Objective-C Categories as Stylesheets

Adrian Kosmaczewski

2010-06-03

It is very important that iPhone and iPad applications use visual styles in a coherent way. This helps users learn how to use your application faster, it helps them scan your UI for important information as quickly as possible, and it also can convey a strong marketing message; companies who want iPhone or iPad applications often have complex visual identities, including predefined fonts and colors, and they will want their applications to match those choices.

However, getting all the UI widgets to look similarly can be complex, particularly in large applications; what if your client or their designers change their minds about the font size or some background color right before shipping your project? Of course you can "search and replace" all occurrences of some color using Xcode, but you have the risk of leaving some unchanged widget somewhere. And believe me, this happens really often.

In this article I will discuss a simple approach, using Objective-C categories, to keep your styling information separated from the rest of the application, using a system that will be familiar to developers used to creating websites using CSS (Cascading Style Sheets).

First Approach: Simple Categories

The first, easiest approach is to create simple UIFont and UIColor categories in your application, and provide semantic descriptions of the data you want to style; for example "customerNameFont" and "companyPhoneColor" are good names. You can add as many class methods as you want to the UIFont and UIColor classes for that:

```
@interface UIFont (YourAppName)

+ (UIFont *)customerNameFont;
+ (UIFont *)customerPhoneFont;

@end

@implementation UIFont (YourAppName)

+ (UIFont *)customerNameFont
{
    return [UIFont fontWithName:@"Helvetica" size:22.0];
}
```

```
+ (UIFont *)customerPhoneFont
{
    return [UIFont boldSystemFontOfSize:15.0];
}

@end
```

And we provide the same treatment to the UIColor class:

```
@interface UIColor (YourAppName)

+ (UIColor *)customerNameColor;
+ (UIColor *)customerPhoneColor;

@end

@implementation UIColor (YourAppName)

+ (UIColor *)customerNameColor
{
    return [UIColor blueColor];
}

+ (UIColor *)customerPhoneColor
{
    return [UIColor redColor];
}

@end
```

Then, you can use those method names in the rest of your application:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
                                       reuseIdentifier:CellIdentifier] autorelease];

        // And here goes the styling!
        cell.textLabel.font = [UIFont customerNameFont];
        cell.textLabel.textColor = [UIColor customerNameColor];

        cell.detailTextLabel.font = [UIFont customerPhoneFont];
        cell.detailTextLabel.textColor = [UIColor customerPhoneColor];
    }
    Customer *cust = [self.data objectAtIndex:indexPath.row];
    cell.textLabel.text = cust.name
    cell.detailTextLabel.text = cust.phone

    return cell;
}
```

The advantage of this simple approach is that all styles are managed from a central location, and you can change the fonts and colors of your application in a single step.

There are, however, two major disadvantages:

1. Colors and fonts defined on their respective categories will not be available in Interface Builder, so you might have to override viewDidLoad or other methods to force the styling at a certain moment, or to set the color and font values manually, which might lead to duplicate information.
2. Some UI elements have support for even more styling information, like shadows or CGAffineTransform values, which cannot be handled in a simple way with this method.

We need some kind of CSS for iPhone applications, and the next section provides a rudimentary approach to that problem.

Second Approach, "Cascading Style" Classes

Let's push the idea of semantic styling a bit; in the previous example, we created separated categories for UIFont and UIColor and used them in atomic form, changing individual properties of your widgets, one by one.

What if you wanted to set several properties at once, just like with CSS classes on web pages? What if you had a system which could be extended to support more properties, for other, as of yet unknown, future UI widgets? For that, we would need a special container for style information, like a CSS stylesheet, and we should be able to assign this container to any kind of visual widget; in turn, those widgets would automatically adapt their layout and appearance. There are many different ways to do this; and this is just one of them.

First I declare a class called AKCascadingStyle, which holds basic styling information, and which is able to apply that information to any kind of object passed in parameter:

```
@interface AKCascadingStyle : NSObject
{
@private
    UIFont *_font;
    UIColor *_textColor;
    UIColor *_backgroundColor;
    UIColor *_tintColor;
}

@property (nonatomic, retain) UIFont *font;
@property (nonatomic, retain) UIColor *textColor;
```

```
@property (nonatomic, retain) UIColor *backgroundColor;
@property (nonatomic, retain) UIColor *tintColor;

+ (id)style;
+ (id)styleFromObject:(id)object;

- (void)applyToObject:(id)object;
- (void)setValuesFromObject:(id)object;

@end
```

This class can be inherited and extended, and its methods can be overridden, as we'll see shortly, to add support for more properties. Then I add a category on NSObject (careful with this!) to support adding and retrieving style information in the form of AKCascadingStyle instances:

```
@class AKCascadingStyle;

@interface NSObject (AKCascadingStyle)

@property (nonatomic, retain) AKCascadingStyle *cascadingStyle;

@end
```

Why a category on NSObject? Because not everything you see on your iPhone screen is a subclass of UIView! UIBarButtonItems, for example, inherit from UIBarItem, which itself inherits from NSObject, and not from UIView. By the way, by extending NSObject, this code could also be used in Mac OS X applications, for example, without modification.

Given that categories cannot add ivars to existing classes, the getter of this property will call the [AKCascadingStyle styleFromObject:] method above:

```
@implementation NSObject (AKCascadingStyle)

@dynamic cascadingStyle;

- (void)setCascadingStyle:(AKCascadingStyle *)style
{
    [style applyToObject:self];
}

- (id)cascadingStyle
{
    return [AKCascadingStyle styleFromObject:self];
}

@end
```

Both [AKCascadingStyle applyToObject:] and [AKCascadingStyle styleFromObject:] are polymorphic (as all Objective-C methods are) so subclasses can extend their functionality as required (but don't forget to send the same message to "super" before!):

```
@interface ShadowStyle : AKCascadingStyle
{
@private
    CGSize _shadowOffset;
    UIColor *_shadowColor;
}

@property (nonatomic) CGSize shadowOffset;
@property (nonatomic, retain) UIColor *shadowColor;

@end
```

Then, to use these style classes in your own controllers, just do the following:

```
- (IBAction)addStyle:(id)sender
{
    self.contentsTextView.cascadingStyle = [ShadowStyle style];
}
```

Setting this property will automatically set all the required parameters in your widget; text color, background colors, transformations, you name it. You can inherit styles in order to reuse them, and you can style pretty much any kind of UIKit widget with it.

You can download the code of this project from Github[1], as usual! Enjoy!

[1]http://github.com/akosma/AKCascadingStyle