Cocoa Programming Guidelines

Wojciech Nagrodzki

May 2, 2014

Special thanks to: Krzysztof Profic, Bartłomiej Hyży, Grzegorz Wikiera, Piotr Węgrzynek and Daniel Garbień

Contents

Code Appearance

Hardly any software is maintained by its original author for its whole life. Writing code in an unified way helps engineers to understand it more quickly.

Class names are not prefixed usually

Prefixes are used only for classes which are to be shared between applications.

Protocol names indicates behavior

Most protocols group related methods that aren't associated with any class in particular. This type of protocol should be named so that the protocol won't be confused with a class. A common convention is to use a gerund ("...ing") form:

• NSLocking	Good
• NSLock	Poor (seems like a name for a class)

Some protocols group a number of unrelated methods (rather than create several separate small protocols). These protocols tend to be associated with a class that is the principal expression of the protocol. In these cases, the convention is to give the protocol the same name as the class. An example of this sort of protocol is the `NSObject` protocol.

Header files follow a structure

Each header file should be succinct, and comply with the following declaration's order:

1. Imports
2. Forward class declarations
3. Enumeration types
4. Constants
5. Notification names and its user info dictionary keys
6. Delegate protocol
7. Data source protocol
8. Class interface

A class interface has a fixed structure as well:

1. Properties
2. Data source property
3. Delegate property
4. Class methods
5. Instance methods

Please pay attention to the linespacing, it is also a rule.

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

@class ExampleClass;


typedef NS_ENUM(NSInteger, Enumeration) {
    EnumerationInvalid,
    EnumerationA,
};

extern CGFloat const ExampleClassDefaultHeight;

extern NSString * const ExampleClassWillPerformActionNotification;
extern NSString * const ExampleClassDidPerformActionNotification;
extern NSString * const ExampleClassActionNameKey;


@protocol ExampleClassDelegate <NSObject>

- (void)exampleClass:(ExampleClass *)exampleClass didPerformAction:(Action *)action;

@optional

- (void)exampleClass:(ExampleClass *)exampleClass willPerformAction:(Action *)action;

@end


@protocol ExampleClassDataSource <NSObject>

- (NSInteger)exampleClassNumberOfActions:(ExampleClass *)exampleClass;

@end


@interface ExampleClass : NSObject

@property (strong, nonatomic) NSURL * initialProperty;
@property (weak, nonatomic) id<ExampleClassDataSource> dataSource;
@property (weak, nonatomic) id<ExampleClassDelegate> delegate;

+ (id)exampleClassWithInitialProperty:(NSURL *)initialProperty;
- (id)initWithInitialProperty:(NSURL *)initialProperty;
- (void)performAction;

@end
```

Implementation files are divided by pragma marks

Methods in an implementation file appear in the same order as they are declared in the header. They are also grouped by pragma marks similar to the following:

```
#pragma mark - Public Properties
#pragma mark - Public Class Methods
#pragma mark - Public Instance Methods
#pragma mark - IBActions
#pragma mark - Overridden
#pragma mark - Private Properties
#pragma mark - Private Class Methods
#pragma mark - Private Instance Methods
#pragma mark - Protocols
#pragma mark - Notifications
```

In need of greater granularity:

```
#pragma mark - Overridden (UIView)
#pragma mark - Overridden (UIContainerViewControllerCallbacks)
#pragma mark - Overridden (UIViewControllerRotation)
```

Tip: Create a code snipped to help you use the same pragma marks through all implementation files.

Property attributes are kept in an order

The attributes are kept in the same order through all property declarations.

```
[assign | weak | strong | copy] + [nonatomic | atomic] + [readonly | readwrite] + [getter = ]
```

None of the attributes can be omitted with the exception of readwrite.

```
@property (assign, nonatomic) CGFloat height;
@property (strong, nonatomic) UIColor * color;
@property (copy, nonatomic) NSString * name;
@property (weak, nonatomic) id <UITableViewDelegate> delegate;
```

Getters for boolean properties, if they are adjectives, are renamed in the following manner:

```
@property (assign, nonatomic, getter = isVisible) BOOL visible;
@property (assign, nonatomic, getter = isEnabled) BOOL enabled;
@property (assign, nonatomic, getter = isTracking) BOOL tracking
```

Protocols and constants are prefixed with a class name

Protocols, notification names, enumeration types and other constants that refer to a particular class are prefixed with the name of that class.

```
typedef NS_ENUM(NSInteger, UITableViewStyle) {
    UITableViewStylePlain,
    UITableViewStyleGrouped
};

UIKIT_EXTERN NSString *const UITableViewIndexSearch;

UIKIT_EXTERN const CGFloat UITableViewAutomaticDimension;

UIKIT_EXTERN NSString *const UITableViewSelectionDidChangeNotification;

@protocol UITableViewDelegate<NSObject, UIScrollViewDelegate>
...
```

IBOutlets are declared privately

Outlets are defined as weak properties at the top of a class extension, and divided by one line from the other properties.

```
@interface PanelViewController ()

@property (weak, nonatomic) IBOutlet UIButton * infoButton;
@property (weak, nonatomic) IBOutlet UIButton * closeButton;
@property (weak, nonatomic) IBOutlet UILabel * descriptionLabel;

@property (weak, nonatomic) UIView * overlayView;

@end
```

Classes may have additional header file for subclasses

Private methods and properties are never exposed. To provide a subclass access to them, they have to be declared in a category entitled ForSubclassEyesOnly. It should be placed in a separate header file, named in the following manner: [Class Name] + Subclass.h.

Tip: UIGestureRecognizerSubclass.h is a good example of that approach.

Methods return early and often

A nesting usually makes code harder to read.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password
{
    if (user.length >= 6) {
        if ((password.length >= 8) {
            // the actual logging code
        }
    }
    return NO;
}
```

Getting invalid cases out of the way first will keep the working code with one level of indentation, and ensure that all parameters are valid. This paradigm is called The Golden Path.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password
{
    if (user.length < 6) return NO;
    if (password.length < 8) return NO;

    // actual logging code
}
```

This approach has one more advantage, it is easier to add an error handling later on.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password error:(NSError **)error
{
    if (user.length < 6) {
        if (error != NULL)
            *error = [NSError errorWithDomain:ExampleDomain
                                         code:1001
                                     userInfo:@{NSLocalizedDescriptionKey: @"User name must have..."}];
        return NO;
    }

    if (password.length < 8) {
        if (error != NULL)
            *error = [NSError errorWithDomain:ExampleDomain
                                         code:1002
                                      userInfo:@{NSLocalizedDescriptionKey: @"Password must have..."}];
        return NO;
    }

    // logging in code
}
```

Methods are not bisected with conditionals

The following method structure is not acceptable.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
    }
    else {
        // perform some other actions
    }
}
```

Bisection can be removed by returning within an if statement.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
        return;
    }

    // perform some other actions
}
```

In one case bisection is allowed.

```
- (void)method
{
    if (self.valueX == 10) {
        // perform some actions
    }
    else {
        // perform some other actions
    }

    // perform some actions no matter what
}
```

Methods can be prefixed with get

Get prefix is used only in situations when a value is returned indirectly via a memory address.

```
@interface NSValue : NSObject <NSCopying, NSSecureCoding>

- (void)getValue:(void *)value;

@end
```

General rules

Header files are imported only if necessary

Importing a header file is allowed:

- if class needs to conform to a protocol (header with protocol declaration)
- if class is inheriting from another class (header with superclass declaration)
- if class uses enums in its interface (header with enums declarations)

In any other case a forward declaration should be applied.

Delegate methods always pass the sender

A delegation method passes the sender as the first parameter. It is a good practice to use will/did paradigm.

```
- (NSInteger)exampleClassNumberOfActions:(ExampleClass *)exampleClass
- (void)exampleClass:(ExampleClass *)exampleClass willPerformAction:(Action *)action;
- (void)exampleClass:(ExampleClass *)exampleClass didPerformAction:(Action *)action;
```

Property's default values are documented conditionally

The `alloc` method clears out the memory allocated for an object's properties, by setting it to zero. If a property value is not altered during the initialization, a note about its default value may be omitted.

```
/*
 * A Boolean value that determines whether the view is hidden.
 */
@property(nonatomic, getter=isHidden) BOOL hidden                // represented by zeroed memory
```

On the contrary, if the default value is changed, a mention about it is included.

```
/*
 * Defines the anchor point of the layer's bounds rectangle.
 * The default value of this property is (0.5, 0.5).
 */
@property (nonatomic) CGPoint anchorPoint                      // not represented by zeroed memory
```

Init methods take only mandatory parameters

All settings required to a proper initialization are passed as initializer's parameters. They are later accessible through readonly properties.

```
@property (strong, nonatomic, readonly) DownloaderMode downloaderMode;

- (id)initWithDownloaderMode:(DownloaderMode)downloaderMode;
```

Tip: If you need a convenience method to create instances, consider creating factory methods.

Accessors are not used in init and dealloc

Instance subclasses may be in an inconsistent state during `init` and `dealloc` method execution, hence code in those methods must avoid invoking accessors.

```
- (id)init
{
    self = [super init];
    if (self) {
        _foo = [NSMutableSet set];
    }
    return self;
}

- (void)dealloc
{
    [_titleLabel removeObserver:self forKeyPath:@"text"];
}
```

Abstract classes can be faked with assertion

Creating instances of an abstract class is foiled by the following assertion.

```
- (id)init
{
    self = [super init];
    if (self) {
        NSAssert1([self isMemberOfClass:[MyAbstractClass class]] == NO,
                  @"%@ is an abstract class. Please do not create instances of it.", [self class]);
    }
    return self;
}
```

Abstract methods raise exceptions

Forcing a subclass to provide an implementation of a method is accomplished by raising an exception in superclass method.

```
- (void)abstractMethod
{
    [NSException raise:NSInternalInconsistencyException
                format:@”It’s template method. Implementation must be provided in subclass.”];
}
```

Enumeration types contain invalid value

The enumeration type equal to zero is considered as invalid. It protects instance variables from being initialized with a meaningful value when not intended.

```
typedef NS_ENUM(NSInteger, Enumeration) {
    EnumerationInvalid,
    EnumerationA,
    EnumerationB,
    EnumerationC,
};
```

It also shields from false positives, when comparing against values returned by methods sent to nil pointer. Let’s assume `EnumerationInvalid` does not exist.

```
@interface MyObject : NSObject

@property (assign, nonatomic) Enumeration enumeration;

@end
```

If `myObject` pointer is `nil`, `performActionA` method is called, and it may not be expected.

```
MyObject * myObject = ...

if (myObject.enumeration == EnumerationA)
    [self performActionA];
else if (myObject.enumeration == EnumerationB)
    [self performActionB];
else
    [self performActionC];
```

The highest level of abstraction is used by default

Lower levels are used only when more control is required. For example, instead using GCD:

```
dispatch_sync(dispatch_get_main_queue(), ^{
    // code
}
```

Use an operation queue.

```
[[NSOperationQueue mainQueue] addOperationWithBlock:^{
    // code
}];
```

Exceptions are not used to control flow

The general pattern is that exceptions are reserved for programming or unexpected runtime errors, such as out-of-bounds collection access, attempts to mutate immutable objects, sending an invalid message, or losing the connection to the window server. A program catching such exception should quit soon afterwards. Exceptions must not be used to control flow in favor of `NSError` objects. When developing a class or a framework exceptions are thrown to indicate that class or framework are being misused:

```
- (void)abstractMethod
{
    [NSException raise:NSInternalInconsistencyException
                format:@"It's template method, you need to implement it in your subclass"];
}
```

Lazy loading reduces memory footprint

Creating an object on demand reduces initialization time of containing class.

```
- (NSMutableDictionary *)cacheDictionary
{
    if (_cacheDictionary == nil) {
        _cacheDictionary = [NSMutableDictionary dictionary];
    }
    return _cacheDictionary;
}
```

No object register other objects as observers

Object registers only itself as an observer.

```
[obj addObserver:self forKeyPath:@"isExecuting" options:NSKeyValueObservingOptionNew context:NULL];
```

Unregistering follows the same rule.

```
[obj removeObserver:self forKeyPath:@"isExecuting" context:NULL];
```

Methods do not return NSError object

A failure is indicated by `nil` or `NO` returned by a method. Success similarly by `YES` or not-nil pointer.

```
- (id)initWithContentsOfURL:(NSURL *)aURL;
- (BOOL)writeToURL:(NSURL *)aURL atomically:(BOOL)atomically;
```

`NSError` object is used only for providing additional information about a failure.

```
- (id)initWithContentsOfURL:(NSURL *)aURL options:(NSDataReadingOptions)mask error:(NSError **)errorPtr;
- (BOOL)writeToURL:(NSURL *)aURL options:(NSDataWritingOptions)mask error:(NSError **)errorPtr
```

Therefore you should always check if the return value is `nil` or `NO` before attempting to do anything with an `NSError` object. Moreover, you ought to check the domain before examining the error code.

Custom errors belong to error domains

A custom error have both the error domain and error code defined.

```
extern NSString *const MyErrorDomain;

typedef NS_ENUM(NSInteger, MyErrorCode) {
    MyInvalidErrorCode
    MyErrorCode1,
    MyErrorCode2,
    MyUnknownErrorCode,
};
```

Both parameters including localized description are used during an error initialization.

```
if (error != NULL) {
    if(error_situation_1) {
        error* = [NSError errorWithDomain:MyErrorDomain
                                     code:MyErrorCode1
                                 userInfo:@{NSLocalizedDescriptionKey: @"Description of error 1"}];
    }
    else if (error_situation_2) {
        error* = [NSError errorWithDomain:MyErrorDomain
                                     code:MyErrorCode2
                                 userInfo:@{NSLocalizedDescriptionKey: @"Description of error 2"}];
    }
    else {
        error* = [NSError errorWithDomain:MyErrorDomain
                                     code:MyUnknownErrorCode
                                 userInfo:@{NSLocalizedDescriptionKey: @"Unknown error"}];
    }
}
```

Properties can be added to existing classes

A property is added to an existing class by using associated objects. Please pay attention to the way the key is defined.

```
static void * const navigationItemKey = (void *)&navigationItemKey;

- (void)setNavigationItem:(UINavigationItem *)navigationItem
{
    objc_setAssociatedObject(self,
                             navigationItemKey,
                             navigationItem,
                             OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (UINavigationItem *)navigationItem
{
    UINavigationItem * navigationItem = objc_getAssociatedObject(self, navigationItemKey);
    if (navigationItem == nil) {
        navigationItem = [[UINavigationItem alloc] init];
        self.navigationItem = navigationItem;
    }
    return navigationItem;
}
```

Concurrency

Recommended reading Concurrent Programming – objc.io

Important: To avoid priority inversion problems with Grand Central Dispatch use default queue priority DISPATCH_QUEUE_PRIORITY_DEFAULT in almost all cases.

Multiple readers one writer

A concurrent isolation queue is used to synchronize access to a property.

```
NSString * queueLabel = [NSString stringWithFormat:@"%@.isolation.%p", [self class], self];
self.isolationQueue = dispatch_queue_create([queueLabel UTF8String], DISPATCH_QUEUE_CONCURRENT);
```

Dispatch barrier async runs the block after all previously scheduled blocks are completed and before any following blocks are run.

```
- (void)setObject:(id)anObject forKey:(id <NSCopying>)aKey
{
    aKey = [aKey copyWithZone:NULL];
    dispatch_barrier_async(self.isolationQueue, ^{
        [self.mutableDictionary setObject:anObject forKey:aKey];
    });
}

- (id)objectForKey:(id)aKey
{
    __block id object;
    dispatch_sync(self.isolationQueue, ^{
        object = [self.mutableDictionary objectForKey:aKey];
    });
    return object;
}
```

NSOperation can be cancelled before it begins execution

The main method checks if an operation is cancelled at the very beginning, and interrupts execution if condition is true.

```
- (void)main
{
    if (self.isCancelled) return;

    // code
}
```

UIView

View is usually initialized in two ways

Either by calling `initWithFrame:`, or `initWithCoder:` method when it is unarchived form a nib file. Both situations are covered.

```
@interface ExampleView ()

@property (strong, nonatomic) UITextField * textField;

@end

@implementation ExampleView

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self initialize];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)aDecoder
{
    self = [super initWithCoder:aDecoder];
    if (self) {
        [self initialize];
    }
    return self;
}

- (void)initialize
{
    _textField = [[UITextField alloc] init];
    _textField.translatesAutoresizingMaskIntoConstraints = NO;
    [self addSubview:_textField];

    NSDictionary * views = NSDictionaryOfVariableBindings(_textField);
    [self addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|-8-[_textField]-8-|"
                                                                  options:0
                                                                  metrics:nil
                                                                    views:views]];
    [self addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|-8-[_textField]-8-|"
                                                                  options:0
                                                                  metrics:nil
                                                                    views:views]];
}
```

Note that we do not implement `encodeWithCoder:` method. `UIView`s and `UIViewController`s does not follow normal serialization process, their state is persisted in a model.

The interface of generic view

The view exposes a minimal set of properties and methods that are required for its configuration. If possible, a property is not backed up by an instance variable, but rather by view's or subview's property.

```
- (NSString *)name
{
    return self.nameLabel.text;
}

- (void)setName:(NSString *)name
{
    self.nameLabel.text = name;
}
```

The view is independent of a model. A mapping between view's and model's properties is kept in a category on that view. This way a view controller is kept cleaner.

```
@implementation UITableViewCell (Person)

- (void)configureWithPerson:(Person *)person
{
    self.imageView.image = person.photo;
    self.textLabel.text = person.name;
}
```

The interface of specific view

The specific view is tightly coupled with model. It exposes one method that takes a model object and configures the view.

```
@interface PersonTableViewCell : UITableViewCell

- (void)configureWithPerson:(Person *)person

@end
```

UIViewController

Properties affecting user interface

It is a common practice to have properties on view controller that influence its view. Putting view adjusting code in the setter is not enough, as view may be not loaded. After it is loaded, view controller should adjust it according to the property. In result it is convenient to create a separate method for adjusting the view as it will be called from the setter and `viewDidLoad` as well.

```
- (void)setClient:(Client *)client
{
    _client = client;
    [self adjustUserInterfaceForClient:client];
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self adjustUserInterfaceForClient:self.client];
}

- (void)adjustUserInterfaceForClient:(Client *)client
{
    if (self.isViewLoaded == NO) return;
    // code adjusting interface
}
```

View controller containment

Container specific items are provided by a property

If a container requires its child to have specific properties, it delivers a category on `UIViewCon-troller`. The category provides an item which encapsulates all of the needed properties.

```
- (UINavigationItem *)navigationItem
{
    UINavigationItem * navigationItem = objc_getAssociatedObject(self, kNavigationItemKey);
    if (navigationItem == nil) {
        navigationItem = [[UINavigationItem alloc] init];
        self.navigationItem = navigationItem;
    }
    return navigationItem;
}
```

Whenever container needs to stay in sync with child's properties, it uses Key Value Observing.

Container is accessible from contained view controllers

Child view controllers can access the container through a property.

```
@interface UIViewController (UINavigationControllerItem)

@property(nonatomic,readonly,retain) UINavigationController *navigationController;

@end
```

The following getter implementation traverses view controller hierarchy, and returns the closest parent view controller of the container class.

```
- (UINavigationController *)navigationController
{
    UIViewController * controller = self.parentViewController;
    while (controller != nil && [controller class] != [UINavigationController class]) {
        controller = controller.parentViewController;
    }
    return (UINavigationController *)controller;
}
```

Core Data

Important: NSManagedObject's properties must not be prefixed with "new". Due to @dynamic compiler directive, the following compiler error is suppressed: "property's synthesized getter follows Cocoa naming convention for returning 'owned' objects".

Category provides helper methods to managed object

`NSManagedObject` subclasses can be generated by Xcode schema editor automatically.

```
@interface Person : NSManagedObject

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSString * surname;

@end
```

Therefore, all helper methods are kept in a category in a separate file, otherwise they would be overridden by the editor.

```
@interface Person (Additions)

- (NSString *)fullName;

@end
```

Custom objects are stored as transformable attributes

A custom object can be stored by Core Data as a transformable attribute, if it conforms to `NSCoding` protocol. Xcode schema editor uses id for this attributes, which deprives developer of good type checking.

```
@interface Person : NSManagedObject

@property (nonatomic, retain) id eyeColorTransformableType;

@end
```

To ameliorate the situation, a property with the proper class type is provided by a category.

```
@interface Person (Additions)

@property (strong, nonatomic) UIColor * eyeColor;

@end
```

Consequently the original property is never used except by the shadowing property's accessors.

```
@implementation Person (Additions)

- (void)setEyeColor:(UIColor *)eyeColor
{
    self.eyeColorTransformableType = eyeColor;
}

- (UIColor *)eyeColor
{
    return self.eyeColorTransformableType;
}

@end
```

Objective-C types are stored through NSValue

`NSValue` is a simple container for Objective-C data types. It conforms to `NSCoding` protocol, thus can be stored by Core Data as transformable attribute.

```
@interface Figure : NSManagedObject

@property (nonatomic, retain) id imaginaryPositionTransformableType;        // stores NSValue object

@end
```

A category provides a shadowing property for the convenience.

```
struct ComplexNumber {
    float real;
    float imaginary;
};
typedef struct ComplexNumber ComplexNumber;

extern ComplexNumber const ComplexNumberZero;


@interface Figure (Additions)

@property (assign, nonatomic) ComplexNumber imaginaryPosition;

@end
```

The getter initializes the local variable so it does not return uninitialized value, if the shadowed property is `nil`.

```
@implementation Figure (Additions)

- (ComplexNumber)imaginaryPosition
{
    ComplexNumber imaginaryPosition = ComplexNumberZero;
    [self.imaginaryPositionTransformableType getValue:&imaginaryPosition];
    return imaginaryPosition;
}
```

The setter creates `NSValue` object with the passed value, and stores it.

```
- (void)setImaginaryPosition:(ComplexNumber)imaginaryPosition
{
    NSValue * value = [NSValue valueWithBytes:&imaginaryPosition objCType:@encode(ComplexNumber)];
    self.imaginaryPositionTransformableType = value;
}

@end
```