APPENDIX D

Advanced Topics

Testing String Mutability with Exceptions

We can test a string's mutability as follows. Again don't use this for production code.

```
BOOL isMutableString(NSString* string)
{
  @try
  { [((id) string) setString:string];
    return YES; }

  @catch(id e) {}

  return NO;
}
```

(Casting `string` to `id` removes a static type check warning.)

Uncaught Exceptions

Any exceptions you do not catch invoke the default exception handler. You can set your own default exception handler with `NSSetUncaughtExceptionHandler` which takes as argument a pointer to a function handler. (Recall that in C, the `&` operator returns the address of its argument. Thus `&myHandler` is a pointer to the `myHandler` function). You can obtain the system's handler with `NSGetUncaughtExceptionHandler`.

For instance, add this code to `main.m`:

```
NSUncaughtExceptionHandler* globalHandler = NULL;

void myHandler(NSException *exception)
{
  NSLog(@"It's the end of the world as we know it! %@", exception);

  if (globalHandler)
    globalHandler(exception);
}

int main(int argc, char *argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  globalHandler = NSGetUncaughtExceptionHandler();
  NSSetUncaughtExceptionHandler(&myHandler);

  int retVal = UIApplicationMain(argc, argv, nil, nil);
  [pool release];
  return retVal;
}
```

Then change `pressVar:` to the following:

```
- (void) pressVar:(UIButton*)sender;
{
  @throw [NSException exceptionWithName:@"test"
                                 reason:@"testing"
                               userInfo:nil];
};
```

When you press the var button, you'll see

```
  It's the end of the world as we know it! testing
  *** Terminating app due to uncaught exception 'test', reason: 'testing'
```

The Implementation of Exceptions

This section is somewhat advanced. Unless you are planning to mix Objective-C and C++, you can skip it. Exceptions are implemented with C's `setjmp` library. `setjmp` does not invoke any of the destructors that C++ would invoke for stack-based C++ objects when unwinding the stack. This means that all of the destructors that would have been invoked by functions between the `@catch` block and the `@throw` statement are ignored. The implementation of exceptions is straightforward. `NSExceptionFrames` record the location of each `@try` block.

```
typedef struct NSExceptionFrame
{
  jmp_buf                  state;
  struct NSExceptionFrame *parent;
  NSException             *exception;
} NSExceptionFrame;

static NSExceptionFrame* _currentExceptionFrame = nil;
```

`@throw` is equivalent to the following:

```
void _throw(NSException* e)
{
  _currentExceptionFrame->exception = e;
  longjmp(_currentExceptionFrame->state, 1);
}
```

Ignoring the exception arguments to `@catch`, the following code:

```
@try   { try_block   }
@catch { catch_block }
```

is equivalent to this code:

```
{
  NSExceptionFrame localFrame;
  _pushException(&localFrame);

  // try part

  if (setjmp(localFrame.state)==0)
  {
    try_block
    _popException(&localFrame);
  }

  // catch part

  else
  {
    NSException* exception = e->exception;
    _popException(&localFrame);
    catch_block
  }
}
```

where

```
void _pushException(NSExceptionFrame* e)
{
  e->exception           = nil;
  e->parent              = _currentExceptionFrame;
  _currentExceptionFrame = e;
}

void _popException(NSExceptionFrame* e)
{
  e->parent->exception   = e->exception;
  _currentExceptionFrame = e->parent;
}
```

NSZones

NSZones are a solution to **memory fragmentation**. Memory fragmentation is the phenomenon in which free memory becomes divided into many small pieces over time. It happens because each request for memory requires finding enough contiguous memory to satisfy the request. A little additional memory is usually left over. Over time, you might end up with enough free memory to satisfy your application's needs, but because it is not contiguous, it cannot be used. Each NSZone has its own private memory heap, free list, and pool of memory pages. For instance, if you will be creating many objects of the same size, you can create a zone and allocate them to that zone with allocWithZone:. Because they all have the same size, they will not fragment their zone. Objects are usually assigned from the default zone (returned by NSDefaultMallocZone()).

Although this is true in theory, in practice, allocWithZone: allocates memory in zones on the simulator but not on the iPhone! This happens because allocWithZone: calls class_createInstance or class_createInstanceFromZone to set up basic information about the object (its isa pointer). The 2.0 Objective-C runtime shipped on the iPhone only supports class_createInstance, which can only create instances of Objective-C objects in the default zone. The simulator runs with the 1.0 Objective-C runtime, which supports the class_createInstanceFromZone method. It seems that Apple is moving away from the concepts of memory zones, but you are still expected to implement copyWithZone: and mutableCopyWithZone: rather than copy and mutableCopy.

Creating Singletons by Overriding allocWithZone:

You can override allocWithZone: to create singletons (classes that only have one instance). Singletons should be used with care as they bind tightly all their clients in the same way global variables do.

Singletons should never be released:

```
- (id)       copyWithZone:(NSZone*) zone { return self; }
- (id)       retain                      { return self; }
- (unsigned) retainCount                 { return UINT_MAX; }
- (void)     release                     {}
- (id)       autorelease                 { return self; }
```

Initializing a singleton more than once will destroy the previous content of the singleton. Apple's recommended solution is to create a factory method (a class method that returns objects) that returns the singleton. Clients should invoke the factory method, and allocWithZone: only returns memory on its first invocation. Later invocations return nil so that init does not destroy the previous content of the singleton.

```
static MySingleton *mySharedSingleton = nil;

+ (MySingleton*) singleton
{
  @synchronized(self)
  {
    if (mySharedSingleton == nil)
      [[self alloc] init]; // assignment not done here
  }

  return mySharedSingleton;
}

+ (id) allocWithZone:(NSZone*) zone
{
  @synchronized(self)
  {
    if (mySharedSingleton == nil)
    {
      mySharedSingleton = [super allocWithZone:zone];
      // assignment and return on first allocation
      return mySharedSingleton;
    }
  }

  return nil; // on subsequent allocation attempts return nil
}
```

The advantage of this solution is that `init` can be defined as usual and derived classes will still work. The disadvantage of this solution is that the NIB file reader does not know about the `singleton` factory method, and might attempt to create the singleton more than once using the standard `alloc` and `init` calls, which will return `nil` on later invocations.

The `@synchronized` keyword creates a lock on the object, to ensure two threads cannot enter its section at the same time.

Extending Objects by Overriding `allocWithZone:`

Another reason to override `allocWithZone:` is to reserve some space at the end of the object, for instance to store a variable-length string:

```
+ allocWithZone:(NSZone*)zone
{
  ...;
  return NSAllocateObject(self, extra_bytes, zone);
}
```

You can obtain the size of an Objective-C object *obj* using `sizeof(obj)`. The standard typed pointer rules apply: `self+1` would point past the end of the object. Because these sizes are static, methods that may be inherited should instead use `class_getInstanceSize` on the class.

`allocWithZone:` can return a static placeholder object and defer allocation to the `init` method for variable-sized objects. For instance, allocating memory for an `NSString` requires knowledge of the length of the string, but the string is given to `init`, not `alloc`. `NSString`'s `alloc` method returns a static `NSPlaceholderString` and defers memory allocation to the `init` method:

```
NSLog(@"%@", [[NSString alloc] className]);
  NSPlaceholderString
```

Did you Know?

If you see undocumented classes containing the word `Placeholder` in the debugger, it is an object of a not-yet initialized class.

How Applications Start

So far, we've glossed over how applications start. In Hour 3, "Simplifying Your Code," you learned that just like in C, applications start at the `main` function. Then in Hour 6, "Understanding How the User Interface Is Built," we mentioned that `UIApplicationMain` is somehow responsible for creating the application object and delegate and starting up Cocoa. Now that we know about run loops, we can be more precise.

`UIApplicationMain` first initializes the Cocoa libraries. Then, if its third argument does not specify the class of the application object, it looks in the application's `Info.plist` file for it under the `NSPrincipalClass` key. In either case, if a class is specified, it checks that the object is a `UIApplication` or its descendent. Otherwise, it simply builds a `UIApplication`. Similarly, if the Application Delegate is specified, it is built and set up.

The second step in `UIApplication`'s `_run` method is to set up the Mach ports the application will listen to. If it cannot connect to the Mach ports of the system event server, it terminates. Otherwise, it sets up a run loop and registers for various events it needs to be notified about, such as memory scarcity. It also informs the iPhone OS of its application bundle ID. To get called from the run loop, it sets up a request for `_runWithURL:` to be run after a few milliseconds delay using `performSelector:withObject:afterDelay`. It sets up the autorelease pools to be used within the run loop, and the observer to drain them. Then, it starts the run loop. The application closes when the run loop returns: `_run` is exited normally, which returns to `UIApplicationMain` and then to `main`.

`_runWithURL:` registers for more system events (for example, change of user defaults, language, locale, time, the display turning on and off). It sets up the basic display state and then loads the NIB file specified in `Info.plist`. The NIB file can set the Application Delegate. It checks whether the delegate responds to `applicationDidFinishLaunching:` and runs it; otherwise, it checks `UIApplication` for `applicationDidFinishLaunching:` and runs that. Your application then takes over. On return, it broadcasts a general `UIApplicationDidFinishLaunchingNotification`.

Watch Out!

Changing `File Owner`'s class in Interface Builder does not change the Application object that is created. To change the Application object, you must change the third argument of `UIApplicationMain`.

Threads

An alternative to inserting calls to `CFRunLoopRunInMode` throughout your code is to place the rendering code in another thread. Threading is an enormous topic, so this section focuses mainly on the Objective-C specific aspects of threading.

Threading introduces nondeterminism into your application: The state of a non-threaded application always changes in the same way given the same inputs; the state transitions can be written down in a line as shown in Figure D.1. The program must be correct for just that one sequence. The state of a threaded application changes in as many ways as there are possible interleavings of threads, as shown in Figures D.2 and D.3. The state transitions can only be written down as a directed acyclic connected graph, an arc representing a transition, as shown in Figure D.3. The program must be correct for any path taken through this graph.

Did you Know?

Graphs are commonly used in computer science. A graph consists of nodes and arcs connecting the nodes. If every arc has a direction (a defined start node and end node), the graph is called directed. If every pair of nodes is connected via a path of arcs, the graph is connected. Acyclic graphs have no cycles: They have no paths formed by following arcs in a given direction which lead back to a previously encountered node. In this example, the nodes represent state, and the arcs state transitions.

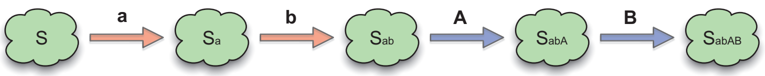


FIGURE D.1
The programmer and computer agree on what's happening

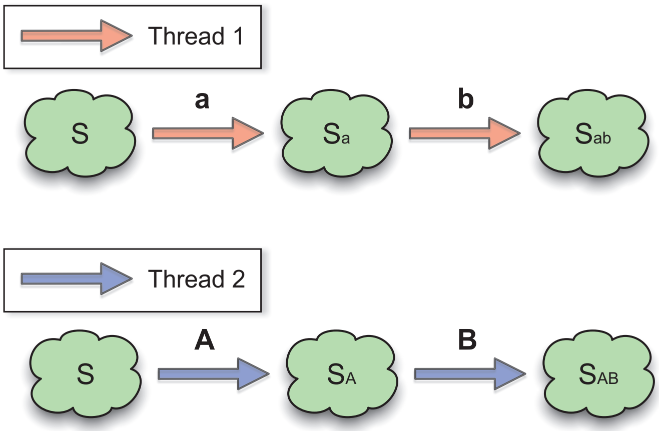


FIGURE D.2
What the programmer thinks is happening

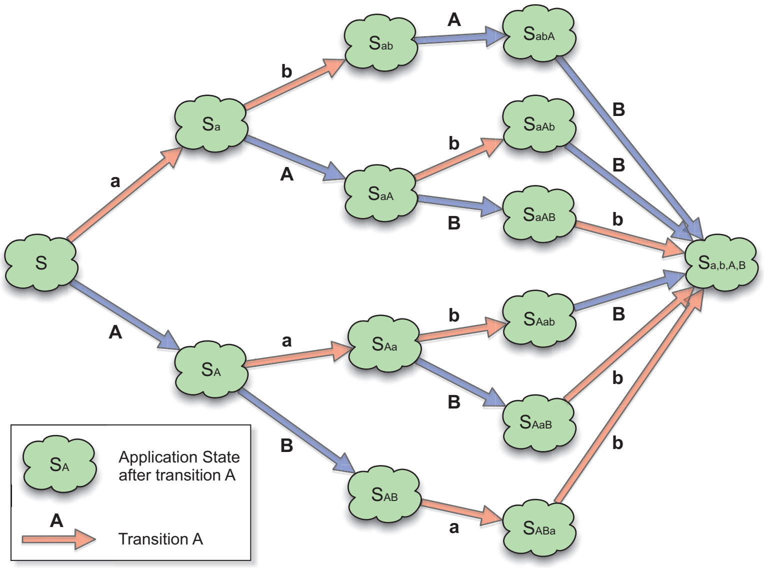
Well-designed threaded programs keep threads as independent as possible: This significantly reduces the number of thread interleavings that differ, represented as the graph's branching. For instance, off-loading work to a worker thread that only communicates back to the main thread when it is done only creates a second branch.

Apple provides standard `pthread`-based threading, `NSThreads`, and `NSOperationQueue`. It also provides a number of synchronization methods.

pthreads

`pthreads` work the same way as they do on other platforms. However, if you want to use Cocoa classes in a `pthread`, you must put Cocoa into multithreaded mode: Cocoa avoids creating locks for single-threaded applications because using mutexes is expensive. (Mutexes provide inter-thread synchronization.) Instead, Cocoa sets all locks to `nil` at initialization and registers for the `NSWillBecomeMultithreadedNotification` to create the locks as soon as the application becomes multithreaded. The `nil` rule means that all lock manipulations are ignored in single-threaded applications.

FIGURE D.3
The computer can take any path through the graph



To put Cocoa in multithreaded mode, your application must create an `NSWillBecomeMultithreadedNotification`. As `NSThread` creates this notification when it starts its second thread, all you need to do is create an `NSThread` that immediately exits before you use `pthreads`. You can check whether Cocoa is in multithreaded mode at any time by invoking NSThread's isMultiThreaded. Notifications are discussed in more detail in Hour 12, "Adding Navigation and Tab Bar Controllers."

NSThreads

NSThread is Cocoa's thread API. Threads should only be used for tasks that share very few (or no) data structures with other tasks so as to reduce the complexity of your state-transition graph. Because creating threads is expensive, the task should perform a large amount of work (more than 10 ms).

Threads are created using NSThread's initWithTarget:selector:object: and launched using start. The selector and target specify the thread's main method, and you can specify an argument object.

```
NSThread* thread = [[NSThread alloc] initWithTarget:obj
                                           selector:@selector(mainMethod:)
                                             object:nil];
// optionally configure the thread's name,
// priority, stack size and thread dictionary
[thread setName:@"BlurImage"];
[thread start];
```

NSThread has a shorthand class method detachNewThreadSelector:toTarget:withObject: to create and launch a thread in a single step. Another alternative is to subclass NSThread and override its main method.

The method specified by the selector will be run in the new thread. It must create and maintain its own NSAutoreleasePool to prevent leaks. It must also catch any exceptions it generates, to prevent the application from being terminated. If your thread only needs to perform computations and not communicate with the world, you don't need to use a run loop.

```
- (void) mainMethod:(id)obj
{
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init]

  @try
  { /* do some work */ }

  @catch (NSException* exception)
  { NSLog(@"Caught an exception %@ in thread %@",
             exception, [NSThread currentThread]); }

  @finally
  { [pool release]; }
}
```

Leaving a thread using NSThread's exit method does not release the autorelease pool or any other resources you might have claimed. Prefer to exit the thread by returning from mainMethod.

If your thread does communicate to the outside world, or you want to use the performSelector:onThread:withObject:waitUntilDone: methods, it needs to use a run loop. NSThread creates a run loop for each thread it starts, but you must run it:

```
- (void) mainMethod:(id)obj
{
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

  @try
  {
    NSRunLoop* runLoop = [NSRunLoop currentRunLoop];

    [self installCustomInputSourceOrTimer];

    leave  = NO;
    while (!leave)
    {
      [runLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:0.05]];

      // You can do some work here which could change the leave variable.
      // Or you could decide to change leave in an action invoked by the run
loop.
    }
  }

  @catch (NSException* exception)
  { NSLog(@"Caught an exception %@ in thread %@",
          exception, [NSThread currentThread]); }
  @finally
  { [pool release]; }
}
```

The performSelectorOnMainThread:withObject:waitUntilDone: and performSelector:onThread:withObject:waitUntilDone: methods inherited from NSObject enable threads with run loops to communicate with one another as shown in Figure D.4. The messages sent in this manner are asynchronous and require no locks. They're great if you don't need speed.

New thread run loops have no input sources or pending timers. Run loops without input sources or pending timers are exited immediately as they have nothing to wait for. The while loop will keep the thread running, but unless you are doing work outside the run loop, you'll be wasting battery charge. If you're planning to make a thread you can send tasks to using performSelector:onThread:withObject:waitUntilDone: and which will use Cocoa-created input sources, you can keep the run loop alive by using a timer that never triggers:

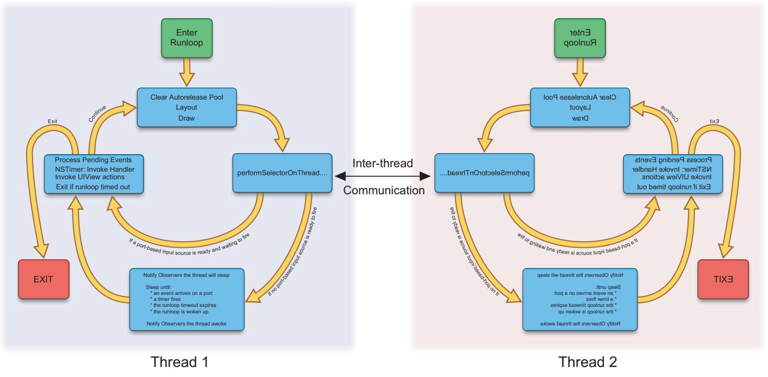

Enter Runloop
Clear Autorelease Pool
Layout
Draw
Continue
Exit
Process Pending Events
NSTimer: Invoke Handler
Invoke UIView actions
Exit if runloop timed out
performSelectorOnThread....
Inter-thread
Communication
If a port-based input source is ready and waiting to fire
If no port-based input source is ready to fire
EXIT
Notify Observers the thread will sleep
Sleep until:
* an event arrives on a port
* a timer fires
* the runloop timeout expires
* the runloop is woken up
Notify Observers the thread awoke
Thread 1
Thread 2

FIGURE D.4
Two threads with run loops can communicate.

```
- (void) installCustomInputSourceOrTimer
{
  NSTimer* timer = [[NSTimer alloc] initWithFireDate:[NSDate distantFuture]
                                            interval:0.0
                                              target:self
                                            selector:@selector(never)
                                            userInfo:nil
                                             repeats:NO];
  [[NSRunLoop currentRunLoop] addTimer:timer forMode:NSDefaultRunLoopMode];
}
```

See Apple's "Threading Programming Guide" for details on how to make your own run loop input source.

NSOperationQueue

NSOperationQueue lets you run a series of tasks in the background. It is appropriate for smaller tasks as it avoids creating a new thread for each operation. Instead, it creates a few threads, which each serially takes operations from the queue and executes them. The number of threads created is system dependent and might differ between the simulator and the iPhone.

Every operation you define should be a child of NSOperation. You must define its main method, which will be invoked to perform the operation. Be aware that once main exits, the operation is considered finished and the object is released. If your main function uses a class that has an asynchronous API such as NSURLConnection (described in Hour 14, "Accessing the Network"), you will need to invoke the run loop until the appropriate delegates have been called.

A nice feature of NSOperations is that you can require them to be called after other NSOperations have completed by adding the other operations as dependencies. NSOperationQueue will run them in the correct order, unless you've created a circular dependency, in which case it might deadlock.

Alternatively, you can use NSInvocationOperation to invoke a method on an object:

```
- (id) init
{
  ...
    queue = [[NSOperationQueue alloc] init];
  ...
}


- (void) runImageProcessing:(UIImage*)image
{
  NSInvocationOperation* operation;
  operation = [[NSInvocationOperation alloc] initWithTarget:self
                                                     selector:@selector(blur:)
                                                       object:image];
  [queue addOperation:operation];
  [operation release];
};
```

Recall that blur: will run in another thread, and should not change self without appropriate locks.

NSOperationQueue also can be given concurrent operations that are supposed to create their own thread when their start method is invoked. It is up to you to write that start method and the associated isExecuting and isFinished methods. http://www.dribin.org/dave/blog/archives/2009/05/05/concurrent_operations/ provides an interesting example of doing this.

Synchronization

The iPhone OS provides a number of mechanisms to guarantee safe data access. The simplest primitive is the BSD declared OSAtomicCompareAndSwapLongBarrier and the other OSAtomic functions declared in <libkern/OSAtomic.h>. For instance, to increment a variable that can be shared, write the following:

```
int32_t inc(volatile int32_t* ptr)
{
  int new = 0;
  do    { int old = *ptr; new = old + 1; }
  while ( !OSAtomicCompareAndSwapLongBarrier(old, new, ptr) );
  return new;
}
```

CPUs and compilers are free to reorder and postpone reads and writes for efficiency. For instance, you would expect a to always be larger or equal to b in the following code:

```
static volatile int a = 0; // static versus non-static because
volatile int        b = 0; // we want a & b on different cache lines
while (1) { ++a; ++b; };
```

However, without memory barriers, a thread on a different CPU could see a being smaller than b. The simplest form of memory barrier OSMemoryBarrier() guarantees that all pending loads and stores issued before the memory barrier instruction complete before the memory barrier instruction itself completes. Variants only force loads or stores to complete. Memory barriers are important when dealing with threads on CPUs that use a weakly ordered memory model such as the ARM processor in multicore systems.

A good discussion of memory barriers is available at
http://ridiculousfish.com/blog/archives/2007/02/17/barrier/.

Did you Know?

OSAtomicCompareAndSwapLongBarrier and friends are described in a manual page on your system that can be invoked from the Terminal application by typing

man OSAtomicCompareAndSwapLongBarrier.

Locks provide a second method of synchronization. pthread provides its standard mutex and read-write locks. NSLock provides a Cocoa API for nonrecursive pthread mutex locks, whereas NSRecusiveLock provides an API for recursive pthread mutex locks. Mutex stands for Mutual Exclusion. You can give your locks a name, which will be printed in error messages. For instance, unlocking a lock that is not locked will print just such an error message. You must unlock NSLock objects from the same thread as locked it.

For convenience, Objective-C provides the @synchronized directive, which prevents any two threads from entering any @synchronized bracketed code if they share the same @synchronized argument. Think of it as trying to breathe in and out and swallow at the same time—it isn't possible.

```
- (void) swallow:(float)food
{
  @synchronized(self)    // self is @synchronized's argument
  { energy += food; }    // This is @synchronized's bracketed code
}

- (void) breatheIn:(float)air
{
  @synchronized(self)
  { oxygen += air * 20.95; }
}
```

The argument can be any Objective-C object. Class methods can use the class object to guarantee no two class methods change its static information simultaneously.

Similarly, you can use unique static objects to prevent multiple threads from calling the same method simultaneously:

```
static id semaphore = nil;

 + (void) initialize
 {
   if (semaphore == nil)
     semaphore = [[NSObject alloc] init];
 };

 - (void) criticalMethod
 {
   @synchronized(semaphore)
   { /* critical code */ }
 }
```

`@synchronized` involves four steps:

1. It locks a global lock to access an object-mutex map. If the object is not in the map, it creates a mutex for it and adds them to the map; otherwise, it increments the object's reference count. Then, it unlocks the global lock.
2. It creates a `@try`/`@catch` block around the critical code so that it can catch any exception within the code and release the object's mutex.
3. It locks the object's mutex, performs the critical code, and unlocks the object's mutex.
4. It locks the global lock and decrements the object's reference count. If zero, it removes the object from the object-mutex map. Then, it unlocks the global lock.

By the Way

`@synchronized` is slower than `pthread`-based mutexes or `OSAtomicCompareAndSwapLongBarrier` as it does more work. However, it is more convenient.

Deciding Whether to Use Threading or One-Shot Invocation

There are many reasons to avoid using threads whenever possible.

As discussed earlier, threading introduces nondeterminism into your application. Because the programmer cannot keep the entire graph in mind, there will be bugs.

Because there is no deterministic way of testing every thread interleaving, testing coverage collapses. Threading bugs are very difficult to reproduce, let alone debug. This is the key point. To quote Brian W. Kernighan, *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*

Just to emphasize this point, Apple only fixed `NSOperationQueue` in Mac OS 10.5.7. `NSOperationQueue` was introduced in Mac OS 10.5 to simplify writing multithreaded code, but it included a multithreading bug that occurs on dual-core machines and that would crash any application that used it. Because the iPhone has only one core, it was immune, but the simulator wasn't.

Furthermore, Apple's documentation on which classes are thread safe is incomplete. It does not clearly state which Cocoa objects are safe to invoke from multiple threads at any time (thread safe), which can only be invoked from the main thread (main thread only), and which can be invoked from any thread but not from two threads simultaneously (non–thread safe).

Most classes are not thread safe. For instance, updating `UIViews` in secondary threads causes unpredictable effects. It can be hard to see when your code invokes methods on the wrong thread: As your code changes over time, some methods used by multiple threads might end up invoking methods that update `UIViews`. For instance, you might add KVO methods to update the user interface (UI). However, the setter you are monitoring might be invoked from a different thread. KVO will invoke your new methods to update the UI from that thread. (See http://lists.apple.com/archives/cocoa-dev/2007/May/msg00022.html for more details.)

In general, only immutable objects are thread safe. The "Thread Safety Summary" appendix of the "Threading Programming Guide" provides some guidance that is "subject to change."

Therefore, although it's ugly, I prefer to sprinkle invocations of the runtime loop around unrelated code, rather than adding multiple threads. Similarly, I'd much prefer to optimize an algorithm instead of adding threading. Not only do I keep my application deterministic, but I also improve battery performance.

Types of Layers

This section focuses on standard CALayers because they are the most commonly used layers. There are three other kinds of layers which solve problems you may encounter: CAScrollLayers for displaying content larger than the screen, CATiledLayers for displaying zoomable content, and CAEAGLLayers for displaying OpenGL content.

CAScrollLayer

CAScrollLayers have sublayers that are too large to fit on the screen. CAScrollLayers clip their subviews to fit within their bounds. You can specify whether they support no scrolling (kCAScrollNone), horizontal scrolling (kCAScrollHorizontally), vertical scrolling (kCAScrollVertically), or bidirectional scrolling (kCAScrollBoth) by using the scrollMode property. You can programmatically scroll so that a point (scrollToPoint:) or a rectangle (scrollToRect:) is visible.

CATiledLayer

CATiledLayers provide support for multiple levels of detail for large layers. For instance, if you have a giant photograph such as NASA/ESA's 6000x6000 pixel stunning Hubble Deep Field image, you can display it on your iPhone with a CATiledLayer. Tiled layers use a new level of detail each time you zoom the image two times its size. The layer's delegate's drawLayer:inContext: method is called to draw new levels of detail. The drawLayer:inContext: method must extract the graphics context's bounds using CGContextGetClipBoundingBox to know which part of the image should be drawn. It does not need to scale or crop the image as the graphics context is already set up to do this.

```
- (void) drawLayer:(CALayer*)layer inContext:(CGContextRef)context
{
  CGRect bounds = CGContextGetClipBoundingBox(context);

  // The part of the image to draw is specified by bounds
  // Draw it here
}
```

Watch Out!

> drawLayer:inContext: is invoked by Core Animation within the Core Animation thread.

`levelsOfDetail` specifies the number of levels of detail the tiled layer will cache. `levelsOfDetailBias` specifies how many levels of detail are reserved for zooming out. For instance, setting `levelsOfDetail` to 4 and `levelsOfBias` to 1 means four layers will be cached, and they will be 2x, 1x, 0.5x, and 0.25x the size of the current zoom factor. After setting up a `CATiledLayer`, you'll only need to send it a `setNeedsDisplay` once. Thereafter, changing the tile layer's zoom factor or its `position` causes any necessary redraws. Zooming is achieved by scaling the `transform` on its *x*- and *y*-axes:

```
CATransform3DMakeScale(zoom, zoom, 1.0);
```

CAEAGLLayer

`CAEAGLLayer` provides an OpenGL ES layer to which OpenGL content can be drawn. Older iPhones and iPod Touches support the OpenGL ES 1.1 API, while the new iPhone 3GS supports this and the OpenGL ES 2.0 API. OpenGL ES 1.1 games will run on the new iPhone because Apple emulates the fixed-function pipeline using shaders in its driver.

Unfortunately, learning OpenGL requires a number of books, and is not specific to Cocoa Touch, so this book does not attempt to introduce it. The principles you learn for placing layers in 3D space will serve you well should you decide to learn it. Instead, this section simply shows you how to open a GL layer and lists iPhone-specific performance recommendations.

The OpenGL template created when creating a new OpenGL project provides a good example of how to build a GL Layer. When views are built, their `layerClass` class method is invoked to determine what kind of layer they should use. By default, `layerClass` returns a `CALayer` class object. However, you can override it to return an `EAGLLayer` class object:

```
+ (Class)layerClass { return [CAEAGLLayer class]; }
```

For efficiency, the created layer must be `opaque`. It must be told which format to use to represent colors (`kEAGLColorFormatRGBA8` or `kEAGLColorFormatRGB565`) and whether drawable surfaces should retain their contents after they have been drawn:

```
layer.opaque = YES;
layer.drawableProperties
  = [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithBool:NO], kEAGLDrawablePropertyRetainedBacking,
            kEAGLColorFormatRGBA8, kEAGLDrawablePropertyColorFormat, nil];
```

Finally, you must create a GL context, the analog of a Core Graphics context, which keeps track of OpenGL state. It is at this point that you specify whether you'll be using OpenGL ES 1.1 or 2.0:

```
context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES1];

if (!context || ![EAGLContext setCurrentContext:context])
{
  [self release];
  return nil;
}
```

Rather than drawing in `drawLayer:inContext:`, which would call `setNeedsDisplay`, Apple recommends you hook into `layoutSubviews` to update the frame buffer. The `EAGLContext` method `renderbufferStorage:fromDrawable:` lets you allocate storage for a render buffer (a 2D image buffer). This method replaces `glRenderbufferStorage`. `presentRenderbuffer:` is used to display the render buffer.

Because you're actually rendering to the `EAGLLayer`'s texture, which will be rendered on the screen, and the PowerVR GPU does not support antialiasing when drawing to a texture, OpenGL objects will not be antialiased.

Performance recommendations are as follows:

- Vertices should be rendered as strip-ordered indexed triangles with per-vertex data interleaved. This guarantees the vertex data will be read as a single continuous read from memory rather than scattered reads, which have lower DRAM signaling performance.
- Opaque layers should be rendered first (for example, all dual-textured opaque objects first, with single-textured objects next).
- The PowerVR GPU renders pictures in tiles of 32x16 or 32x32 pixels. Sorting vertex data by screen locality will boost performance.
- Do not sort for depth: The PowerVR performs Hidden Surface Removal faster than other GPU architectures.
- Textures may only be power of 2 sizes up to 1024x1024.
- Using PVRTC compressed textures (rather than JPEG) boosts speed significantly.

- All textures and vertex data must fit within a 24Mb buffer.
- Load textures before rendering. Changing textures during rendering stalls the hardware. Avoid rendering a texture, changing it, and using it again within a single frame.
- Use texture atlases to reduce changing textures.
- Use GL_LINEAR_MIPMAP_NEAREST as mipmapping mode.
- `CAEAGLLayers` do not work well with other layers. Your OpenGL code and Cocoa Touch's layer code will both assume they have complete ownership of the GPU. Both may try to use the entire 24Mb texture and vertex data buffer. To avoid this, set your `CAEAGLLayer` to use the entire screen, and do not use other types of layers simultaneously. For instance, do not place `UIKit` user-interface elements on the same screen as a `CAEAGLLayer`. Similarly, make sure the `CAEAGLLayer` has `opaque` set to `YES` and do not set the layer's `transform` properties.

Although Apple does not specify the performance of each device, game programmers have determined that devices run at different speeds. Here is a list of the currently known models, ranked from slowest to fastest:

- 1st Generation iPod Touch
- Original iPhone
- iPhone 3G
- 2nd Generation iPod Touch
- iPhone 3GS

As a rule of thumb, the first four can render 1 million triangles per second (100 million pixels per second), whereas the new iPhone 3G PowerVR increases this two- to fourfold. Games such as Tap Tap Tap Revenge stop rendering optional visual effects when the frame rate drops too far.

View Controller Hierarchy Used in the Twitter Application

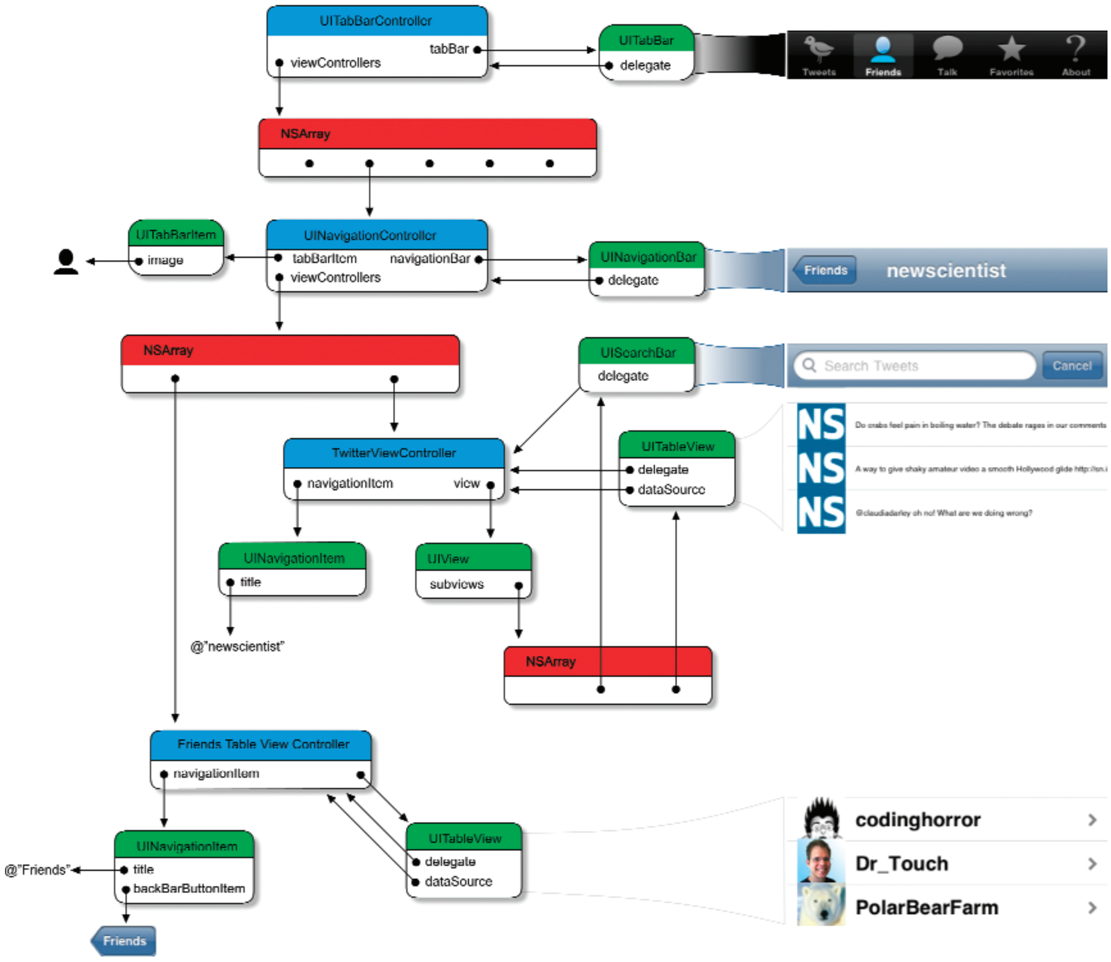


FIGURE D.5
View Controller Hierarchy used in the Twitter Application