User Interface Validation

Cocoa > User Experience

2007-07-10

Contents

Introduction to User Interface Validation 5

Organization of This Document 5

Implementing Validation 7

The Target Object 7
Implementation Steps 7
Example Implementation 8

Implementing a Validated Item 9

Document Revision History 11

Introduction to User Interface Validation

The protocols `NSUserInterfaceValidations` and `NSValidatedUserInterfaceItem` provide a standard way to validate user interface items—that is, to set their state as appropriate for the current application context (for example, to disable the Paste menu item if there is no suitable data on the pasteboard).

You should read this document to learn how to implement user interface item validation and how to extend the user interface validation protocol.

For more information about other ways to validate menus and pop-up lists, see *Application Menu and Pop-up List Programming Topics for Cocoa*.

Organization of This Document

"Implementing Validation" (page 7) describes how to use the `NSUserInterfaceValidations` and *NSValidatedUserInterfaceItem* protocols to validate user interface items.

"Implementing a Validated Item" (page 9) describes you can implement an item that uses the validation protocol to determine its state, and how to extend the `NSUserInterfaceValidations` protocol to provide custom user interface item validation.

Implementing Validation

Before it is displayed, a user interface item checks to see if its target implements `validateUserInterfaceItem:`. If it does, then the enabled status of the item is determined by the return value of the method. You can therefore conditionally enable or disable an item by implementing `validateUserInterfaceItem:` in the target object.

The Target Object

In some situations (typically if you set the target and action in Interface Builder), the target is the object to which the user interface item is connected directly; in other situations (when the user interface item's target is `nil`—such as if you connected it to First Responder in Interface Builder), the target is the first object in the responder chain that implements its action method. For more details, see "Responder Chain for Action Messages" in *Cocoa Event-Handling Guide* > Event Architecture. In either case, the important thing to realize is that *the target is the object that implements the user interface item's action method*. (The target is also the object that has the suitable context to know whether the action is appropriate.) Hence *the object that must implement* `validateUserInterfaceItem:` *is the same object that implements the action method*.

For example, suppose you have a controller object that implements a `paste:` method. When the `paste:` method is invoked, it retrieves a value from the pasteboard and then pastes it into the current selection. It can only do this if there is a valid value on the pasteboard. The controller object therefore also implements `validateUserInterfaceItem:`. The `validateUserInterfaceItem:` checks to see if the pasteboard contains useable data, and if it does it returns `YES` otherwise it returns `NO`. If the pasteboard does not contain useable data, the user interface item is disabled.

If in your application you have more than one controller that implements `paste:`—each responsible for pasting different data—then each should have its own implementation of `validateUserInterfaceItem:` that checks to see if the pasteboard contains data useable by it.

Implementation Steps

The implementation of `validateUserInterfaceItem:` should follow these steps:

1. To decide whether or not an item should be enabled, you need to know what it will do if the user selects it. The sender implements the `NSValidatedUserInterfaceItem` protocol, so you can find out what tag and action are associated with it. You typically first therefore check to see what *action* is associated with the item (you need to test for each of the actions you're interested in).

   Checking the action rather than the tag means you avoid the fragility of having to remember to use the same tag for each user interface element that invokes the same method on the target.

2. If the action is something you're interested in, then return a Boolean value appropriate for the current context.

3. If the action is not something you're interested in, then either:

    a. If your superclass implements the validation method (for example, `NSDocument` and `NSObjectController` implement `validateUserInterfaceItem:`), invoke super's implementation; otherwise

    b. Return a default value (typically `YES`).

Example Implementation

The following example illustrates the implementation of `validateUserInterfaceItem:` in a subclass of `NSDocument`.

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)anItem
{
    SEL theAction = [anItem action];

    if (theAction == @selector(copy:))
    {
        if ( /* there is a current selection and it is copyable */ )
        {
            return YES;
        }
        return NO;
    } else if (theAction == @selector(paste:))
    {
        if ( /* there is a something on the pasteboard we can use and
               the user interface is in a configuration in which it makes sense
 to paste */ )
        {
            return YES;
        }
        return NO;
    } else
        /* check for other relevant actions ... */
    }
    // subclass of NSDocument, so invoke super's implementation
    return [super validateUserInterfaceItem:anItem];
}
```

Implementing a Validated Item

NSValidatedUserInterfaceItem is used by the Application Kit's standard user interface validation mechanism, and must be implemented by validated objects.

Validated objects send validateUserInterfaceItem: to validators that can be determined by NSApplication's targetForAction:to:from:.

You can extend this functionality by introducing a new set of protocol pairs that is targeted to your specific validated objects. NSMenuItem protocol is one example extending this validation machinery to allow validators to modify menu items being validated. You can extend UI validation by:

1. Declare a protocol that inherits from NSValidatedUserInterfaceItem.

   You can add as many features you want for your validated objects in this protocol, for example:

   ```
   @protocol NSValidatedToolbarItem <NSValidatedUserInterfaceItem>
   - (NSImage *)image;
   - (void)setImage:(NSImage *)theImage
   - (NSString *)toolTip;
   - (void)setToolTip:(NSString *)theToolTip;
   @end
   ```

2. Declare validation method for validators.

   You should declare the new selector that takes your objects as the argument, for example:

   ```
   @protocol NSToolbarItemValidations
   - (BOOL)validateToolbarItem:(id <NSValidatedToolbarItem>)theItem;
   @end
   ```

3. Implement your update method.

   You should, first, check if your current validator responds to your validation method, then, the generic validateUserInterfaceItem:. This way, your object can be automatically enabled/disabled by the Application Kit's standard objects like NSTextView without any additional coding, for example:

   ```
   - (void)update {
       id validator = [NSApp targetForAction:[self action] to:[self target]
   from:self];

       if ((validator == nil) || ![validator respondsToSelector:[self action]])
       {
            [self setEnabled:NO];
       }
       else if ([validator respondsToSelector:@selector(validateToolbarItem:)])
       {
           [self setEnabled:[validator validateToolbarItem:self]];
       }
      else if ([validator respondsToSelector:@selector(validateUserInterfaceItem:)])
       {
   ```

```
            [self setEnabled:[validator validateUserInterfaceItem:self]];
        }
        else
        {
             [self setEnabled:YES];
        }
    }
```

4. Optionally, implement category methods for standard objects .

Now you can implement default validation methods for standard objects like `NSTextView` or `NSDocument`, for example:

```
@implementation NSTextView (NSToolbarValidation)

- (BOOL)validateToolbarItem:(id <NSValidatedToolbarItem>)theItem
{
    BOOL returnValue = [self validateUserInterfaceItem:theItem];
     // Your own validation
     return returnValue;
}
@end
```

Document Revision History

This table describes the changes to *User Interface Validation*.

Date	Notes
2007-07-10	Added link to Menus Programming Guide.
2003-02-24	Corrected typographical errors in "Implementing a Validated Item" (page 9).
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.