Change

INHERITS FROM Object

DECLARED IN Change.h

CLASS DESCRIPTION

The Change class is an abstract superclass that is part of the undo mechanism. Create subclasses of Change to represent user actions that should be undoable. Each time the user performs one of these actions, your application should create a change object (an instance of a subclass of Change).

INSTANCE VARIABLES

Inherited from Object

```
Class          isa;
```

Declared in Change

```
struct {
    unsigned int    disabled:1;
    unsigned int    hasBeenDone:1;
    unsigned int
                    changeInProgress:1;
    unsigned int    padding:29;
```

```
} _changeFlags;
id _changeManager;
```

_changeFlags.disabled	YES if this change should not be remembered.
_changeFlags.hasBeenDone	YES after the change has been originally made or redone.
_changeFlags.changeInProgress	YES if the change has not yet been done the first time.
_changeManager	The id of the ChangeManager that owns this change.

METHOD TYPES

Initializing a Change	- init
Called by application code	± startChange ± startChangeIn: ± endChange ± changeManager
Called by ChangeManager	± disable ± disabled ± hasBeenDone ± changeInProgress ± changeName

Used exclusively by ChangeManager ± saveBefoRWCange
± saveAfterChange
± undoChange
± redoChange
± subsumeChange:
± incorporateChange:
± finishChange

INSTANCE METHODS

changeInProgress

- (BOOL)**changeInProgress**

Returns YES if the receiving Change has been sent a **startChange** or **startChangeIn:** message but has not yet received an **endChange** message. You should not need to override this method.

See also: **± startChange, ± endChange**

changeManager

- **changeManager**

Returns the ChangeManager responsible for handling the receiving Change. This method will return **nil** until either a **startChange** or **startChangeIn:** message has been sent to the Change, at which point the Change will find the responsible ChangeManager by searching up the responder chain for the nearest ChangeManager. You should not need to override this method.

See also: **± startChange**

changeName

- (const char *)**changeName**

Override this method to return the name to be used by the ChangeManager in the Undo and Redo menu items. This method is called by **validateCommand:** in the ChangeManager class.

See also: **± validateCommand:** (ChangeManager)

disable

- **disable**

This method is called to tell the receiving Change that it won't be recorded as an explicit change, and won't ever be asked to **undoChange** or **redoChange**. The actual changes represented by the change object will still take place, but the ChangeManager won't record them aRWDseparate action. ChangeManager does not send **saveBeforeChange** and **saveAfterChange** messages to disabled Change objects. A Change object will be disabled by its ChangeManager if any of the following conditions are true: changes have been explicitly disabled in the ChangeManager; the Change was initiated while another Change was already in progress and the Change in progress declined to **incorporateChange:** the new change; or the previous (complete) Change elects to **subsumeChange:** the new Change. You should not need to override this method.

See also: **± saveBeforeChange, ± saveAfterChange, ± incorporateChange:, ± subsumeChange:, ± disableChanges:** (ChangeManager)

disabled

- (BOOL)**disabled**

Returns YES if the change object has received a **disable** message.

See also: **± disable**

endChange

- **endChange**

Signals that a change is complete. This method should be called after the **startChange** method or **startChangeIn:** method has been sent to the same Change. If the receiver has not been disabled, the **endChange** method will send a **changeComplete:** message to the receiver's ChangeManager. Before this method returns, the ChangeManager will send a **saveAfterChange** message back to the Change. If the receiver has been disabled or was unable to find a ChangeManager when it started then **endChange** will cause the receiver to free itself. You should not need to override this method.

See also: **± saveAfterChange**, **± startChange, ± changeComplete:** (ChangeManager)

finishChange

- **finishChange**

The vast majority of all subclasses of Change will not need to use this method. The **finishChange** method is intended to be overridden only in subclasses who's insRWEes subsume other Change instances, and only then by subclasses that need to perform some special action after the last subsumable Change has been subsumed. ChangeManager sends **finishChange** just before

the receiving Change is asked to **undoChange** or just after the receiving Change declines to **subsumeChange:** another Change. If a change is repeatedly undone and redone, the ChangeManager will repeatedly send the **finishChange** message to the same Change, so it is important that the Change keep track of whether this method has already been called.

See also: **± subsumeChange:**

hasBeenDone

- (BOOL)**hasBeenDone**

Returns YES if the Change has been done for the first time or if the change has been redone. Specifically, **hasBeenDone** returns NO if the receiver has never been sent an **endChange** message or if the receiver has been sent an **undoChange** message more recently than a **redoChange** message.

incorporateChange:

- (BOOL)**incorporateChange:***change*

The **incorporateChange:** method is called by the ChangeManager if the receiving Change is in progress when a new *change* is initiated. The receiving Change is given the opportunity to incorporate the new *change*. This mechanism can be used when one user action would create multiple Change objects. For example, a paste command might implemented using two independent, Change producing methods, one for deleting the current selection and one for creating the new selection. In this case, both the deletion Change and the creation Change should really be part of a single paste Change, which will incorporate them as sub-changes. Unlike **subsumeChange:**, this method is called

only when a Change is in progress.

Most subclasses of Change will not need to use this method. You should never need to call this method directly, although you may occasionally want to override it. Your impRWFntation should return YES if the specified *change* should be incorporated into the receiving Change. By returning YES, the receiving Change accepts responsibility for the incorporated *change*, and the ChangeManager will not keep track of it nor free it. Your implementation should return NO when *change* can't or shouldn't be incorporated in the receiving Change. In this case, *change* will be disabled and ignored. The default implementation always returns NO. Note that in either case the receiving Change must still be able to undo any changes in state that happen from the time it receives a **startChange** message until it receives an **endChange** message.
See also: **± disable, ± subsumeChange:**

init

- **init**

Initializes the receiver, a newly allocated Change object.

redoChange

- **redoChange**

Called by the change manager to re-issue a change after it has been undone. This is accomplished by restoring the state of the application using the state information recorded by **saveAfterChange**. You should not need to call this method directly. When overriding this method you should end your method

with ªreturn [super **redoChange**]º.

See also: **± undoChange, ± saveAfterChange**

saveAfterChange
- **saveAfterChange**

Called by the ChangeManager after the receiving Change is sent an **endChange** message, provided the Change is not disabled. Override this method to save any state information modified during the course of the change. This state information can be used by the **redoChange** method to redo a change after it has been undone. You should not need to call this method directly.

See also: **± saveBeforeChange, ± redoChange**

saveBeforeRWGge
- **saveBeforeChange**

Called by the ChangeManager after the receiving Change is sent a **startChange** or **startChangeIn:** message, provided the Change is not disabled. Override this method to save any state information necessary to undo the change later on. For example, if a change causes a variable to be updated, the **saveBeforeChange** method could save the current value of the variable for later use by **undoChange**. You should not need to call this method directly.

See also: **± saveAfterChange, ± undoChange**

startChange
- **startChange**

This method, or its sibling method **startChangeIn:**, is called once

per Change by your application code to signal that a change is about to take place. The Change will open a connection to the nearest ChangeManager on the responder chain. The id of this ChangeManager will be saved in the changeManager instance variable. If the application is not active **startChange** will fail to find a ChangeManager. Use **startChangeIn:** instead of **startChange** if the application is not active. The **startChange** method will return **nil** if no ChangeManager is found. If a ChangeManager is found, it will be sent a **changeInProgress:** message and it will either send the Change either a **disable** message or a **saveBeforeChange** message before **startChange** returns. The code for causing the change should follow a call to **startChange** and should be followed directly by a call to **endChange**. You should not need to override this method.

See also: **± endChange, ± saveBeforeChange, ± startChangeIn:, ± isActive** (Application)

startChangeIn:

- **startChangeIn:***aView*

This method is identical to the **startChange** method, except that **startChangeIn:** may successfully locate a ChangeManager even if the application is not the active application, which **startChangewill not. In order to find a ChangeManager startChangeIn:** must be passed *aView* in which the change is occurring, which it will use to find the beginning of the responder chain. You should not need to override this method.

See also: **± endChange, ± saveBeforeChange, ± startChange, ± isActive** (Application)

subsumeChange:

- (BOOL)**subsumeChange:***change*

This method is called by the ChangeManager to offer the receiver (which is the last completed Change) the opportunity to subsume the next Change about to be performed by the application. Override this method when you want to coalesce a series of similar Changes into one large Change. For example, a series of cursor movements could be collapsed into a single Change. The first Change created by cursor movement would subsume all cursor Changes following it directly. The ChangeManager only calls this method on completed Changes.

Most subclasses of Change will not need to use this method. You should never need to call this method directly, although you may occasionally want to override it. Your implementation should return YES if you wish to signal that *change* should be subsumed. In this case, *change* will be disabled and will be freed as soon as it receives an **endChange** message. Note that the current change is expected to be able to undo any changes in state that occur before *change* receives the **endChange** message. You should return NO when *change* cannot be subsumed by the current change. When this happens, the ChangeManager will send the receiver a **finishChange** message and then record *change* as an independent change The default implementation always returns NO.

See also: **± disable, ± incorporateChange:**

undoChange

- **undoChange**

This method tells the receiving Change to restore the state information first saved when **saveBeforeChange** was called. This information should be suffRWInt to restore the state of the application to the way it was before the change took place. This method may either be called to undo the Change after the first time the Change was made, or after a Change has been redone. You should not need to call this method directly. When overriding this method you should end your method with ªreturn [super **undoChange**]º.

See also: ± **redoChange**, **± saveBeforeChange**