undo operation about to be established. Following this, you send the message that will revert the target's state—in this case, **setFont:color:**. Because EOUndoManager doesn't respond to this method, **forwardInvocation:** is invoked, which EOUndoManager implements to record the NSInvocation containing the target, selector, and all arguments. Performing undo later results in **textObject** being sent a **setFont:color:** message with the old values.

Performing Undo and Redo

Performing undo and redo is usually as simple as sending **undo** and **redo** messages to the EOUndoManager. **undo** closes the last open undo group and then applies all of the undo operations in that group (recording any undo operations as redo operations instead). **redo** likewise applies all of the redo operations on the top redo group.

undo is intended for undoing top-level groups, and shouldn't 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 use 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.

Cleaning the Undo Stack

EOUndoManager doesn't retain the targets of undo operations, for several reasons. Foremost is that the client—the object registering operations—typically owns the EOUndoManager, so retaining it would create cycles. The EOUndoManager does contain references to the targets of undo operations, however, which it uses to send undo messages when undo is performed. If a target object has been deallocated, this will cause errors.

To remedy this, the client must take care to clear undo operations for targets that are being deallocated. This typically occurs in one of three ways:

- The client is the exclusive owner of the EOUndoManager and the target of all undo operations. In this case the client can simply release the EOUndoManager in its **dealloc** method.
- The client shares the EOUndoManager with other clients. To handle this the client should send **forgetAllWithTarget:** to the EOUndoManager 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 **forgetAllWithTarget:**, or the other objects must do so themselves when deallocated (which requires that they have a reference to the EOUndoManager). 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, EOUndoManager defines the **forgetAll** method, which clears both stacks.

Undo Checkpoint Notifications

Objects sometimes delay performing changes, for various reasons. This means they may also delay registering undo operations for those changes. Because EOUndoManager 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 EOUndoManager opens or closes a new undo

group (except when it opens a top-level group), it posts an EOUndoManagerCheckpointNotification so that observers can apply their pending undo operations to the group in effect. The EOUndoManager's client should register itself as an observer for this notification and record undo operations for all pending changes upon receiving it.

Method Types

Registering undo operations	– registerUndoWithTarget:selector:arg: – prepareWithInvocationTarget: – forwardInvocation:
Checking undo ability	– canUndo – canRedo
Performing undo and redo	– undo – undoNestedGroup – redo
Limiting the undo stack	– setLevelsOfUndo: – levelsOfUndo
Creating undo groups	– beginUndoGrouping – endUndoGrouping – setGroupsByEvent: – groupsByEvent
Disabling undo	– disableUndoRegistration – reenableUndoRegistration
Checking whether undo or redo is being performed	– isUndoing – isRedoing
Clearing undo operations	– forgetAll – forgetAllWithTarget:

Instance Methods

beginUndoGrouping

– (void)**beginUndoGrouping**

Marks the beginning of an undo group. All individual undo operations before a subsequent **endUndoGrouping** message are grouped together and reversed by a later **undo** message. Undo groups can be nested, thus providing functionality similar to nested transactions.

This method posts an EOUndoManagerCheckpointNotification.

canRedo

– (BOOL)**canRedo**

Returns YES if the receiver has any actions to redo, NO if it doesn't.

Because any undo operation registered clears the redo stack, this method posts an EOUndoManagerCheckpointNotification to allow clients to apply their pending operations before testing the redo stack.

See also: – **canUndo**, – **redo**

canUndo

– (BOOL)**canUndo**

Returns YES if the receiver has any actions to undo, NO if it doesn't. This does *not* mean that you can safely invoke **undo** or **undoNestedGroup**; you may have to close open undo groups first.

See also: – **canRedo**, – **endUndoGrouping**, – **registerUndoWithTarget:selector:arg:**

disableUndoRegistration

– (void)**disableUndoRegistration**

Disables the recording of undo operations, whether by **registerUndoWithTarget:selector:arg:** or by invocation-based undo. This method can be invoked multiple times; **reenableUndoRegistration** must be invoked an equal number of times to actually reenable undo registration.

endUndoGrouping

– (void)**endUndoGrouping**

Marks the end of an undo group. All individual undo operations back to the matching **beginUndoGrouping** message are grouped together and reversed by a later **undo** or **undoNestedGroup** message. Undo groups can be nested, thus providing functionality similar to nested transactions. Raises an NSInternalInconsistencyException if there's no **beginUndoGrouping** message in effect.

This method posts an EOUndoManagerCheckpointNotification.

See also: – **levelsOfUndo**

forgetAll

– (void)**forgetAll**

Clears the undo and redo stacks and reenables the receiver.

See also: – **reenableUndoRegistration**, – **forgetAllWithTarget:**

forgetAllWithTarget:

– (void)**forgetAllWithTarget:**(id)*target*

Clears the undo and redo stacks of all operations involving *target* as the recipient of the undo message. Doesn't reenable the receiver if it's disabled. An object that shares an EOUndoManager with other clients should invoke this message in its implementation of **dealloc**.

See also: – **reenableUndoRegistration**, – **forgetAll**

forwardInvocation:

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*

Overrides NSObject's implementation to record *anInvocation* as an undo operation. Also clears the redo stack. Raises an NSInternalInconsistencyException if **prepareWithInvocationTarget:** wasn't invoked before this method; this method then clears the prepared invocation target. See "Invocation-Based Undo" in the class description for more information.

Raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **registerUndoWithTarget:selector:arg:**, – **groupsByEvent**

groupsByEvent

– (BOOL)**groupsByEvent**

Returns YES if the receiver automatically creates undo groups around each pass of the run loop, NO if it doesn't. The default is YES.

See also: – **beginUndoGrouping**, – **setGroupsByEvent:**

isUndoing

– (BOOL)**isUndoing**

Returns YES if the receiver is in the process of performing its **undo** or **undoNestedGroup** method, NO otherwise.

See also: – **isRedoing**

isRedoing

– (BOOL)**isRedoing**

Returns YES if the receiver is in the process of performing its **redo** method, NO otherwise.

See also: – **isUndoing**

levelsOfUndo

– (unsigned int)**levelsOfUndo**

Returns the maximum number of top-level undo groups the receiver will hold. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

See also: – **endUndoGrouping**, – **setLevelsOfUndo:**

prepareWithInvocationTarget:

– (id)**prepareWithInvocationTarget:**(id)*target*

Prepares the receiver for invocation-based undo with *target* as the subject of the next undo operation and returns **self**. See “Invocation-Based Undo” in the class description for more information.

See also: – **forwardInvocation:**

redo

– (void)**redo**

Performs the operations in the last group on the redo stack, if there are any, recording them on the undo stack as a single group.

This method posts an EOUndoManagerCheckpointNotification.

See also: – **redo**, – **registerUndoWithTarget:selector:arg:**

reenableUndoRegistration

– (void)**reenableUndoRegistration**

Balances a prior **disableUndoRegistration** message. Undo registration isn't actually reenabled until a reenable message balances the last disable message in effect. Raises an NSInternalInconsistencyException if invoked while no **disableUndoRegistration** message is in effect.

registerUndoWithTarget:selector:arg:

– (void)**registerUndoWithTarget:**(id)*target* **selector:**(SEL)*aSelector* **arg:**(id)*anObject*

Records a single undo operation for *target*, so that when undo is performed it's sent *aSelector* with *anObject* as the sole argument. Also clears the redo stack. Doesn't retain *target*. See "Simple Undo" in the class description for more information.

Raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **forwardInvocation:**, – **groupsByEvent**

setGroupsByEvent:

– (void)**setGroupsByEvent:**(BOOL)*flag*

Sets whether the receiver automatically groups undo operations during the run loop. If *flag* is YES, the receiver creates undo groups around each pass through the run loop; if *flag* is NO it doesn't. The default is YES.

If you turn automatic grouping off, you must close groups explicitly before invoking either **undo** or **undoNestedGroup**.

See also: – **groupsByEvent**

setLevelsOfUndo:

– (void)**setLevelsOfUndo:**(unsigned int)*anInt*

Sets the maximum number of top-level undo groups the receiver will hold to *anInt*. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

If invoked with a limit below the prior limit, old undo groups are immediately dropped.

See also: – **endUndoGrouping**, – **levelsOfUndo**

undo

– (void)**undo**

Closes the top-level undo group if necessary and invokes **undoNestedGroup**. Raises an NSInternalInconsistencyException if more than one undo group is open (that is, if the last group isn't at the top level).

This method posts an EOUndoManagerCheckpointNotification.

See also: – **endUndoGrouping**, – **groupsByEvent**

undoNestedGroup

– (void)**undoNestedGroup**

Performs the undo operations in the last undo group (whether top-level or nested), recording the operations on the redo stack as a single group. Raises an NSInternalInconsistencyException if any undo operations have been registered since the last **endUndoGrouping** message.

This method posts an EOUndoManagerCheckpointNotification.

See also: – **undo**

Notifications

EOUndoManagerCheckpointNotification

Posted whenever an EOUndoManager opens or closes an undo group (except when it opens a top-level group), and when checking the redo stack in **canRedo**. The notification contains:

Notification Object	The EOUndoManager
Userinfo	nil