ChangeManager

INHERITS FROM Responder : Object

DECLARED IN ChangeManager.h

CLASS DESCRIPTION

The ChangeManager class is the part of the undo mechanism that collects change objects and manipulates the undo and redo menu items. This class works with the Change class to provide a simple way to implement multi-level undo. Change managers communicate with change objects through the responder chain. By deriving window delegates from ChangeManager you can easily implement document-level undo. By installing a change manager as an application delegate you can also implement application wide undo.

INSTANCE VARIABLES

Inherited from Object Class isa;

Inherited from Responder id nextResponder;

Declared in ChangeManager

List *_changesList;
Change *_lastChange;
Change *_nextChange;

```
Change          *_changeInProgress;
int             _numberOfDoneChanges;
int
                _numberOfUndoneChanges;
int
                _numberOfDoneChangesAtLastClean;
BOOL            _someChangesForgotten;
int             _changesDisabled;
```

_changesList	A list of changes that have been made.
_lastChange	The id of the change that can be undone.
_nextChange	The id of the change that can be redone.
_changeInProgress	The id of the change which is currently underway.
_numberOfDoneChanges	The number of changes made.
_numberOfUndoneChanges	The number of changes that have been undone.
_numberofDoneChangesAtLastClean	A count of changes made when **clean** was last called.
_someChangesForgotten	YES if some changes have been thrown away

_changesDisabled	The number of nested calls to **disableChanges:**.

METHOD TYPES

Initializing and freeing	- init ± free
Disabling undo	± disableChanges: ± enableChanges:
Examining state	± canUndo ± canRedo ± isDirty
Setting state	± dirty: ± clean: ± reset:
Validating Menu Commands	± validateCommand:
Undoing and Redoing	± undoOrRedoChange: ± undoChange: ± redoChange:
Tracking change progress	± changeInProgress: ± changeComplete:
Subclass notification	± changeWasDone ± changeWasUndone ± changeWasRedone

INSTANCE METHODS

canRedo

- (BOOL)**canRedo**

Returns YES if there is a Change that can be redone. The name of this Change will be visible in the redo or undo/redo menu item. You should not need to override this method.

See also: **± validateCommand:**

canUndo

- (BOOL)**canUndo**

Returns YES if there is a Change that can be undone. The name of this Change will be visible in the undo or undo/redo menu item. You should not override this method.

See also: **± validateCommand:**

changeComplete:

- **changeComplete:***change*

Called by Change objects to signify that *change* is done. The receiving ChangeManager will then ask *change* to save the new state information via **saveAfterChange**. Just before returning, the **changeComplete:** method sends a **changeWasDone** message to self, which provides subclasses of ChangeManager with an opportunity to react to the change. You should never call **changeComplete:** directly, nor should you override it.

See also: **± changeInProgress:, ± changeWasDone, ± saveAfterChange** (Change)

changeInProgress:

- **changeInProgress:***change*

Called by Change objects to signify that a *change* is about to be made. If changes have been disabled using **disableChanges:** then **changeInProgress:** will send a **disable** message to *change* and immediately return. If changes have not been disabled, the receiving ChangeManager tries to find a home for *change*. If another Change is already in progress that Change is sent an **incorporateChange:** message with *change* as the argument. If the Change in progress returns YES then *change* is sent a **saveBeforeChange** message, otherwise it is sent a **disable** message. If there is no Change already in progress, but there is a previous completed Change then the previous Change is sent a **subsumeChange:** message with *change* as the argument. If the previous Change returns YES then *change* is sent a **disable** message. If the previous Change returns NO, or if there is no previous Change, *change* is sent a **saveBeforeChange** message and set to be the current Change in progress, and the previous Change, if there is one, is sent a **finishChange** message. You should never need to call **changeInProgress:** directly, nor should you need to override it.

See also: ± **changeComplete:, ± saveBeforeChange** (Change), ± **incorporateChange:** (Change), ± **subsumeChange:** (Change), ± **finishChange** (Change)

changeWasDone

- **changeWasDone**

Override this method if your subclass needs to know when a change has been made. For example, this hook can be used to update the close box on a document window to reflect the dirty state of the ChangeManager. You should not call this method directly.

See also: **± changeWasRedone, ± changeWasUndone, ± isDirty**

changeWasRedone

- **changeWasRedone**

Override this method if your subclass needs to know when a change has been redone. For example, this hook can be used to update the close box on a document window to reflect the dirty state of the ChangeManager. You should not call this method directly.

See also: **± changeWasDone, ± changeWasUndone, ± isDirty**

changeWasUndone

- **changeWasUndone**

Override this method if your subclass needs to know when a change has been undone. For example, this hook can be used to update the close box on a document window to reflect the dirty state of the ChangeManager. You should not call this method directly.

See also: **± changeWasDone, ± changeWasRedone, ± isDirty**

clean:

- **clean:***sender*

Tells the receiving ChangeManager to consider its current state to be clean. Calls to **isDirty** will return NO until further change activity occurs. In ChangeManagers that correspond to documents, you should call **clean:** each time the document is saved. By doing this, the **isDirty** method can be used to tell whether the saved representation of the document matches the internal memory representation. When overriding this method you should begin your method with ª[super **clean**:sender]º.

See also: ± **dirty:**, ± **reset:**, **± isDirty**

dirty:

- **dirty:***sender*

Forces the receiving ChangeManager to appear dirty. Call this method when your code as made a change that wasn't recorded with a Change object. After a **dirty** message is received the **isDirty** method will return YES until a **clean:** or **reset:** message is received. When overriding this method you should begin your method with ª[super **dirty**:sender]º.

See also: **± clean:, ± reset:, ± isDirty**

disableChanges:

- **disableChanges:***sender*

This method increments the receiver's changesDisabled instance variable. As long as changesDisabled is non-zero, new change objects will be disabled. You should not need to override this method.

See also: **± enableChanges, ± disable** (Change)

enableChanges:

- **enableChanges:***sender*

Decrements the receiver's changesDisabled instance variable. You should not need to override this method.

See also: **± disableChanges**

free

- **free**

Calls **reset:** to clean out any change objects and frees the ChangeManager object.

init

- **init**

Initializes the receiver, a newly allocated ChangeManager object.

isDirty

- (BOOL)**isDirty**

Returns NO if no net change activity has occurred since the ChangeManager was initialized or since the last **clean:** or **reset:** message was received. For example, if a single Change has been undone and then redone since the last **clean:** message, then isDirty will return NO. The completion of the next new, non-disabled Change will cause **isDirty** to return YES. You should not need to override this method.

See also: **± disableChanges:, ± clean:, ± dirty:, ± reset:**

redoChange:

- **redoChange:***sender*

This method should be the action performed by the redo menu item in an application with multiple-undo. The **redoChange:** method sends a **redoChange** message to the last Change that was undone. The name of this Change will then appear in the undo menu item. Your application should not use both **redoChange:** and **undoOrRedoChange:** at the same time. You should not need to override this method.

See also: **± undoChange:, ± undoOrRedoChange:**

reset:

- **reset:***sender*

Causes the receiving ChangeManager to free all the Change objects that it is managing. The state of the ChangeManager is re-initialized to the state after it first received the **init** message. When overriding this method you should

begin your method with ª[super **reset**:sender]º.

undoChange:

- **undoChange:***sender*

This method should be the action performed by the undo menu item in an application with multiple-undo. The **undoChange:** method sends an **undoChange** message to the last Change that was done or redone. The name of this Change will then appear in the redo menu item. Your application should not use both **undoChange:** and **undoOrRedoChange:** at the same time. You should not need to override this method.

See also: **± redoChange:, ± undoOrRedoChange:**

undoOrRedoChange:

- **undoOrRedoChange:***sender*

This method should be the action performed by the undo menu item in an application offering single-level undo. If the last change has already been done, then it will be undone. If was just undone, then it will be redone. In order to make your application use single-level undo you must edit ChangeManager.m and define the N_LEVEL_UNDO constant to be 1. Your application should not use both **undoChange:** and **undoOrRedoChange:** at the same time. You should not need to override this method.

Although **undoOrRedoChange:** is really intended for applications with single-level undo, it will attempt to do something reasonable in applications with multiple-undo. If there is a Change that can be undone **undoOrRedoChange:** sends an **undoChange** message to the Change. If there is no Change that can be undone, but there is a Change that can be redone then **undoOrRedoChang**e: sends a **redoChange** message to the Change.

See also: **± undoChange:, ± redoChange:**

validateCommand:

- (BOOL)**validateCommand:***menuCell*

This method can be used to change the state of menu items corresponding to undo, redo and undo/redo. Use this method as the update action for menu cells that invoke **undoChange:**, **redoChange:**, or **undoOrRedoChange:**. The value returned is YES if the command specified in the update action of *menuCell* is valid.

Independent of whether the command is valid or not, the change manager may update the title of *menuCell* to contain the correct name of the current changes.

See also: **± undoChange:, ± redoChange:, ± undoOrRedoChange:, ± setUpdateAction:forMenu:** (MenuCell)