Sailmaker System's coding guidelines

These are my coding guidelines as they currently stand. The source Scrivener document is available at https://github.com/richardbuckle/CodingGuidelines.

Points to note:

1. They are *coding* guidelines, not *layout* guidelines. If you are still obsessing about where braces should be placed, spaces vs tabs, etc, then I humbly submit that it's time to let go of all that. Professional coders can quickly adapt to any sane layout style if it's used consistently.
2. These are guidelines, not tram-lines. Sometimes it's OK to deviate from them, but the mantra that "only one who has fully understood the rules can judge when to break them" applies.
3. Above all: enjoy your work! I strongly believe that only a happy coder can produce good code. If you're not enjoying your work then that's your first priority to solve.

Richard Buckle.
Sailmaker Systems Ltd.
London, 22 August 2013.

Contents

Zero warnings and static analyser issues always

Guideline:

Code submitted for review or integration must always compile cleanly with zero warnings and zero static analyser issues. Warnings must be investigated and resolved properly, not merely suppressed.

Rationale:

A zero-tolerance policy for compiler warnings is the only way to ensure that potentially serious warnings receive the attention that they need.

Sometimes a warning will flag a more insidious problem than the warning's message may itself suggest: a full investigation is always needed.

Distracting the team with "benign" warnings is a sure way to ensure that the important warnings will be overlooked.

Permitted Exceptions:

There may be cases where it is better to suppress warnings in 3rd-party code than to attempt to fix them, however this is usually a sign that we should look for a newer version of the 3rd-party code. If you any encounter warnings in 3rd-party code, take it up with the lead developer.

A 3rd-party code library with a significant number of warnings and no published fixes is very probably no longer fit for use.

Use the standard TODO: comment

Guideline:

Use the standard `TODO:` comment, not variations such as `FIXME`, `todo` , `XXX` etc, and explain clearly what needs to be done.

Bad:

```
foo(); // FIXME clean this up
```

Preferred:

```
foo(); // TODO: a clear explanation follows here
```

Rationale:

The TODO: comment is recognized by Xcode and displayed prominently in the jump-bar's dropdown menu.

Permitted Exceptions:

None.

DLog not NSLog

Guideline:

We should nearly always prefer the DLog() macro from Marcus Zarra, Lumberjack, or the ECLogging module to NSLog(), so that we do not log unnecessarily in release builds.

Bad:

```
NSLog(@"someDict is @", someDict);
```

Preferred:

```
DLog(@"someDict is @", someDict);
```

Rationale:

Logging in release builds is very rarely of any value since we have very little access to logs on customers' devices. Additionally, logging incurs a significant speed hit in instantiating an NSCalendar (surprisingly expensive), doing printf-style formatting and in serialising its writing to disk.

Permitted Exceptions:

There are a very few cases where we would want to log in release builds. If you have one, comment it to make it clear that you're deliberately using NSLog rather than DLog.

For flat-out programmer errors, use ALog() and ZAssert() judiciously.

Minimise use of logging

Guideline:

Don't spam the log. Don't log success (what did you expect: a medal or something?). If your logging is to diagnose a particular ticket, delete it when you're done (and, ideally, when you have written a unit test for the ticket).

Rationale:

Spamming the log console with unnecessary logging is rude to your colleagues because it makes it harder for them to see their own log messages when they are diagnosing something.

Permitted Exceptions:

If you have some logging that you need periodically, guard it with a #if statement, or use Lumberjack or the ECLogging module.

No meaningful code in NSAssert

Guideline:

Do not put code that has side-effects in `NSAssert` and family, as it will be compiled away in release builds.

Bad:

```
NSAssert([self someMethod], @"someMethod failed");
```

Good:

```
BOOL success = [self someMethod];
NSAssert(success, @"someMethod failed");
```

Rationale:

Even if `[self someMethod]` looks benign today, the bad code introduces a subtle divergence in behaviour between debug and release builds that can bite us later should `[self someMethod]` be revised to have side effects.

Permitted Exceptions:

None.

Careful with NSAssert in blocks

Guideline:

Be aware that `NSAssert` references `self`. If used in a block that is retained by `self`, a retain cycle and thus a memory leak will occur.

Bad:

```
self.completionBlock = ^(BOOL success) {
     NSAssert(success, @"failed");
};
```

Good:

```
self.completionBlock = ^(BOOL success) {
     if(!success) {
          // DLog(@"failed");
          // Now handle it properly
     }
};
```

Rationale:

A fuller description and an alternative macro that doesn't reference self can be found here. However, I find it simpler not to use assertions in blocks.

Permitted Exceptions:

None.

Don't repeat yourself

Guideline:

Don't copy-and paste code. Instead refactor that code. Particularly, don't copy and paste and then change just a few things here and there. Instead, parameterise and refactor.

Rationale:

See the wikipedia article.

Repeated code is IMHO the most pernicious kind of technical debt. It is a massive maintenance risk because maintenance programmers will not be aware that if they find a bug, it will need to be fixed in multiple unrelated places of which they probably will not be aware.

There is a good probability that said maintenance programmer will be future you, who has forgotten all the state that you currently have in your head at the time that you select and copy that code.

Do the right thing by your client, future you, and any other future maintenance programmers and **DON'T REPEAT YOURSELF**.

Permitted Exceptions:

None.

Learn the "SOLID" principles

Guideline:

Learn and apply the "SOLID" design principles.

That is all.

Be judicious about singletons

Guideline:

Think very carefully before using the singleton design pattern. Prefer dependency injection.

Bad:

```
- (void)someMethod {
    FooManager *fooManager = [FooManager sharedInstance];
    // do something with fooManager ...
```

Good:

```
- (void)someMethodWithFooManager:(FooManager *)fooManager {
// do something with fooManager ...
```

Rationale:

The singleton design pattern is often more of an anti-pattern. Singletons are just globals in disguise. They introduce hidden coupling between modules, make the writing of test cases and mock objects significantly harder, and generally make the code base more brittle and harder to maintain.

Permitted Exceptions:

Only as agreed with the lead developer. And you'd better make a bloody good case.

Minimise header dependencies

Guideline:

Strive to minimise header dependencies. Use forward declarations where possible.

Bad:

```
#import "Foo.h";

@interface Qux : NSObject {
Foo *foo;
// …
```

Good:

```
@class Foo;

@interface Qux : NSObject {
Foo *foo;
// …
```

Rationale:

Unnecessary header dependencies not only increase compile time unnecessarily, they increase code coupling because you are forcing these unnecessary dependencies on all users of your class.

As a rule, you only need to import headers in your implementation file. Your interface file need have only forward declarations using `@class` and `@protocol`.

Tips:

AppCode has a command for optimizing `#import` statements.
objc_dep is a useful tool for graphing them.

Permitted Exceptions:

None.

Use string resources

Guideline:

All strings visible in the UI should be in string resources, not string literals in code. Use informal structured naming for the key, not the default English text.

Bad:

```
label.text = @"Congratulations!";
```

Still not ideal:

```
label.text = NSLocalizedString(@"Congratulations!", nil);
```

Best:

```
label.text =
NSLocalizedString(@"labels.text.congratulations", nil);
```

Rationale:

It is a horrible job to have to go through the entire code base looking for string literals that should be in resources. Always write `NSLocalizedString()` from the get-go.

Structured naming for the key makes any omissions stand out, and focuses translators and proof-readers on the text that needs attention.

Permitted Exceptions:

Absolutely none whatsoever. But do remember that this applies only to UI-visible strings, not strings used purely internally as plist keys etc.

No magic numbers

Guideline:

Use named constants instead of magic numbers.

Bad:

```
myView.size.width = 1024 - 42 - (4 + 4);
```

Better:

```
CGSize containerSize = containerView.frame.size;
myView.size.width = containerSize.width
      - kReservedWidth - (kLeftPadding + kRightPadding);
```

Best:

Use AutoLayout.

Rationale:

One of the ultimate maintenance programming headaches. Six months later, no-one will have the faintest idea what the magic numbers mean.

Permitted Exceptions:

Dividing by 2 to get a midpoint, or better multiplying by 0.5f, provided the intent is clear from the variable naming.

Performance considerations:

In the case of view geometry, you may want to check for misaligned views with Instruments and use the stuff in CGGeometry.h to get things aligned on integer values.

Always get the screen size at runtime

Guideline:

Always get the screen size at runtime. Never hardwire it.

Bad:

```
myView.size.width = 1024 - 42 - (4 + 4);
```

Dubious:

```
CGSize screenSize = [[UIScreen mainScreen] bounds].size;
// who said we're on the main screen?
myView.size.width = screenSize.width - kReservedWidth -
(kLeftPadding + kRightPadding);
```

Best:

Use AutoLayout.

Rationale:

Hard-coding a fixed screen size hurts portability between iPhone and iPad, and doesn't cater for new hardware and Apple TV.

Permitted Exceptions:

None.

Never compare against YES etc

Guideline:

Never compare boolean expressions against `YES`, `true`, `1`, etc.

Bad:

```
if(foo == YES) {
     // ...
}
```

Good:

```
if(foo != NO) {
     // ...
}
```

Best:

```
if(foo) {
     // ...
}
```

Rationale:

All non-zero values represent truth. Comparison against `YES`, `true`, etc will give incorrect results when the value being compared is any other non-zero value.

Permitted Exceptions:

None. Such code is flat wrong.

Minimise preprocessor usage

Guideline:

Minimise preprocessor usage. Prefer `const` and `inline` to `#define`.

Bad:

In the header:

```
#define kLookupKey @"blah"
```

Good:

In the header:

```
extern NSString *const kLookupKey;
```

In the implementation:

```
NSString *const kLookupKey = @"blah";
```

Rationale:

The C preprocessor is arguably the purest manifestation of programming evil imaginable. It smashes through all scoping rules and encapsulation defences. Worst of all, there is little or no defence against a preprocessor macro in a header that you are forced to include.

Putting a preprocessor macro in a header that clients of your code need to `#import` is one of the rudest things you can do. We tolerate it from the platform vendors, but it is not an example that should be followed.

Solution:

Use `extern const` definitions in the header as shown above.

Also, by defining the value only in the implementation file, we have eliminated a dependency. Should we need to change `kLookupKey`, only the implementation file need be recompiled.

For functions, define inline functions rather than macros.

Permitted Exceptions:

Macros in header files are banned, no exceptions.

Judicious use of #define WANT_FOO_LOGGING in implementation files to control logging via #if WANT_FOO_LOGGING blocks is acceptable if it doesn't get out of hand.

No Hungarian naming

Guideline:

Do not use hungarian naming to encode the type of a variable or whether it is an objective-C property or instance variable in its name.

Rationale:

Quite apart from the unnecessary cognitive burden it adds to reading code, and the bad fit with a dynamically-typed language, to quote "How to Write Unmaintanable Code":

> Hungarian Notation is the tactical nuclear weapon of source code obfuscation techniques... Due to the sheer volume of source code contaminated by this idiom nothing can kill a maintenance engineer faster than a well planned Hungarian Notation attack.

Permitted Exceptions:

The usual Objective-C convention of prefixing reusable classes, and class extension methods, with a unique prefix.

No uninitialised locals, ever

Guideline:

Local variables should always be initialised to a safe value at the point of declaration, and should not be declared until they are needed.

Bad:

```
CGRect rect;
switch(something) {
     // some code that doesn't cover all values
}
```

Good:

```
CGRect rect = CGRectNull;
switch(something) {
     // some code that doesn't cover all values
}
```

Rationale:

If not initialised, local variables get garbage values. This is never good.

Permitted Exceptions:

Under ARC, Cocoa objects are always initialized to nil, so they get a free pass. Objective-C properties and instance variables are also guaranteed to be zeroed out.

Dot notation is for genuine properties only

Guideline:

Dot notation is only for getting and setting properties. Do not abuse it to call methods that are not property getters or setters.

Bad:

```
UIView *view = …;
view.sizeToFit; // WRONG, not a property
```

Good:

```
UIView *view = …;
[view sizeToFit];
CGRect frame = view.frame;
```

Rationale:

Abusing dot notation to call non-property methods is not supported and may well break in future. Regardless, it is highly misleading to maintenance programmers.

Permitted Exceptions:

None.

Commentary:

Yes, Apple engineers have been seen breaking this rule in WWDC vids. No, that does not mean that it'll get past my code review.

Avoid mutable container properties

Guideline:

Avoid mutable container properties.

Bad:

```
@interface Foo {
@property (readwrite, retain) NSMutableArray *someArray;
```

Good:

```
@interface Foo {
@property (readwrite, retain) NSArray *someArray;
```

Rationale:

Mutable container properties allow the contents of the property to be modified without the owning object's knowledge. This severely violates encapsulation, and breaks fundamental Cocoa technologies such as key-value-observing.

Because Cocoa container classes are pointer-based, the overhead of sticking to immutable container properties is small. Indeed, it was one of Cocoa's design goals. Additionally, immutable containers implement many optimisations that mutable containers cannot.

Permitted Exceptions:

Properties that are declared only in the class extension in the .m file. But that should be viewed as a last resort: KVO and KVC can still cause problems.

Beware of retain cycles with blocks and self

Guideline:

If self retains a block that references self, a retain cycle and a memory leak will occur.

Bad:

```
self.completionBlock = ^() {
     [self someMethod];
};
```

Good:

Without ARC…

```
__block id weakSelf = self;
self.completionBlock = ^() {
     [weakSelf someMethod];
};
```

With ARC…

```
__weak id weakSelf = self;
self.completionBlock = ^() {
     [weakSelf someMethod];
};
```

Rationale:

Obviously the retain cycle is bad news and needs to be avoided.

Permitted Exceptions:

If the block is not retained by self, these gymnastics are unnecessary.

Follow the standard for Git commit messages

Guideline:

To fit well with the various tools, Git commit messages should have an initial, brief, summary line followed by any necessary detail on subsequent lines.

Bad:

[a ton of text all on one line]

Good:

Fix ticket #345
The issue turned out to be […].
The fix was to […].
Unit test […] has been added to guard against regressions.

Rationale:

All the various tools for viewing Git logs work best if the initial summary line is relatively short, preferably less than 80 characters and certainly no more than 120.

Do feel free to take as much space as you need on subsequent lines for detailed explanations.

Permitted Exceptions:

None.

Cite ticket numbers in commit messages

Guideline:

If your commit is related to any ticket numbers (for bugs or work units), cite those numbers in your Git commit message.

Rationale:

This permits two-way linking between commits (aka changesets) and tickets. It is particularly useful for maintenance programmers to use `git blame` to find the changeset for a given line of code and thus tell what ticket motivated it.

Permitted Exceptions:

Sometimes, especially during bring-up, commits will be general work in progress without a ticket number. In these cases, write the fullest commit message you can.

Use submodules for 3rd-party code

Guideline:

3rd-party code should be maintained in a Git repo of its own and a defined tag should be pulled into the working project as a Git submodule.

Rationale:

This allows us to track revisions of the 3rd-party code, integrate any fixes that we've made in any branches of our own, all without disturbing production code. Once integration testing is complete, we can then point the production code to a new tag on the submodule.

Permitted Exceptions:

Legacy projects that are no longer in development.

Don't commit commented-out code

Guideline:

Keep a clean code base without any dead wood. Before you commit, delete any commented-out code.

Rationale:

We have version control to get back the old code if we need it. Commented-out code wastes everyone's time by forcing every maintenance programmer who encounters it to try to figure out why it was commented out and whether it needs to be kept.

Permitted Exceptions:

If you have code that is needed periodically e.g. for testing or benchmarking, disable it with `#if` and add a comment explaining what it is for and why it needs to be kept.

Don't sign your comments

Guideline:

The old school convention of signing your comments with your initials and a timestamp are redundant in the age of version control. Write good version control comments instead.

Rationale:

Don't add redundant cognitive burden to the codebase. If I need to see who wrote it, when and why, I'll look in version control.

Permitted Exceptions:

None.

Document History

Initial GitHub upload, 2012-08-23.