David Chisnall

ESSENTIAL CODE AND COMMANDS

Objective-C 2.0

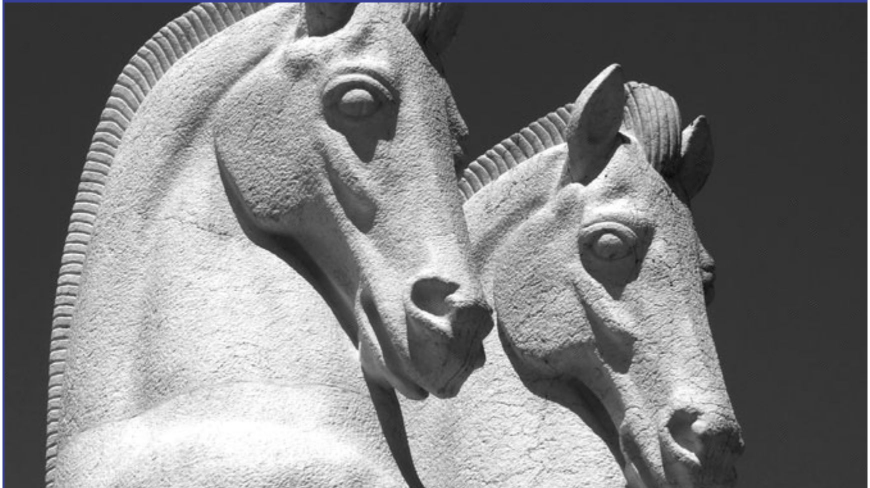
PHRASEBOOK

David Chisnall

ESSENTIAL CODE AND COMMANDS

Objective-C

PHRASEBOOK



the class by other threads.[1]

Note: The Apple documentation recommends using **`@synchronized`**`(`**`self`**`)` in the `+sharedInstance` method for creating singletons. That approach is both significantly slower and requires more code than the approach proposed in this section, so it has little to recommend it.

You don't need to override `+allocWithZone:` in a singleton, but it provides a little bit extra runtime checking if you do. The sample code will throw an exception if you try to allocate a new instance of the class after the singleton instance has been created.

Note the tests in both the `+allocWithZone:` and `+initialize` methods. These are needed for subclassing. You don't want to throw an exception if someone creates a subclass of your singleton object. Subclassing singletons is quite tricky, so you may skip this extra test if you don't want to support it.

Delegation

Objective-C doesn't support multiple inheritance. This is not a huge limitation, because it's very hard to use multiple inheritance

[1] There is a long-standing bug in the GCC runtime that prevents this from working correctly, but it's fixed in the GNUstep runtime.

well, but with some problems it seems like the natural solution.

The idea behind the *delegation pattern* is similar to that behind inheritance and prototypes: You allow one object to define some subset of the behavior of another object.

You won't see this in the Foundation framework much, but it's very common in AppKit or UIKit code. Each user interface object uses delegation to allow you to define what happens in response to user interface events.

Several patterns are related to delegation, and they are all related to the general rule that you should favor object composition over inheritance. This is important for loose coupling, because it makes it much easier to reuse the code.

If you have a C++ class with three superclasses providing some aspects of its behavior, then it is a lot harder to modify than an Objective-C class delegating aspects of its functionality to three other classes.

Providing Façades

```
@interface Control : View
{
  id cell;
}
@end
@implementation Control
- (id)selectedCell
{
  return cell;
}
- (BOOL)isEnabled
{
    return [[self selectedCell] isEnabled];
}
- (void)setEnabled: (BOOL)flag
{
  [[self selectedCell] setEnabled: flag];
  [self setNeedsDisplay: YES];
}
@end
```

From: facade.m

One very common use for delegation is the *façade pattern*. This wraps one or more private or semi-private objects in a public interface. This is very commonly used in Objective-C to provide something like multiple inheritance, where an object combines behavior from several distinct objects, delegating some of its functionality to each one.

The `NSControl` hierarchy in AppKit is a good example of this. Classes in this hierarchy inherit behavior from `NSView` and delegate behavior to an `NSCell` subclass. `NSView` provides features such as a graphics context, event handling, and interaction with the view hierarchy. The cell provides features such as drawing. The example at the top of this section is a (very) simplified version of `NSControl`.

When you click on a button in OS X, you are usually clicking on an instance of `NSButton`, which is an `NSControl` subclass using an `NSButtonCell` for its implementation. As a programmer, you can use controls almost interchangeably and you can also reuse the cells that they contain. If you see a button in a table or outline view, for example, this is drawn with an `NSButtonCell`, not with an `NSButton`.

In Chapter 19, we'll look at using the Objective-C forwarding mechanisms to quickly and easily implement this kind of façade. It's also possible—and common—to implement them simply by calling the methods in the wrapped object or objects directly.

This approach is much more flexible than multiple inheritance, because it allows the same class to be used with lots of different delegates. In C++, you would use template classes to achieve the same effect. These have different advantages. The Objective-C version generates a lot less code, which translates to better instruction cache usage. The C++ version makes certain categories of optimization (primarily inlining) easier at compile time.

Creating Class Clusters

```
static Pair *placeHolder;
+ (void)initialize
{
  if ([Pair class] == self)
  {
    placeHolder = [self alloc];
  }
}
+ (id)allocWithZone: (NSZone*)aZone
{
  if ([Pair class] == self)
  {
    if (nil == placeHolder)
    {
      placeHolder =
        [super allocWithZone: aZone];
    }
    return placeHolder;
  }
  return [super allocWithZone: aZone];
}
- (Pair*)initWithFloat: (float)a float: (float)b
{
  return [[FloatPair alloc] initWithFloat: a
       float: b];
}
- (Pair*)initWithInt: (int)a int: (int)b
{
  return [[IntPair alloc] initWithInt: a int: b];
}
```

From: classCluster.m

Class clusters are very common in Objective-C. A lot of the Foundation classes are class clusters, and you may find it useful to implement some more of your own.

A class cluster is an *abstract superclass* that hides concrete subclasses. This is easier in Objective-C than many other languages, because there is no keyword for object creation. When you send a message to a class asking for a new instance, the class may return an instance of itself, but it may also return an instance of some subclass.

If you create an `NSArray` using `+arrayWithObjects:`, you may get a subclass that wraps a simple C array. If you create one using `+arrayWithArray:`, you are likely to get one that just references the other array.

Another good example is `NSNumber`. This wraps a single C primitive value. You could implement this with two instance variables: a union of all of the possible value types and another saying which one it is. This would waste a lot of space and be quite complicated. A simpler solution is to implement a different subclass for each type that you support.

Most class clusters provide named constructors, such as `+numberWithFloat:`, that create a new instance of the correct subclass. If you create them with `+alloc`, you typically get a placeholder class returned and then get the real object in response to the initialize message.

The example is a `Pair` class. This does not declare any instance variables and is never used directly. If you send a `+alloc` message to this class, you get a *singleton* instance. When you