

# Contents

<b>Visual Appearance</b>	3
Class names are not prefixed usually	3
Protocol names indicates behavior	3
Header file follows a structure	3
Implementation file is divided by pragma marks	5
Property attributes are kept in order	5
Protocols and constants are prefixed with class name	6
IBOutlets are declared privately	6
Classes may have additional header file for subclasses	6
Methods return early, return often	7
Method are not bisected with conditionals.	8
 <b>General rules</b>	 9
Header file is imported only if necessary	9
Delegate method always passes the sender	9
Init method takes only mandatory parameters	9
Accessors are not used in init and dealloc	10
Abstract class can be faked with assertion	10
Abstract method raises exception	10
Enumeration type contains invalid value	11
The highest level of abstraction is used by default	11
Exceptions are not used to control flow	12
Lazy loading reduces memory footprint	12
Object registers itself as an observer	12
Property in category is realized by associated object	13
 <b>Concurrency</b>	 14
Multiple readers one writer	14
NSOperation can be cancelled before it begins execution	14
 <b>UIView</b>	 16
View is usually initialized in two ways	16
Interface of generic view	17
Interface of specific view	17
 <b>UIViewController</b>	 18
Properties affecting user interface	18
View controller containment	18
Container specific item is provided by property	18
Container is accessible from contained view controllers	19

# Visual Appearance

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

## Class names are not prefixed usually

Prefixes are used only for classes that 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 file follows a structure

Header should be kept as simple as possible. Declarations appears in it in the following 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

The 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 number of separating lines, 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 file is divided by pragma marks

Methods in implementation file appear in the same order they are declared in the header. They are grouped by pragma marks similar to 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
```

When greater granularity is needed:

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

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

## Property attributes are kept in order

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 should be renamed as follows.

```
@property (assign, nonatomic, getter = isVisible) BOOL visible;
```

## Protocols and constants are prefixed with class name

Protocols, notification names, enumeration types and other constants that refer to 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;

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

## IBOutlets are declared privately

Outlets are defined as weak properties at the top of class extension, divided by one line from 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 subclass access to them, they have to be declared in ForSubclassEyesOnly category. It should be placed in 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, return often

Nesting makes code harder to read.

```
- (BOOL)loginUser:(NSString *)user withPassword:(NSString *)password
{
    if (user.length >= 6) {
        if ((password.length >= 8) {
            // 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 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
}
```

## Method are not bisected with conditionals.

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 in the 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
}
```

# General rules

## Header file is imported only if necessary

Importing a header files 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 cases forward declaration should be applied.

## Delegate method always passes the sender

Delegation method passes sender as 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;
```

## Init method takes only mandatory parameters

All settings required to proper initialization are passed as initializer parameters. They are accessible later 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 class can be faked with assertion

Creating instances of abstract classes can be thwarted with following assertion.

```
NSAssert1([self isKindOfClass:[MyAbstractClass class]] == NO,
    @"%@ is an abstract class. Please do not create instances of it.", [self class]);
```

## Abstract method raises exception

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

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

## Enumeration type contains invalid value

Enumeration type equal to zero is considered as invalid. It protects instance variables from being initialized with meaningful value.

```
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.

```
NSError * error;  
BOOL valid = [self isValid:user error:&error];  
if (valid == NO) {  
    if(error.code == LoginOrderBlockedErrorCode) {  
        // assuming error is nil and LoginOrderBlockedErrorCode is equal to 0  
    }  
}
```

**Important:** Always check error domain before checking error code.

## The highest level of abstraction is used by default

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

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

Use 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, and losing the connection to the window server. The program catching such an 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 is 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 object on demand reduces initialization time of containing class.

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

## Object registers itself as an observer

Unregistering follows the same rule.

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

## Property in category is realized by associated object

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

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

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

- (UINavigationController *)navigationItem
{
    UINavigationController * navigationItem = objc_getAssociatedObject(self, navigationItemKey);
    if (navigationItem == nil) {
        navigationItem = [[UINavigationController 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

Concurrent isolation queue is used to synchronize access to a property.

```
NSString * queueLabel = [NSString stringWithFormat:@"%s.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

Main method checks if 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

By calling initWithFrame, or initWithCoder method when it is unarchived from 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. UIViews and UIViewControllers does not follow normal serialization process, their state is persisted in model.

## Interface of generic view

View exposes minimal set of properties and methods that are required for its configuration. Property is not backed up by instance variable, but by view's or subview's property, if possible.

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

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

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

```
@implementation UITableViewCell (Person)

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

## Interface of specific view

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 practise 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 item is provided by property

If the container requires the child to have specific properties, it delivers category on UIViewController that provides an item which encapsulates all of them.

```
- (UINavigationController *)navigationItem
{
    UINavigationController * navigationItem = objc_getAssociatedObject(self, kNavigationItemKey);
    if (navigationItem == nil) {
        navigationItem = [[UINavigationController 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 controller can access the container through a property.

```
@interface UIViewController (UINavigationControllerItem)

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

@end
```

Example 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;
}
```