A MULTICAST DELEGATE PATTERN FOR IOS CONTROLS

by Colin Eberhardt (http://blog.scottlogic.com/ceberhardt)

This blog post introduces a simple pattern for adding multicasting capabilities to existing iOS controls. Adding multicasting allows for improved clarity and code re-use.

Most iOS controls have a concept of a *'delegate'* - a protocol which is used to handle various user interactions and control state changes. If, for example, you want to detect when a `UIWebView` starts to load a web page, you set the `delegate` property to a class which adopts the `UIWebViewDelegate` protocol, and implement the `webViewDidStartLoad:` method. This works well in practice for simple cases, the fact that each control has a single associated delegate is quite limiting. There are various reasons why you might want to handle delegate messages, and naturally you might want to handle these in different parts of your code in order to promote code re-use. For this reason, a multicasting delegate would be a much better option.

The runtime behaviour of Objective-C is based on message passing between object instances. In most cases message passing is equivalent to invoking a method directly n the target object, however, it gives us the opportunity to re-route messages to one or more targets.

Image courtesy of kainet (http://www.flickr.com/photos/kainet/112885753/sizes/m/in/photostream/), used under Creative Commons ShareAlike license

In a recent article for Ray Wenrderlich's site (http://www.raywenderlich.com/22174/how-to-make-a-gesture-driven-to-do-list-app-part-33), I demonstrated how `forwardingTargetForSelector:` could be used to forward messages from one object to another. The code below shows how a class can handle `UIScrollViewDelegate` messages, whilst forwarding these messages to a 'chained' delegate:

```
#pragma mark - UIScrollViewDelegate forwarding
-(BOOL)respondsToSelector:(SEL)aSelector {
    if ([self.delegate respondsToSelector:aSelector]) {
        return YES;
    }
    return [super respondsToSelector:aSelector];
}


-(id)forwardingTargetForSelector:(SEL)aSelector {
    if ([self.delegate respondsToSelector:aSelector]) {
        return self.delegate;
    }
    return [super forwardingTargetForSelector:aSelector];
}
```

For more information on this example, see the original article (http://www.raywenderlich.com/22174/how-to-make-a-gesture-driven-to-do-list-app-part-33).

Message forwarding allows you to handle messages sent to the delegate, forwarding them to another delegate implementation, resulting in an implementation that follows the Proxy Pattern (http://en.wikipedia.org/wiki/Proxy_pattern). This does allow for multiple delegate implementations, but the use of chaining rather than multicasting is quite messy.

So, what's the alternative? Fortunately Objective-C provides a more generic mechanism for processing messages by implementing `forwardInvocation:`. Using this to create a multicasting concept is really quite simple, so we'll just dive right into the code for a class that provides this functionality. The interface of this class simply allows the user to add multiple delegates:

```
#import <Foundation/Foundation.h>

// handles messages sent to delegates, multicasting these messages to multiple
observers
@interface SHCMulticastDelegate : NSObject

// Adds the given delegate implementation to the list of observers
- (void)addDelegate:(id)delegate;

@end
```

The implementation stores these delegates in an array. Whenever a message is sent to the `SHCMulticastDelegate` it determines whether the delegates can handle this message via `respondsToSelector:`, if so, `forwardInvocation:` iterates over the delegate using the supplied `NSInvocation` instance to forward the message to each of these delegates.

```
#import "SHCMulticastDelegate.h"

@implementation SHCMulticastDelegate
{
    // the array of observing delegates
    NSMutableArray* _delegates;
}

- (id)init
{
    if (self = [super init])
    {
        _delegates = [NSMutableArray array];
    }
    return self;
}

- (void)addDelegate:(id)delegate
{
    [_delegates addObject:delegate];
}

- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ([super respondsToSelector:aSelector])
        return YES;

    // if any of the delegates respond to this selector, return YES
    for(id delegate in _delegates)
    {
        if ([delegate respondsToSelector:aSelector])
        {
            return YES;
        }
    }
    return NO;
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    // can this class create the signature?
    NSMethodSignature* signature = [super methodSignatureForSelector:aSelector];
```

```
    // if not, try our delegates
    if (!signature)
    {
        for(id delegate in _delegates)
        {
            if ([delegate respondsToSelector:aSelector])
            {
                return [delegate methodSignatureForSelector:aSelector];
            }
        }
    }
    return signature;
}

- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    // forward the invocation to every delegate
    for(id delegate in _delegates)
    {
        if ([delegate respondsToSelector:[anInvocation selector]])
        {
            [anInvocation invokeWithTarget:delegate];
        }
    }
}

@end
```

The implementation of `methodSignatureForSelector:` is required by `forwardInvocation:` as part of the standard forwarding procedure.

NOTE: This implementation does nothing to check that each of the supplied delegates conform to the same delegate protocol. It could certainly be made more robust!

In practice, this class can be used as follows:

```
// add multicasting capabilities to a UITextView
SHCMulticastDelegate* multicast = [[SHCMulticastDelegate alloc] init];
self.textView.delegate = multicast;

// add multiple delegates
[multicast addDelegate:_someDelegateImplementation];
[multicast addDelegate:_anotherDelegateImplementation];
```

In the above code, each of the delegates that were added to the `SHCMulticastDelegate` will be informed when `UITextViewDelegate` messages are sent.

This implementation looks pretty good, but there is still room for improvement. In the above code, any class that wishes to handle message sent to the delegate must obtain a reference to the SHCMulticastDelegate instance. It would be much better if the multicasting capability could be added to the `UITextView` directly.

Categories to the rescue ...

The following category adds a property to `UITextView`

```
#import <UIKit/UIKit.h>
#import "SHCMulticastDelegate.h"

@interface UITextView (Multicast)

@property (readonly) SHCMulticastDelegate* multicastDelegate;

@end
```

Typically you would use an instance variable to 'back' a property, however, you cannot add instance variables to class using a category. Fortunately, Objective-C provides a way to associate data with an object using string keys. See the use of `objc_getAssociatedObject` in the code below:

```
#import "UITextView+Multicast.h"
#import <objc/runtime.h>

@implementation UITextView (Multicast)

NSString* const UITextViewMulticastDelegateKey = @"multicastDelegate";

- (SHCMulticastDelegate *)multicastDelegate
{
    // do we have a SHCMulticastDelegate associated with this class?
    id multicastDelegate = objc_getAssociatedObject(self, (__bridge const void *)(UI
    if (multicastDelegate == nil) {

        // if not, create one
        multicastDelegate = [[SHCMulticastDelegate alloc] init];
        objc_setAssociatedObject(self, (__bridge const void *)(UITextViewMulticastDe

        // and set it as the delegate
        self.delegate = multicastDelegate;
    }

    return multicastDelegate;
}

@end
```

With the above code `UITextView` now has multicasting capabilities built in. We'll look at how this category can be used in a slightly more in-depth example.

A common use of the `UITextViewDelegate` is to prohibit newlines in a multi-line text input. Typically this would require pasting the standard code required into the class which handles the delegate. Here we'll see how multicasting can be used to make this code more easily re-used.

The following class handles the delegate in order to provide the hide-keyboard-on-enter behaviour:

```
#import <Foundation/Foundation.h>

@interface UITextViewHideKeyboardOnEnterBehaviour : NSObject <UITextViewDelegate>

- (id) initWithTextView:(UITextView*) textView;

@end
```

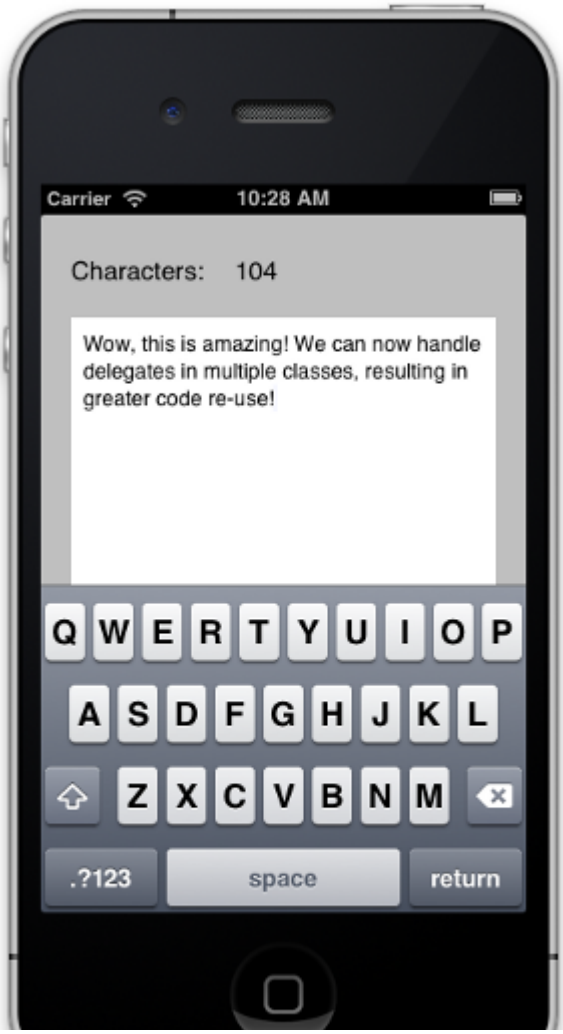

The implementation is pretty trivial:

```
#import "UITextViewHideKeyboardOnEnterBehaviour.h"
#import "UITextView+Multicast.h"

@implementation UITextViewHideKeyboardOnEnterBehaviour

- (id)initWithTextView:(UITextView *)textView
{
    if (self=[super init])
    {
        [textView.multicastDelegate addDelegate:self];
    }
    return self;
}

- (BOOL)textView:(UITextView *)textView shouldChangeTextInRange:(NSRange)range repla
    // see: http://stackoverflow.com/questions/703754/how-to-dismiss-keyboard-for-ui
    if([text isEqualToString:@"\n"]) {
        [textView resignFirstResponder];
        return NO;
    }

    return YES;
}

-(BOOL)textViewShouldBeginEditing:(UITextView *)textView
{
    return YES;
}

@end
```

This behaviour can now be added to any `UITextView` without interfering with application-specific logic. For example, this simple view controller also handles the `UITextViewDelegate` in order to display a character count:

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    // add the hide-keyboard-on-enter behaviour
    [[UITextViewHideKeyboardOnEnterBehaviour alloc] initWithTextView:self.textView];

    // add 'self' as a delegate to provide a character count
    [self.textView.multicastDelegate addDelegate:self];

}

-(void)textViewDidChange:(UITextView *)textView
{
    self.countLabel.text = [NSString stringWithFormat:@"%d", self.textView.text.leng
}

@end
```

Nice :-)



The above example was deliberately selected because the implementation of this 'behaviour' is quite simple. For a more complex example, take a look at the 'Clear Style' (https://github.com/ColinEberhardt/iOS-ClearStyle) project I have over on github. Here I use multicasting in order to add multiple pull-to-add-new behaviours to a list, where each behaviour handles the same delegate methods.

Finally, you might have noticed that the `UITextViewHideKeyboardOnEnterBehaviour` implementation above handles a delegate method with a non-void signature. Despite the fact that we are multicasting these messages return values are still possible! I must admit I could not find any information about how the runtime handles return values when `invokeWithTarget:` is used to invoke multiple targets. Through testing I have discovered that the value returned by the last invocation is the one that is ultimately returned. I would certainly recommend caution if you plan to use this feature!

You can download the code for the simple example here: MulticastDelegates.zip (http://blog.scottlogic.com/archive/2012/11/MulticastDelegates.zip)

Regards, Colin E.

Comments for this thread are now closed.

12 Comments Scott Logic Ltd 1 Login

Recommend Share Sort by Best

737 Flight Simulator · a year ago

Definitely it is a better pattern than KVO observation. The only problem with your implementation I see here is that you keep strong references to all delegates. That will force user to remember to remove the delegate, for instance what if one of your textview's delegate is a view controller, you would create retain loop if you didn't remove delegate in the right place.
Please take a look at this answer stackoverflow.com/a/14219598 or NSPointerArray which might be solution to this problem.

1 · Share ›

Alfred ↗ 737 Flight Simulator · a year ago

Correct, SHCMulticastDelegate should keep weak references to its delegates.

I advise using a weak objects NSHashTable for that. The only change required is:

```
@implementation SHCMulticastDelegate
{
// the observing delegates
NSHashTable* _delegates;
}

- (id)init
{
if (self = [super init])
{
_delegates = [NSHashTable weakObjectsHashTable];
}
return self;
}
```

2 · Share ›

Simon ↗ Alfred · 9 months ago

I think that is best solution! Weak references don't force to hold memory AND if weak reference will get zeroed, hash table will reuse this free space. So even after heavy use hash tables will be relatively small...

1 · Share ›

Colin E. Mod ↗ 737 Flight Simulator · a year ago

Thanks, I see your point - I think I was showing my lack of Objective-C experience when I wrote this article!

· Share ›

hfossli · 2 months ago

I don't think this should replace NSNotificationCenter or KVO when implementing classes. BUT when using UIKit I do think this has a great application when it comes to UIScrollView and friends. When implementing e.g. parallax effect there are mulitple instances that would like to have callbacks.

· Share ›