Good API Design

Today I'm going to talk about what I've learned about Good API Design. I've been working on some frameworks for the past year or so and would love to share what I've learned with you.

Why?

Why learn good API Design?

Chances are you'll be dealing a lot with various APIs in standard libraries and frameworks as a software engineer. Chances are, you've been frustrated with them at some point or another. You may've even felt the urge to rewrite functionality of a framework because of inelegance or some other problem. Or maybe you've fallen in love with a framework but still felt it could be improved or expanded.

Especially when dealing with open source code, there are tons of options out there. People will likely choose or not choose your code on the API it presents.

Five Principles

for Good API Design

Throughout my, although brief, time designing APIs, I've collected what I've learned into five principles.

If you're developing a piece of code built for a platform or on top of a popular framework, your users are going to be comfortable and familiar with that framework. They're going to be expecting similar style. Know the environment you're working in and be consistent with that. Make sure you learn the patterns behind the framework you're using. Learn what works well in those frameworks. The people most likely to use your code will be expecting this.

Make it specific.

Realise that your code is probably not going to satisfy every single person that comes across it. Design it for pretty specific users and use cases. Avoid implementing features far out of scope of the purpose of the project.

By designing API a certain way, you can lead your users down a particular path.

Think about use cases.
What are most people going to want to do with your code?
Design the default behaviour for this. Use sensible defaults.
As little configuration as possible. Settings are generally an admission of defeat. Good software is opinionated.

Make it understandable.

When people are looking through options of frameworks to accomplish their tasks, they'll take a quick look at your API and see what it has to offer and if this is what they were expecting. With a ton of open source code available online and easily searchable through services like Github, if a user is looking at your API and doesn't find it understandable immediately, they'll probably move on to try and find something else.

The API should be easy to learn and intuitive. The API should provide what the user is expecting.

The names of variables (or properties) and functions (or methods) should be self-explanatory.

The code should lead the user in a straightforward manner.

The user should be able to figure out how to use your code just by the design of the API, without reading much documentation.
However, you should still have it thoroughly documented.

You'll find that no matter how much you try to make things self-explanatory, there are often very specific questions that won't be clear. For this, you will need some great documentation. Try to make it as brief as possible. Avoid talking about implementation details, if possible. Don’t say more than you have to.

Expect some common misconceptions and misuse cases. Your documentation should handle those. Be forgiving.

Make it difficult
to use incorrectly or make mistakes.

Don’t offer anything that is misleading. This is important if you’re subclassing an object and inheriting the super class’s interface, too.

Be forgiving.

Let’s see some examples.

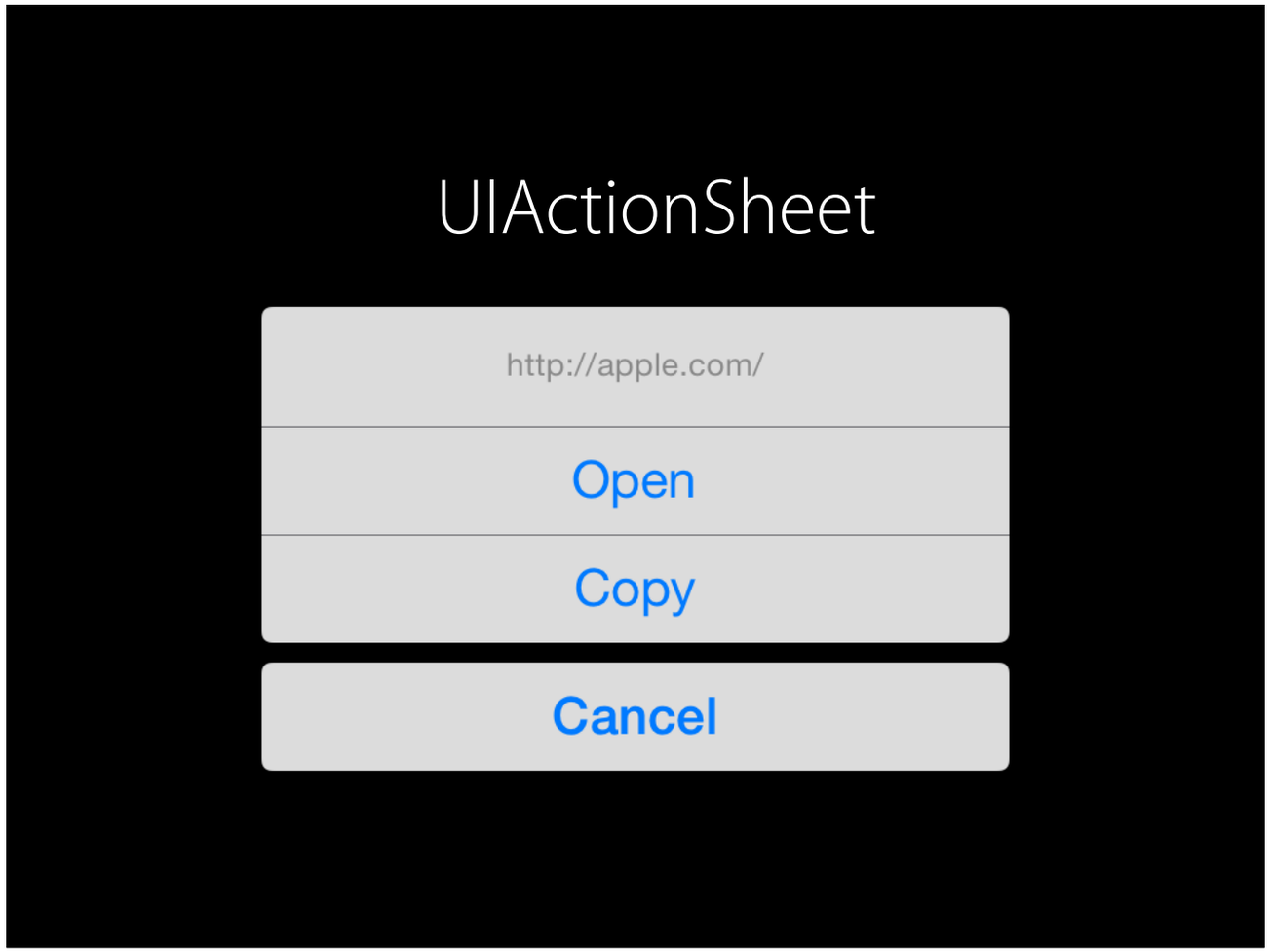
I’ve come across some code where the API could be improved. I rewrote the interfaces to those classes and will compare them here, showing you why I made the choices I did.

UIActionSheet & UIAlertView

On iOS, UIKit, the user interface framework provides two very similar classes: UIActionSheet and UIAlertView.

The API for these classes are much more complicated than it needs to be and is often misleading. For such straightforward classes conceptually, the code is not as straightforward.

So I rewrote it.

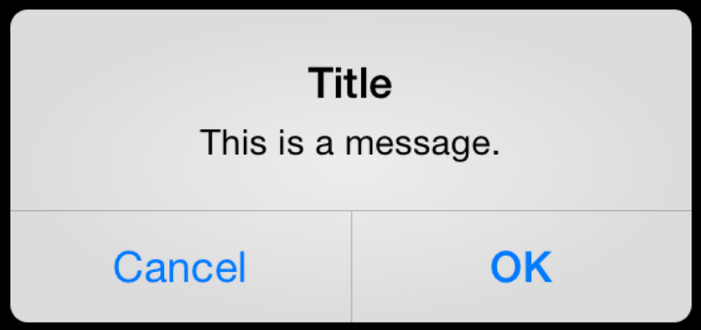


The UIActionSheet class is used to present the user with a set of alternatives for how to proceed with a given task.

You might be familiar with seeing this appear from the button of your iOS device when tapping on an action button, long-pressing on a link in Safari, etc.

The action sheet contains an optional title and one or more buttons, each of which corresponds to an action to take.

UIAlertView



The UIAlertView class is used to display an alert message to the user. An alert view functions similarly to, but differs in appearance from, an action sheet.

The Current API

These are fairly common user interface patterns. I think you're familiar with them and can imagine how you would expect the API to work. We'll see if you're right.

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

Here we have the init method, which is similar to that of a constructor in other languages.

This is one method. Some people have an issue with long, verbose method names often found in Objective-C, and that's sort of by design, but this is a clear offender.

You have five arguments and a lot of them can be NULL, so why force them all into one long method name. It's useless.

Also, a nil-terminated list? That's really inelegant because the calling code might want to compose the titles dynamically into a data structure and pass it along. If it's a nil-terminated list in a method, it must be done at compile-time.

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

Here we have some basic things about the title, style, and delegate of the action sheet. That's fine.

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

Here we have some methods for adding a button. And wait. What? There's a method for finding the title of a button at a specific index. If I just created it, why should I ever need to go back and check it? Seems very funky. So if something goes wrong with initializing it, you're expected to and reach into the data structure and see if it's OK. No, that's not OK.

Again, we don't need to know the number of buttons since the calling class should know that.

The index of the cancel button and destructive button. Those should be placed consistently and automatically, so the calling class should not need to mess with this.

Again, more things relating to indexes, which you shouldn't ever need to know about.

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

Here’s some methods for showing the action sheet. That’s fine.

```
@interface UIActionSheet : UIView

- (id)initWithTitle:(NSString *)title delegate:
(id<UIActionSheetDelegate>)delegate cancelButtonTitle:(NSString
*)cancelButtonTitle destructiveButtonTitle:(NSString *)destructiveButtonTitle
otherButtonTitles:(NSString *)otherButtonTitles, ...
NS_REQUIRES_NIL_TERMINATION;

@property (nonatomic, assign) id<UIActionSheetDelegate> delegate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic) UIActionSheetStyle actionSheetStyle;

- (NSInteger)addButtonWithTitle:(NSString *)title;
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex;
@property (nonatomic, readonly) NSInteger numberOfButtons;
@property (nonatomic) NSInteger cancelButtonIndex;
@property (nonatomic) NSInteger destructiveButtonIndex;

@property (nonatomic, readonly) NSInteger firstOtherButtonIndex;
@property (nonatomic, readonly, getter=isVisible) BOOL visible;

- (void)showFromToolbar:(UIToolbar *)view;
- (void)showFromTabBar:(UITabBar *)view;
- (void)showFromBarButtonItem:(UIBarButtonItem *)item animated:(BOOL)animated;
- (void)showFromRect:(CGRect)rect inView:(UIView *)view animated:(BOOL)animated;
- (void)showInView:(UIView *)view;

- (void)dismissWithClickedButtonIndex:(NSInteger)buttonIndex animated:
(BOOL)animated;

@end
```

And now a method to fake a “click” of a specific button index. Why someone would want to use the ideal way of describing each button, I have no idea.

If I were to instruct a user on how to do that on the control itself, I wouldn’t say here’s an action sheet, tap the button at the third index. No. I’d say tap the button with this title.

Also, on iOS, there’s no “clicking”, so the name is just wrong.

Using the current API:

Let’s take a look at how one would use the current API for UIActionSheet provided by UIKit.

```
UIActionSheet *actionSheet = [[UIActionSheet alloc]
         initWithTitle:@"http://apple.com/"
              delegate:self
     cancelButtonTitle:@"Cancel"
destructiveButtonTitle:nil
     otherButtonTitles:@"Open", @"Copy", nil];

[actionSheet showInView:self.view];
```

The initializer is overwhelmingly complicated. You're required to set everything up all at once and often leave stuff empty with "nil".

```
UIActionSheet *actionSheet = [[UIActionSheet alloc]
         initWithTitle:
              delegate:
     cancelButtonTitle:
destructiveButtonTitle:
     otherButtonTitles:

[actionSheet showInView:self.view];

- (void)actionSheet:(UIActionSheet *)actionSheet
didDismissWithButtonIndex:(NSInteger)buttonIndex
{
   switch (buttonIndex) {
      case 0: // Open
         [self openActionSheetButtonTapped];
         break;
      case 1: // Copy
         [self copyActionSheetButtonTapped];
         break;
      default:
         break;
   }
}
```

This API also requires the user to constantly manage the order of buttons and their indices. This is complicated for just one possible action sheet in a view controller. If you have multiple, you have to then switch on different action sheets and then on their indices. It quickly becomes quite a mess!

Using the rewritten API:

Now let me show you a similar example using my rewritten API:

Using the rewritten API:

```
MTZActionSheet *as = [[MTZActionSheet alloc] init];
as.title = @"http://apple.com/";
as.cancelButtonTitle = @"Cancel";
[as addButtonWithTitle:@"Open"
           andSelector:@selector(openButtonTapped)];
[as addButtonWithTitle:@"Copy"
           andSelector:@selector(copyButtonTapped)];

[as showInView:self.view];
```

The initializer here is simple. Everything else is just customized afterwards.
And unlike UIActionSheet, instead of asking the delegate to handle mapping indices to buttons, the action sheet performs selectors automatically. Blocks can also be used for even concentrated code.

This is a very common case, and it's much cleaner, clearer, and concise. It's been optimized for cases where action sheets should be used.

Current

```
UIActionSheet *actionSheet = [[UIActionSheet alloc]
         initWithTitle:@"http://apple.com/"
              delegate:self
     cancelButtonTitle:@"Cancel"
destructiveButtonTitle:nil
     otherButtonTitles:@"Open", @"Copy", nil];

[actionSheet showInView:self.view];

- - - - - - - - - - - - - - - - - - - - - - - - -

- (void)actionSheet:(UIActionSheet *)actionSheet
didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    switch (buttonIndex) {
        case 0: // Open
            [self openActionSheetButtonTapped];
            break;
        case 1: // Copy
            [self copyActionSheetButtonTapped];
            break;
        default:
            break;
    }
}
```

Rewritten

```
MTZActionSheet *as = [[MTZActionSheet alloc] init];
as.title = @"http://apple.com/";
as.cancelButtonTitle = @"Cancel";
[as addButtonWithTitle:@"Open"
          andSelector:@selector(openButtonTapped)];
[as addButtonWithTitle:@"Copy"
          andSelector:@selector(copyButtonTapped)];

[as showInView:self.view];
```

Let's see how it passes the test, meeting the goals I mentioned earlier.

For those familiar with other Frameworks APIs commonly used in Objective-C, this feels right at home. It uses familiar patterns as well as some up and coming ones to be more modern.

It's specific. While it could definitely have more features, this class is designed for something specific, and nothing more.

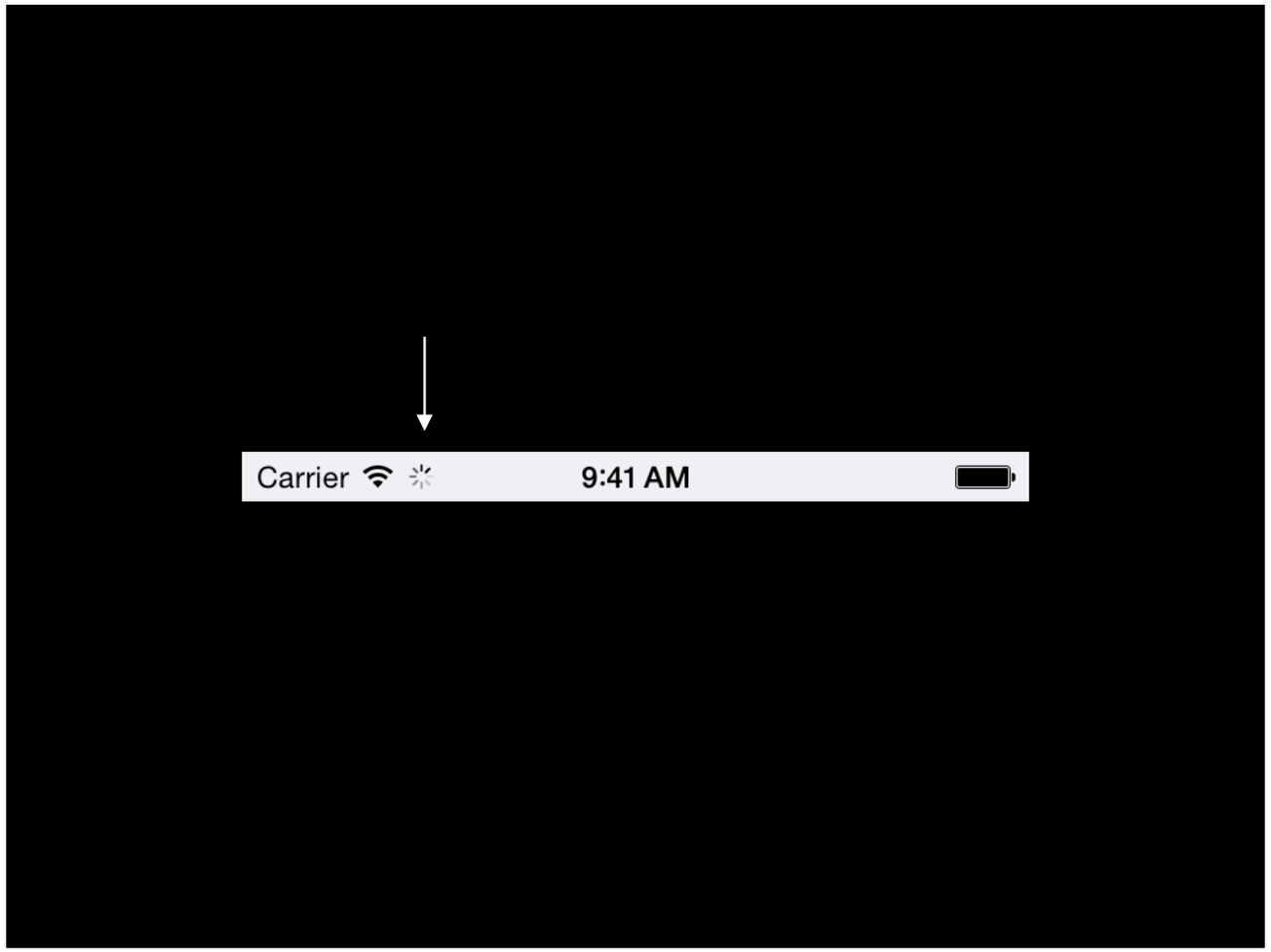
This is much more understandable. All relevant code is next to each other, instead of being spread across the file.

This code is much easier to write because the methods are closer to what you would imagine this class would implement.

This API is harder to misuse because it doesn't give you misleading API and the available API leads you to using it a certain way.

Network Activity Indicator

I'm going to give you another example in UIKit. The network activity indicator.



This indicator is the spinner that appears in the status bar. Applications can tell this item whether or not network activity is occurring and it can help the user debug issues. It's also conveniently located next to other information regarding cellular signal and strength.

Over the years, I've noticed more and more apps not using this or misusing this. This is a sign of bad API.

Seems like a pretty simple component, so how hard could it be to control it? Well let's take at the current API.

Current API

```
@interface UIApplication : UIResponder

...

@property (nonatomic,
getter=isNetworkActivityIndicatorVisible) BOOL
networkActivityIndicatorVisible

...

@end
```

It's just one property. Seems pretty simple. However, this is a network activity indicator. Networking code should be multithreaded, so it's possible that multiple things are going on once, starting and stopping. Since it's a boolean, the code is responsible for turning it on and back off when it's done. So when it's turning it off, how does that networking code know that everything's completed? It can't. Well, not without some other code managing it, but why set that up just for this? This is why a lot of app developers have avoided it.

A Solution.

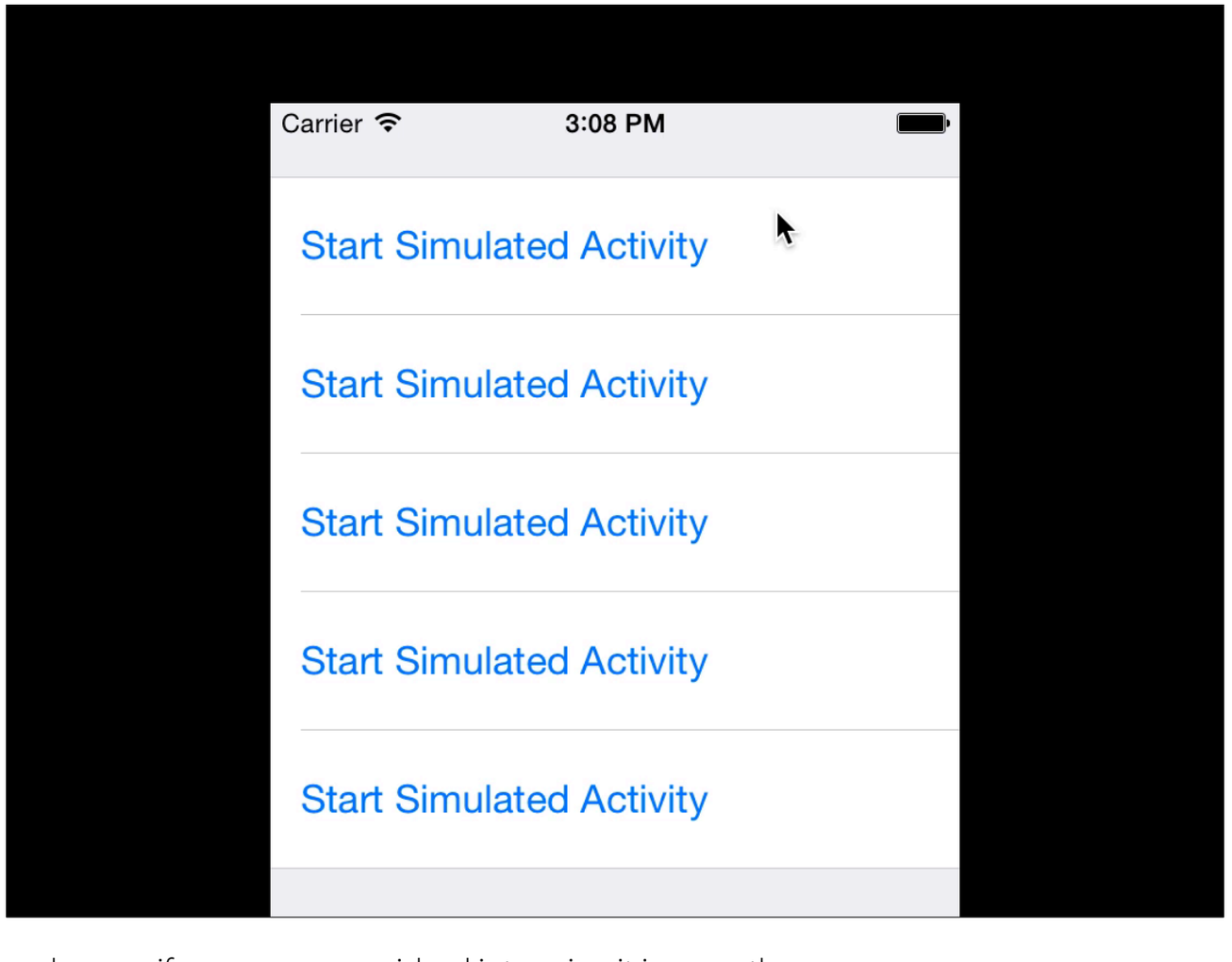
So the best way I thought of solving this issue is not being able to set the boolean property at all. There's no reason to do that directly.

And instead, having two methods which indicate starting and stopping of network code.

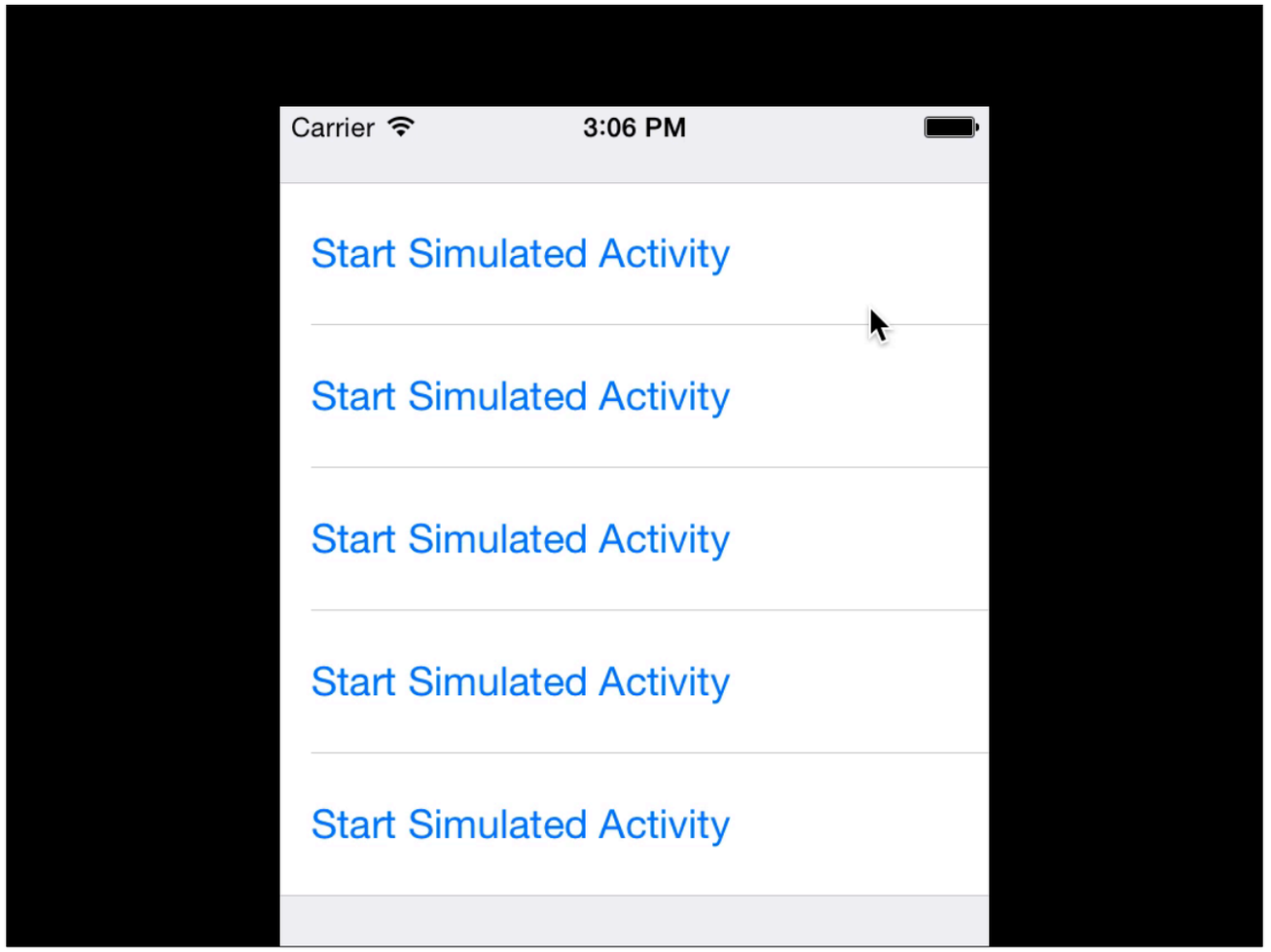
Rewritten API

```
@interface UIApplication : UIResponder

...

- (void)beganNetworkActivity;
- (void)endedNetworkActivity;

...

@end
```

Call beganNetworkActivity when networking activity. And when it ends, call endedNetworkActivity. Internally, the spinner will have a counter. The counter will increment and decrement, and when the counter is 0, no network activity is being performed, and the spinner doesn't display.



Here's the original code running on an example appp if someone was mislead into using it incorrectly.



And simply switching setting the property to on and off to began and ended network activity yields desired results.

Bad API can *work* but that doesn't necessarily mean developers will use it correctly.

Thank you

Dr. Goldschmidt, Moorthy, Sean O'Sullivan, and RCOS.