Tweet

How To: Objective C Initializer Patterns

Thursday, February 27, 2014 | By Jake Jennings (@jakej) [23:02 UTC]

Initializer patterns are an important part of good Objective-C, but these best practices are often overlooked. It's the sort of thing that doesn't cause problems most of the time, but the problems that arise are often difficult to anticipate. By being more rigorous and conforming to some best practices, we can save ourselves a lot of trouble. In this article, we'll cover initialization topics in-depth, with examples to demonstrate how things can go wrong.

Part 1: Designated and Secondary Initializers
Part 2: Case Studies
Part 3: Designated and Secondary Initializer Cheat Sheet
Part 4: - initWithCoder:, + new, and - awakeFromNib

Part 1: Designated and Secondary Initializers

Designated initializers define how we structure our initializers when subclassing; they are the "canonical initializer" for your class. A designated initializer does not define what initializer you should use when creating an object, like this common example:

```
1 [[UIView alloc] initWithFrame:CGRectZero];
```

It's not necessary to call the designated initializer in the above case, although it won't do any harm. If you are conforming to best practices, It is valid to call any designated initializer in the superclass chain, and the designated initializer for every class in the hierarchy is guaranteed to be called. For example:

```
1 [[UIView alloc] init];
```

is guaranteed to call [NSObject init] and [UIView initWithFrame:], in that order. The order is guaranteed to be reliable regardless of which designated initializer in the superclass chain you call, and will **always go from furthest ancestor to furthest descendant**.

When subclassing, you have three valid choices: you may choose to reuse your superclass's designated initializer, to create your own designated initializer, or to not create any initializers (relying on your superclass's).

If you override your superclass's designated initializer, your work is done. You can feel safe knowing that this initializer will be called.

If you choose to create a new designated initializer for your subclass, you must do two things. First, create a new initializer, and document it as the new designated initializer in your header file. Second, you must override your superclass's designated initializer and call the new one. Here's an example for a UIView subclass:

```
// Designated initializer
- (instancetype)initWithFoo:(TwitterFoo *)foo
{
    if (self = [super initWithFrame:CGRectZero]) {
        _foo = foo;
        // initializer logic
    }
    return self;
}

- (instancetype)initWithFrame:(CGRect)rect
{
    return [self initWithFoo:nil];
}
```

Apple doesn't mention much about it in the documentation, but all Apple framework classes provide valuable guarantees due to their consistency with these patterns. In the above example, if we did not override our superclass's designated initializer to call the new one, we would break the guarantee which makes calling any designated initializer in the hierarchy reliable. For example, if we removed our initWithFrame: override,

```
[[TwitterFooView alloc] init];
```

could not be relied upon to call our designated initializer, initWithFoo:. The initialization would end with initWithFrame:.

Finally, not all initializers are designated initializers. Additional initializers are referred to as convenience or secondary initializers. There is one rule here you'll want to follow: Always call the designated initializer (or another secondary initializer) on **self** instead of super.

Example 1:

```
<a href="https://twitter.com/intent/user?screen_name=

<a href="https://twitter.com/intent/user?screen_name=e

// Designated Initializer
- (instancetype)initWithFoo:(TwitterFoo *)foo
{
    if (self = [super initWithFrame:CGRectZero]) {
        _foo = foo;
        // do the majority of initializing things
    }
    return self;
}

// Super override
- (instancetype)initWithFrame:(CGRect)rect
{
    return [self initWithFoo:nil];
}

// Instance secondary initializer
- (instancetype)initWithBar:(TwitterBar *)bar
{
    if (self = [self initWithFoo:nil]) {
        _bar = bar;
        // bar-specific initializing things
    }
    return self;
}

// Class secondary initializer
```

```
+ (instancetype)fooViewWithBaz:(TwitterBaz *)baz
{
    TwitterFooView *fooView = [[TwitterFooView alloc]
    if (fooView) {
        // baz-specific initialization
}
    return fooView;
}

<a href="https://twitter.com/intent/user?screen_name=e
```

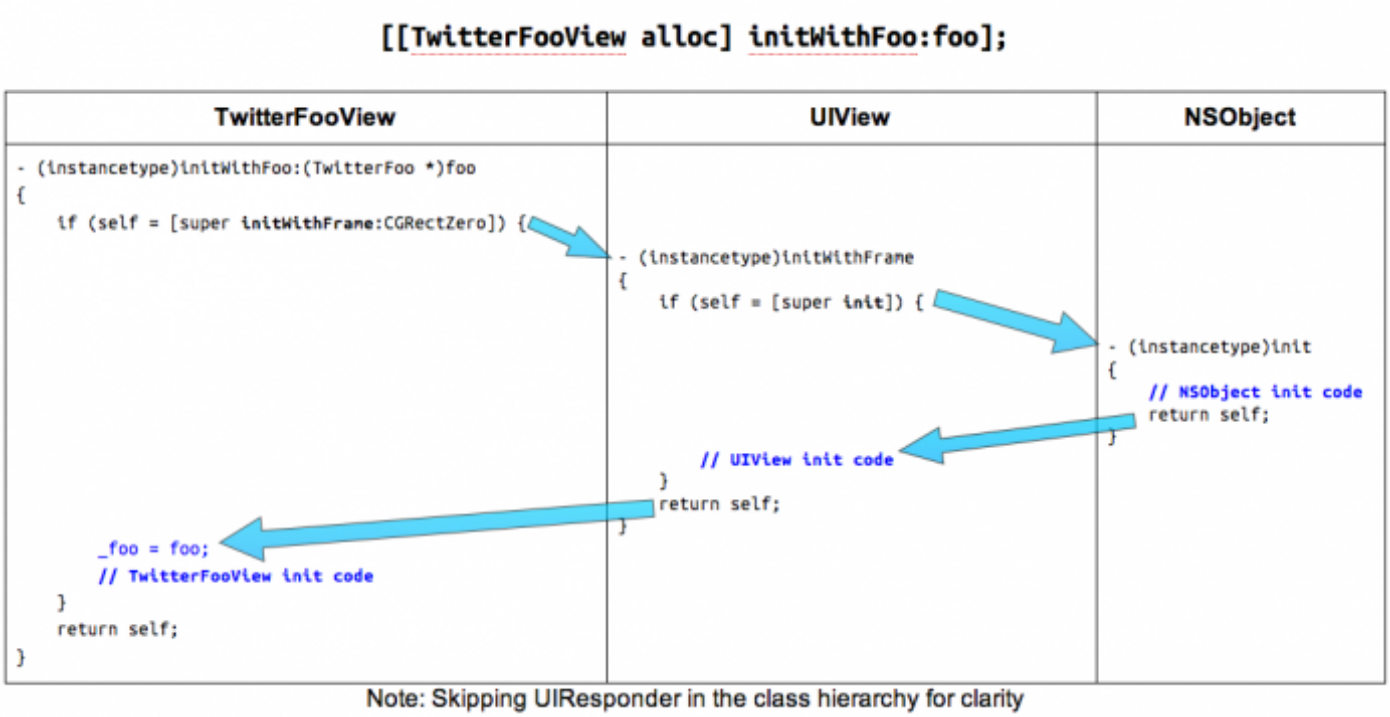
Again, the key takeaway from this example is that in both - initWithBar: and + fooViewWithBaz:, we call - initWithFoo:, the designated initializer, on self. There's one more rule to follow to preserve a deterministic designated initializer behavior: When writing initializers, don't call designated initializers beyond your direct superclass. This can break the order of designated initializer execution. For an example of how this can go wrong, see Part 2, Case 2.

Part 2: Case Studies

Now that we've covered the rules and guarantees relating to designated and secondary initializers, let's prove these assertions using some concrete examples.

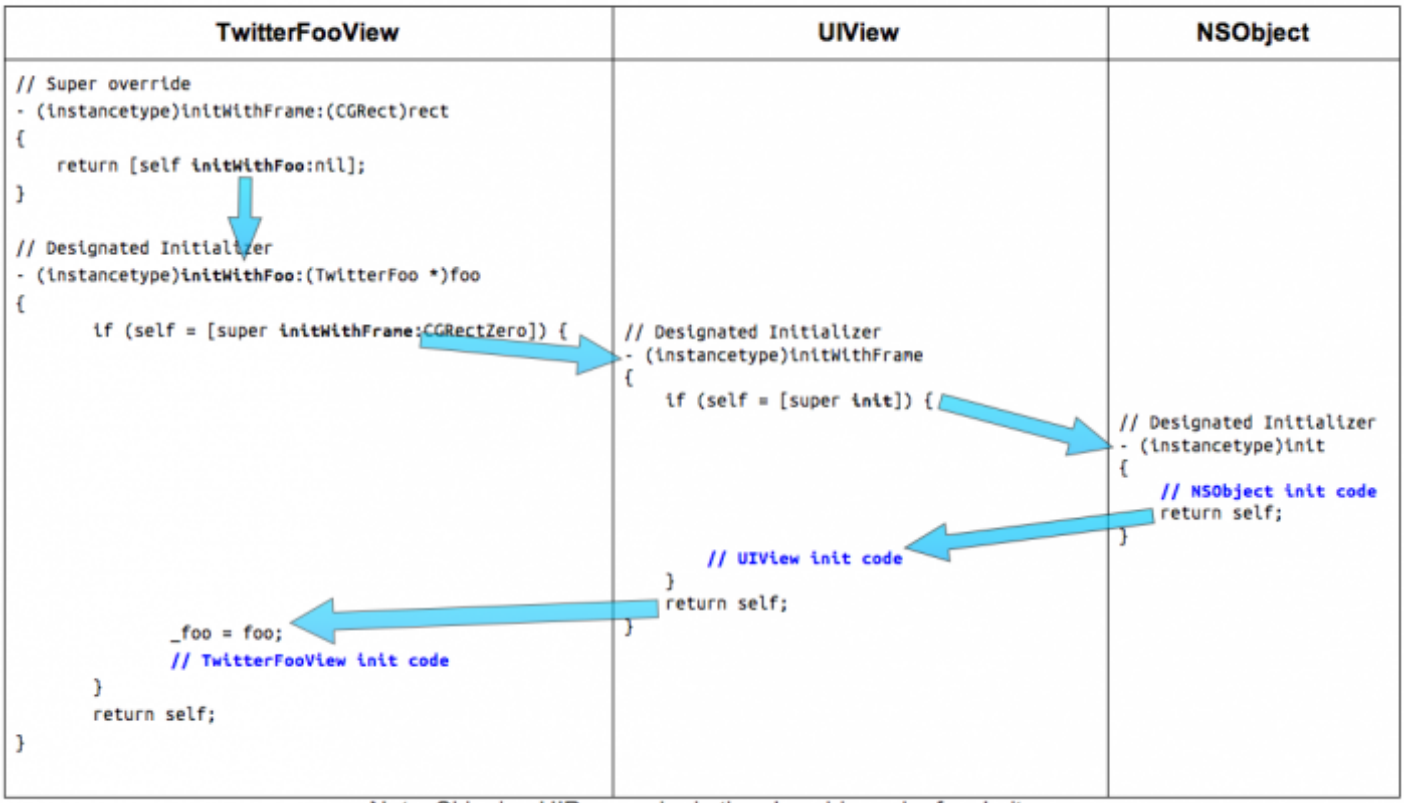
Case 1: Designated Initializer Ordering

Based on the code from Example 1, let's prove the following assertion: Calling any designated initializer in the superclass chain is valid, and designated initializers are guaranteed to be executed in order from furthest ancestor ([NSObject init]) to furthest descendant ([TwitterFooView initWithFoo:]). In the following three diagrams, we'll show the order of initializer execution when calling each designated initializer in the hierarchy: initWithFoo:, initWithFrame:, and init.



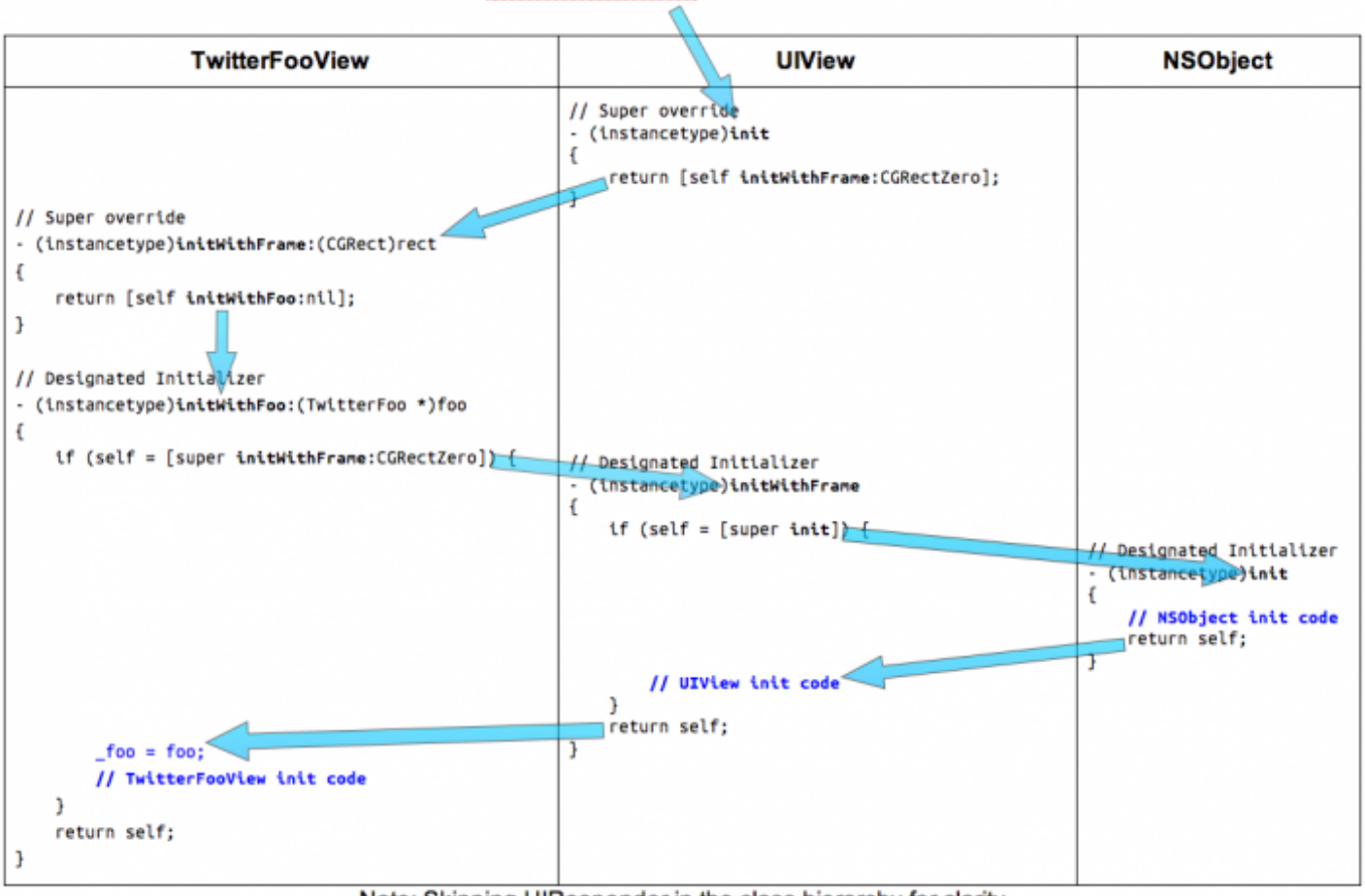
Note: Skipping UIResponder in the class hierarchy for clarity

[[TwitterFooView alloc] initWithFrame:CGRectZero]:



Note: Skipping UIResponder in the class hierarchy for clarity

[[TwitterFooView alloc] init]:



Note: Skipping UIResponder in the class hierarchy for clarity

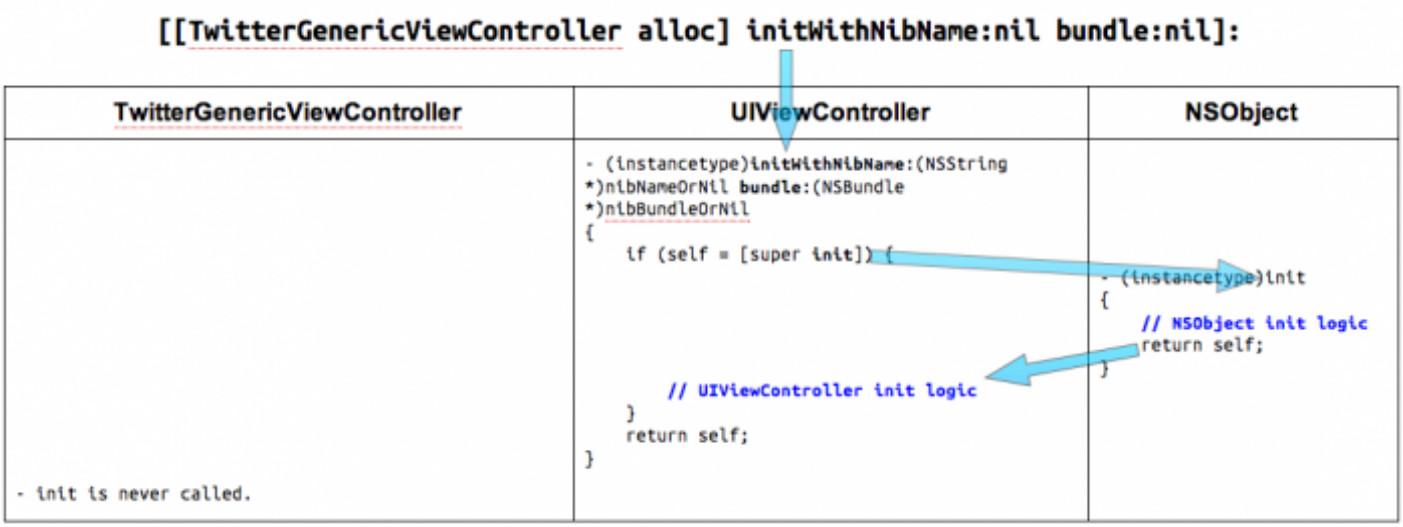
Case 2: Example bug in UIViewController subclass

In the following example, we will analyze what can happen when we violate the following rule: When writing initializers, don't call designated initializers beyond your immediate superclass. In context, this means we shouldn't override or call [NSObject init] from a UIViewController subclass.

Let's say we start with a class TwitterGenericViewController, and incorrectly override [NSObject init]:

```
<a href="https://twitter.com/intent/user?screen_name=?

// Incorrect
- (instancetype)init
{
    if (self = [super init]) {
        _textView = [[UITextView alloc] init];
        _textView.delegate = self;
    }
    return self;
}

<a href="https://twitter.com/intent/user?screen_name=e
```

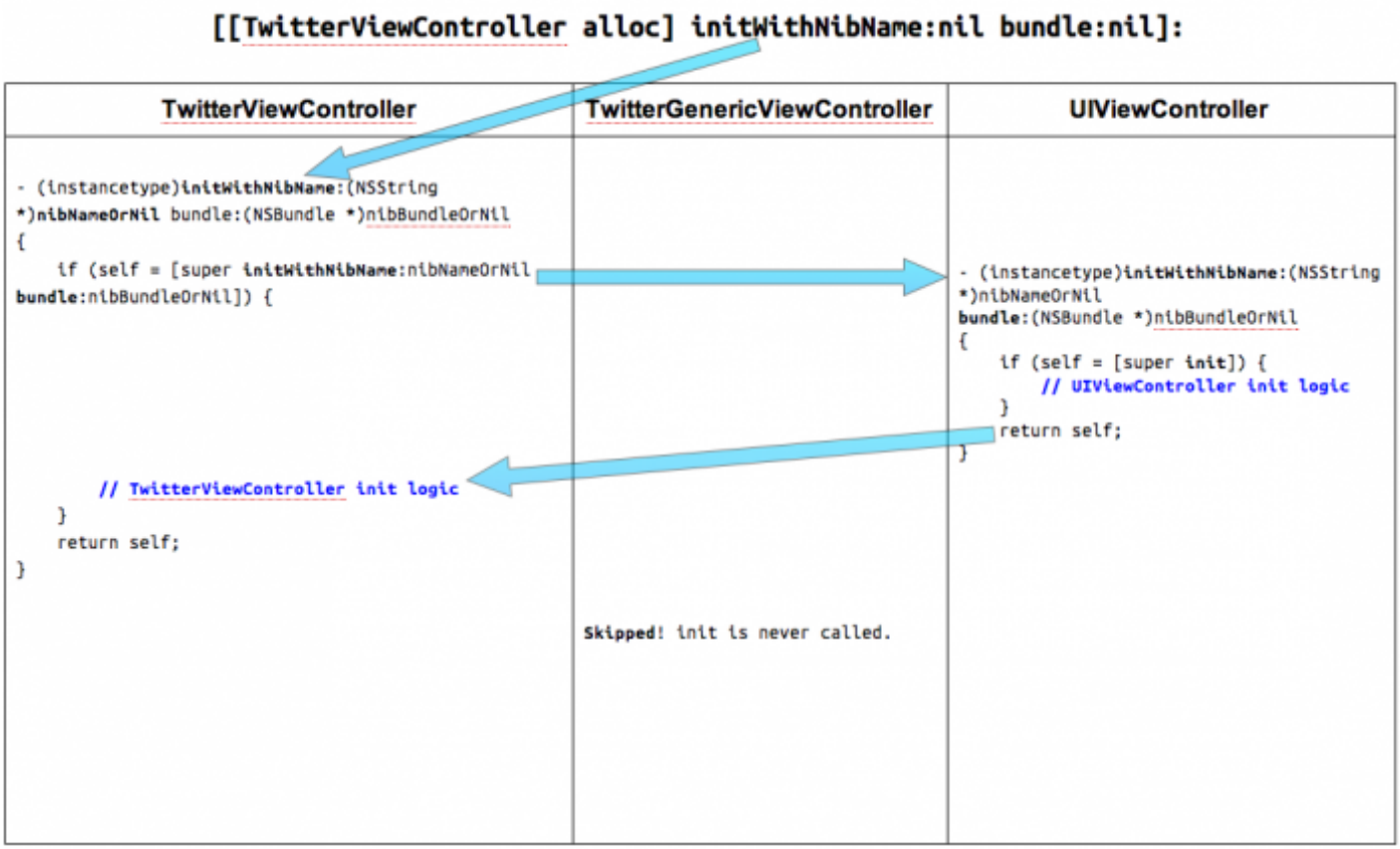
If we instantiate this object using [[TwitterGenericViewController alloc] init], this will work fine. However, if we use [[TwitterGenericViewController alloc] initWithNibName:nil bundle:nil], which should be perfectly valid, this initializer method will never be called. Let's look at the order of execution for this failure case:



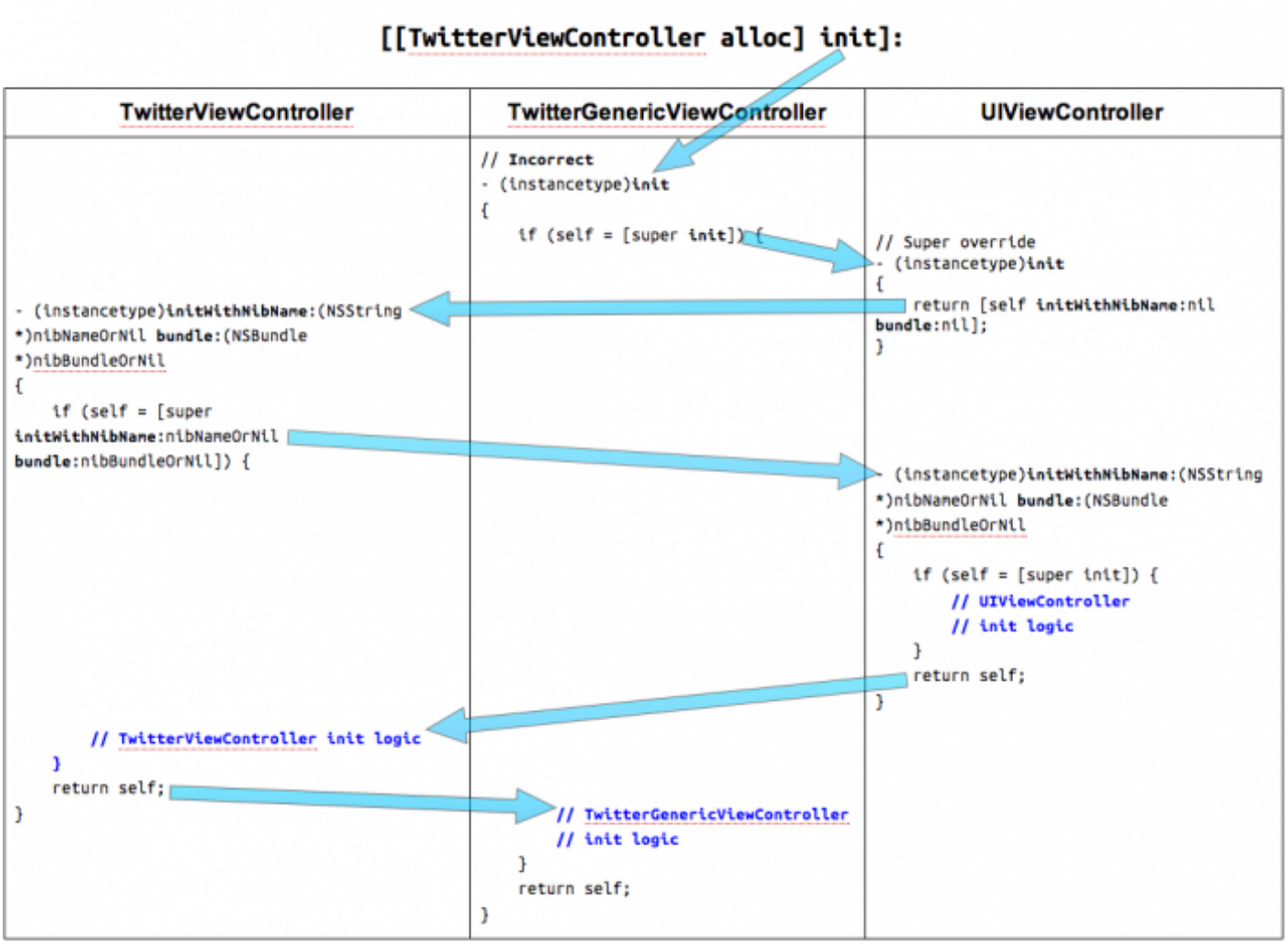
Things begin to break even further when subclasses are introduced below this incorrect - init implementation. Consider the following subclass to TwitterGenericViewController which correctly overrides initWithNibName:bundle:

```
<a href="https://twitter.com/intent/user?screen_name=?

- (instancetype)initWithNibName:(NSString *)nibNameOrN
{
    if (self = [super initWithNibName:nibNameOrNil bun
        _textView = [[UITextView alloc] init];
        _textView.delegate = self;
    }
    return self;
}

<a href="https://twitter.com/intent/user?screen_name=e
```

Now, we have failure no matter which initializer we choose.



In this case, there is a failure because TwitterGenericViewController's initializer is never called.



In this case, all initializers are called, but in the wrong order. TwitterViewController will populate _textView and set it's delegate, and then TwitterGenericViewController (the superclass) will initialize, overriding the _textView configuration. That's backwards! We always want subclasses to initialize after superclasses, so we can override state properly.

Part 3: Designated and Secondary Initializer Cheat Sheet

When creating an object

Calling any designated initializer in the superclass chain is valid when creating an object. You can rely on all designated initializers being calling in order from furthest ancestor to furthest descendant.

When subclassing
Option 1: Override immediate superclass's designated initializer

Be sure to call immediate super's designated initializer first
Only override your immediate superclass's designated initializer

Option 2: Create a new designated initializer

Be sure to call your immediate superclass's designated initializer first
Define a new designated initializer
Document new designated initializer as such in your header
Separately, override immediate superclass's designated initializer and call the new designated initializer on self

Option 3: Don't create any initializers

It is valid to omit any initializer definition and rely on your superclass's
In this case, you 'inherit' your superclass's designated initializer as it applies to the last rule in Option 1

Additionally, you may define class or instance secondary initializers

Secondary initializers, must always call self, and either call the designated initializer or another secondary initializer.
Secondary initializers may be class or instance methods (see Example 1)

Part 4: - initWithCoder:, + new, and - awakeFromNib

+ new

The documentation describes the [NSObject new] as "a combination of alloc and init". There's nothing wrong with using the method for initialization, since we've established that calling any designated initializer in the hierarchy is valid, and all designated initializers will be called in order. However, when contrasted with [[NSObject alloc] init], + new is used less often, and is therefore less familiar. Developers using Xcode's global search for strings like "MyObject alloc" may perhaps unknowingly overlook uses of [MyObject new].

- initWithCoder:

Reading Apple's documentation on object initialization when using NSCoding is a helpful first step.

There are two key things to remember when implementing initWithCoder: First, if your superclass conforms to NSCoding, you should call [super initWithCoder:coder] instead of [super (designated initializer)].

There's a problem with the example provided in the documentation for initWithCoder:, specifically the call to [super (designated initializer)]. If you're a direct subclass of NSObject, calling [super (designated initializer)] won't call your [self (designated initializer)]. If you're not a direct subclass of NSObject, and one of your ancestors implements a new designated initializer, calling [super (designated initializer)] WILL call your [self (designated initializer)]. This means that apple's suggestion to call super in initWithCoder encourages non-deterministic initialization behavior, and is not consistent with the solid foundations laid by the designated initializer pattern. Therefore, my recommendation is that you should treat initWithCoder: as a secondary initializer, and call [self (designated initializer)], not [super (designated initializer)], if your superclass does not conform to NSCoding.

-awakeFromNib

The documentation for - awakeFromNib is straightforward:

> NSNibAwaking Protocol Reference

The key point here is that interface builder outlets will not be set while the designated initializer chain is called. awakeFromNib happens afterwards, and IBOutlets will be set at that point.

Documentation

NSCell has four designated initializers for different configurations. This is an interesting exception to the standard single designated initializer pattern, so it's worth checking out:

> NSCell Class Reference

Documentation relating to initialization:

> Object Initialization
>
> Multiple Initializers
>
> Encapsulating Data
>
> Encoding and Decoding Objects