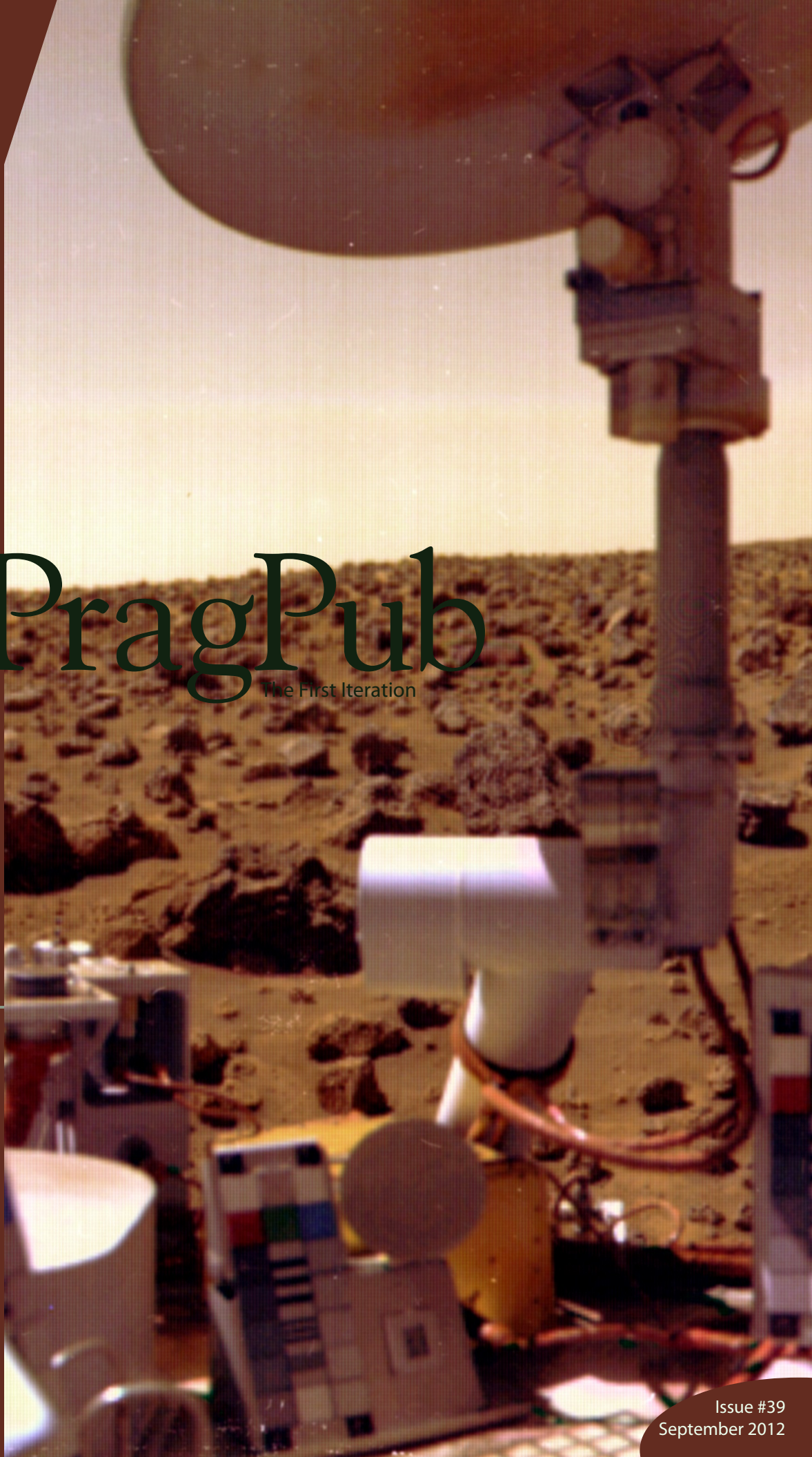
The Pragmatic Bookshelf



# PragPub

The First Iteration

IN THIS ISSUE

* Paul Callaghan on Haskell's type system
* Michael Nygard on the site that always crashed at 5 A.M.
* Alexander Demin on his first iPhone app
* Chris Espinos's three ages at Apple
* John Shade on Apple v. Samsung

Issue #39
September 2012

# Contents

**FEATURES**



Thinking Functionally with Haskell ..... 3
by Paul Callaghan

In which we explore what modern type systems bring to the table.



Tales from the Ops Side ..... 24
by Michael Nygard

Mike Nygard tells another story of his adventures trying to keep large websites up and running. This time it's a site that went down every morning at 5 A.M.




US Visa: My First iPhone App ..... 28
by Alexander Demin

Alexander shares his experience in developing an iPhone app—and shares the code, too.



Threads ..... 47
by Michael Swaine

Chris Espinosa was just a precocious kid when he started working at Apple. He has kids of his own now, he's still at Apple, and he has some great memories.

**DEPARTMENTS**


## Up Front ..... 1

by Michael Swaine

Familiar faces, new ideas. All of our authors this month have appeared in these pages before, but they all have something decidedly new to share. (Or in the case of the history article, something old, but that's the only kind of history we know about.)

## Choice Bits ..... 2

Hands-on Backbone.js, Raspberry Pi, and Cucumber Recipes are topping our chart.

## Calendar ..... 50

Author sightings, upcoming conferences, and other events of note.

## Shady Illuminations ..... 56

by John Shade

John reflects on copying, and how Samsung lost a billion dollars and still won.



Except where otherwise indicated, entire contents copyright © 2012 The Pragmatic Programmers.

Feel free to distribute this magazine (in whole, and for free) to anyone you want. However, you may not sell this magazine or its content, nor extract and use more than a paragraph of content in some other publication without our permission.



Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

# Up Front

## Haskell, History, iOS Development, and the System that Crashed Reliably at 5AM

*by Michael Swaine*



We have two meaty articles in *PragPub* this month, plus some tasty side dishes.

Paul Callaghan returns with another article on the Haskell language. According to Wikipedia, "The language is rooted in the observations of Haskell Curry and his intellectual descendants, that 'a proof is a program; the formula it proves is a type for the program.'" Paul shows you what that means in this deep but engaging article. He shows you how Haskell deals with types, but he goes beyond that to explore just how much sophisticated intellgence can be encoded in a type specification. And from there he goes on to the relationships among testing and types and proofs. It's quite a ride.

Alexander Demin has appeared in our pages before, with an article on concurrency in Google's Go language [U1] in June and an article on building a CPU with only one instruction [U2] in March. He's back this month with something a little more conventional: a step-by-step walkthrough on writing an app for the iPhone. What makes it a little different is that although Alexander is an experienced developer, he had never before written iOS app, or even written anything in Objective-C. So the article is also about a developer trying out a new language and environment and sharing his experience by sharing his code. We think this kind of article can be educational in a slightly different way from the platform expert showing how it's done. Let us know what you think.

Meanwhile, Mike Nygard tells another story of his adventures trying to keep large websites up and running. This time it's a site that went down every morning at 5 A.M. Your editor has another glimpse into personal computer history, with three snapshots from the life of Chris Espinosa, who has been at Apple all of its life and most of his own.

And John Shade weighs in on the Apple-Samsung lawsuit decision.

# Choice Bits

## What's Hot

Hands-on Backbone.js, Raspberry Pi, and Cucumber Recipes are topping our chart.



## What's Hot

Top-Ten lists are passé—ours goes to 11. These are the top titles that folks are interested in currently, along with their rank from last month. This is based solely on direct sales from our online store.

| | | |
|---|---|---|
| 1 | NEW | Hands-on Backbone.js |
| 2 | NEW | Raspberry Pi |
| 3 | NEW | Cucumber Recipes |
| 4 | 1 | Agile Web Development with Rails |
| 5 | 3 | Seven Databases in Seven Weeks |
| 6 | 6 | Programming Ruby 1.9 |
| 7 | 4 | Practical Vim |
| 8 | 9 | The Cucumber Book |
| 9 | NEW | Rapid Android Development |
| 10 | 2 | Deploying Rails |
| 11 | 8 | The RSpec Book |

This is already a pretty fat issue, so our samplings from the twitterverse will be back next issue.

# Thinking Functionally with Haskell

## Types? Tests? We Need a New Word

*by Paul Callaghan*

In which we explore what modern type systems bring to the table.



Imagine an approach to programming where you write down some description of what your code should do, then before running your code you run some automatic tool to see if the code matches the description. That's Test-driven development, you say!

Actually, this is what you are doing when you use static types in most languages too. Types are a description of the code's inputs and outputs, and the check ensures that inputs and outputs match up and are used consistently. Modern type systems—such as in Haskell or above—are very flexible, and allow these descriptions to be quite detailed; plus they are not too obtrusive in use and often very helpful.

One point I'll investigate here is how advances in types are converging with new ideas on testing, to the point where (I claim) the old distinctions are starting to blur and starting to open up exciting new possibilities—hence my suggestion that *we need a new word to describe what we're doing* that is free from preconceptions and out-dated thinking.

So put aside your bad experiences from Java, and prepare to be amazed!

### TL;DR

Apologies for length by the way—but it is a big topic and I wanted to give a high-level view of the whole area. Plus I've had to skimp on a few details to keep this introductory discussion readable. Later articles will fill in the gaps.

Here are the essential points, though:

- Haskell's type system is pretty flexible and mostly unobtrusive (unlike some static systems).
- A good type system is a powerful design language, too.
- We can also work with types in a "type-driven development" style.
- Parametric polymorphism is about using the same code for everything.
- Interfaces provide clean overloading, and encourage and support approaches like DCI.
- Often Haskell feels like a dynamically typed language, though with a good safety net in reserve.
- Haskell has its limits, but the cost-benefit tradeoff is probably in our favor.
- There are several options beyond Haskell, such as dependent types.
- Dependently typed languages are an elegant combination of computation *and* reasoning.

- Many TDD specs can be encoded as (dependent) types, and vice versa.
- I believe the distinction between tests and types is blurring, and each side can learn much from the other to capitalize on this.

## What Are Types For?

Types do sometimes get some bad press. Many people will say types are about nailing details down and trying to avoid errors. This sounds to me a bit dry and negative, too much like a strait-jacket, too *conservative* (if you read the recent Yegge rant).

Some type systems undoubtedly are too restrictive and just don't give enough return for the work required to use them (Java is a prime example) but I'm talking about Haskell's type system and above here, which are quite a different kettle of fish. In this context, I prefer to see and use types as both *a design language and a safety net*. That is, they are more about help and opportunity rather than constraint, and this is pretty much how the type systems have been designed into these modern languages.

Design-wise, I can sketch some of the ideas in my mind as types, add some code, and have the compiler tell me when I'm missing some details or trying something silly. Quite often, I only write down types for the hard stuff when I want to be sure about it, and *I let the compiler work out the rest*. In this mode, the type checker is definitely the slave and not the master! When my code gets past the type-checker, this gives high confidence that the code isn't going to have problems at run-time, at least not in the aspects described via the types. It's a highly valuable tool in development, with obvious similarities to TDD. Many functional programmers even refer to *type*-directed development (and observe that we've been doing it for a while...)

Type systems vary a lot, from minimal (like C) to cumbersome (like Java) to flexible (like Haskell) to awesome (like dependent type theory). The differences involve how articulate the descriptions (i.e. types) can be, and how much reasoning (or inference) power can be applied. That is, how much useful stuff we can say directly in the types, rather than in separate tests or comments or documentation, and how much the tools can exploit this richer knowledge. The more advanced systems offer new ways of using types as well, such as semantic-directed editing and increased automation of the routine aspects of programming.

## A Concrete Example: Sorting

Let's look at various types for a sort routine/method/function. We'll look at how this is handled in several languages and consider what the type is saying. Obviously these library routines are reliable, etc.—the point here is what the types can tell us or help us do.

- Ruby, etc. don't say much, maybe just to indicate what can be sorted by virtue of some object being able to respond to a method called *sort*, and there's a general expectation (though no requirement) that sorting will return an `Array`. We will, however, get run-time exceptions if the sorting can't proceed, e.g. with `[2, "a"].sort` it won't permit comparisons of `Fixnum` and `String`, or my personal favourite gotcha, `[[], nil].sort`.

- C allows something like `int sort(int *vals, int num_vals)` for sorting an array of `int`s in place. Typically, we want to generalize our sort routine to work with arbitrary types, so will find in `stdlib.h` something like `void qsort(void *vals, int num_vals, int size_vals, int comp_fn(void *, void *))`, which allows in-place sorting of an array of items by using a supplied comparison function.

- Java has several options, but the basic pattern is this for general sorting `void sort(T[] a, Comparator<T> c)` which allows sorting of an array of some underlying element type `T`, using some comparison function on `T` values, wrapped up in a comparator object. The `T`, etc. is a placeholder in Java's so-called "generics" mechanism, and gets instantiated when the above code is applied to a concrete array. One effect of the generics is to guarantee that the comparison function is suitable for use on the element types. Earlier versions of Java provided a method `sort(Object[] a, Comparator c)`—this would allow sort to be called on an array containing anything. Notice that the generics version moves some of the expensive run-time checking to a single check at compile time, and provides a bit more confidence that the resulting array hasn't got funny data added.

- Haskell's version is `sort :: Ord a => [a] -> [a]`, which basically says "given a type `'a'` for which ordering is defined, then we can sort a list of such `'a'`s to provide another list of the same element type." It says a pretty much what the Java generics version does, albeit (IMHO) more succinctly and with no OO overhead. We could replace the `Ord a =>` part with an explicit function argument, to give `(a -> a -> Bool) => [a] -> [a]`, i.e. we require a function that takes two arguments and indicates whether a swap of elements is needed. (The `Ord a` stuff is effectively wrapping up a particular comparison function; we'll come back to this.)

But consider now, what is missing? Nothing? We're saying something about the data structures expected as input, and about the output type, plus ensuring that the ordering test can be used with the data. However, it still can't rule out bad definitions like `sort xs = []` (always return an empty list) or `sort xs = reverse xs` (return a reversed copy of the list). Do we want to rule these out?

If we're sensible, then we'll probably sketch out some tests, maybe boiling down to checking assertions like `sort [] == []` (empty list case) or `sort [2,1] == [1,2]` or `sort [2,1,2] == [1,2,2]`. We can even define some properties using the QuickCheck library and run automatic checks on a few hundred random examples to try to uncover counter-examples. Perhaps we'll look at the code and identify some edge cases that need particular attention (depending on the algorithm), or want to check that the implementation has the stability property, i.e. `sort_by first [[1,"b"],[1,"a"]] == [[1,"b"], [1,"a"]]` and similar cases.

Do these have to be encoded as separate tests? Why can't we move some of this thinking into the code itself?

It turns out, we can! and in several ways. Possibilities include "Design by Contract" or some extensions to Haskell. However, I'm going to focus on a more general approach that subsumes all of the others. It is based on "dependent type theory," and effectively it's about *merging types and programs into a single language*. With this advanced language, we have the tools to encode our tests *inside* our program in a sensible way.

We can also encode conditions or invariants that are hard to express as tests—such as requiring that the code will never lose elements or duplicate any, and that indeed it does always return a sorted result.

First, we need to understand how Haskell's type system works, and see how it can be used for good in programming.

## The Language of Haskell's Type System

I suggested that one important feature for a type system is how much it allows us to describe, so we'll start looking at the *language* of Haskell types first. It's perfectly accurate and appropriate to describe a type system as a language—it has words, grammar rules, meanings. Languages differ in what they can be used to express (or articulate), by virtue of the forms of "sentence" available.

Propositional logic can only talk about atomic propositions, like "Rubies are red," and combinations of these via the logical operators, e.g. "Rubies are red and not(Rails is written in C)." Predicate logic allows some more flexibility by allowing the ability to quantify over entities, hence enabling sentences like "For all m, m is mortal -> father_of(m) likes Rails" (ie, every mortal's father likes Rails—probably untrue but we're not talking truth here, just talking about what we can articulate). In this example, we speak of the variable `m` being bound by the `for-all` quantifier. We can have many other kinds of logic, e.g. modal logic where we talk about possibility vs necessity, or 2nd order logic where we can quantify over properties of elements, or—well, whatever we want.

Simple type systems (e.g., Pascal) can be like propositional logic, only able to say basic things, such as "this method takes two Ints and returns a String." Typically, we'd express this as `(Int, Int) -> String`, using the thin arrow symbol as a kind of implication—in the sense of "if you give me two Ints then you get a String back."

Programmers can add new type names, e.g. a `Person` record to contain `name`, `age` and `address`, and use these type names among the ones supplied in the language.

C allows references to functions to be passed around, and these can be represented using nesting, e.g. `(Int, Int -> Int) -> String` for taking an `Int` and a function (from `Int` to `Int`) and returning a `String`. This is OK for basic work, but clearly more flexibility is needed.

Haskell's core type system is a restricted form of predicate logic. It allows variables in the type expressions to stand for arbitrary types, enabling a feature called *parametric* polymorphism. This kind of polymorphism (for there are several) is about *using the same piece of code whatever the actual input types*. For example, taking the length of a list has type `[a] -> Int`, and can be computed without considering the type of list elements: all it needs to do is walk through the list structure counting the nodes. It's completely irrelevant what is in the list, and no choice of element type can affect how the counting works! Similarly, list reversal has type `[a] -> [a]`: again, it is parametrically polymorphic and able to work with any list. Notice that type variable `'a'` appears twice: this means that the input type is the same as the output list, so it can never (e.g.) convert a list of `Int`s to a list of `String`s.

Stepping back a bit, this "it works for everything, and works consistently" view is often precisely what we want to articulate, and the benefits for checking code should be clear. For example, the mapping operation on lists has type

```
(a -> b) -> [a] -> [b]
```

meaning in English, "give me a function from some a to some b, and a list of some 'a's, and you'll get back a list of some 'b's. Notice how it says something useful about what `map` does, that we no longer need to explain in the documentation? Furthermore, when we use `map`, the compiler can check automatically that its use is appropriate.

Similarly, function composition `f . g = \x -> f (g x)` has type `(b -> c) -> (a -> b) -> a -> c`, translated as "if we can convert 'b's to 'c's and 'a's to 'b's, then we can convert 'a's to 'c's, for whatever choice of `a,b,c`." Again, the type clearly says what the requirements are and indicates what we can expect back.

We can use polymorphism when defining own data types too, e.g.

```
data SamePair a = MkSamePair a a
```

to have a constructor that combines two values of the same type. You saw some other examples last time. Let's now look at how the type language is used.

## Type Inference and Type Checking

A language by itself isn't much use—we need some reasoning rules and algorithms to make it useful. That's what we'll cover here. The core inference rule is actually quite simple: if you have a function `f` that expects some value of type `A`, and you have a value `x` of type `C` such that `A` and `C` match in some way, then `f(x)` is firstly OK and also it has type `B`.

```
f : A -> B     x : C
-------------------- if A = C
       f(x) : B
```

Notice that this rule doesn't "do" anything—it just says how we can relate bits of information. So next we can think about the two key algorithms: type *inference* and type *checking*.

Inference takes a piece of code and works out what the type could be, and most versions walk through a syntax tree and apply the above rule from the bottom up.

Checking takes a piece of code and a type, and decides whether the type is appropriate. There are several techniques we can use, such as using the code alone to infer a second type and seeing if it matches the original, or a top-down tree walk that basically uses the above rule in reverse (which is also great for inferring omitted type information).

Many compilers (quite reasonably) use a mixture of inference and checking when compiling the code, so don't get too hung up on the differences. Just be aware of the options, and that we have several techniques to work with the types in our code.

Extra features like type variables fit easily into this framework, with the main change being that our `A = C` test might need to do *unification* (just as in Prolog)

on two expressions, and anything learned from the unification needs to be applied to B. For example, where reverse :: [a] -> [a] and foo :: [Int], then A is [a] and C is [Int], which matches with the substitution a := Int. So we can conclude reverse foo :: [Int] because the B part is updated from [a] to [Int] because of the substitution. If you're happy with this example, then you've just understood the heart of Haskell's type checking. (If you want an interesting challenge, see if you can write this simple inference algorithm in about 30 lines of Haskell. See the forum thread for this article for hints!)

## Some Examples

```
foo1 text = unlines (map reverse (lines text))
```

Will the above definition pass the type checker? What is its type? Given that lines :: String -> [String] and unlines :: [String] -> String, then it can infer that text should be a String, can approve the use of map, and conclude that the result is going to be a String, i.e. conclude foo1 :: String -> String. No programmer annotations needed, and our safety net has passed the code for use.

Let's try some type-driven development now. Suppose I want a function group_on that works a bit like Ruby's group_by, e.g. where (1..10).group_by {|x| x % 2 == 0 } gives {false=>[1, 3, 5, 7, 9], true=>[2, 4, 6, 8, 10]}. For the sake of making the example a bit more complex, I'm going to do it via a wrapper around one of the Haskell library functions. (It could also be done as a fold loop that adds one element to a hash table on each step.) A Haskell List library function groupBy :: (a -> a -> Bool) -> [a] -> [[a]] can group *adjacent* equal elements, e.g. groupBy (==) "Mississippi" gives ["M","i","ss","i","ss","i","pp","i"], but it's not quite what we want, so we need some wrapping. Let's get the type system to help us.

To start with, what's the type we want? Informally, we split a list of something into a hash or map, where the key is the result of some operation and the value is a list of the original items that gave that value. Notice that we're not talking concrete types so we can expect to make this (parametrically) polymorphic—we can use the same piece of code no matter what the actual payload data is. Instead of a Hash, I'll use a simpler type of a list of pairs, e.g. for the above example we expect [(false, [1,3,5,7,9]), (true, [2,4,6,8,10])]

First attempt: list in, list of pairs out, so we come up with something like group_on :: [a] -> [(b, [a])] as the type to aim for. But we need a way to generate the keys in the result, so need to pass in a function for this. Hence group_on :: (a -> b) -> [a] -> [(b, [a])] as the next guess. But—how do we know when two 'b' values are the same? We'll need this to be able to group properly. Basically, we don't know yet. At present, all we know is how to get 'b' values from 'a' values (with the first argument) and that we want to put 'b' values as the keys in the result.

This is actually a key point about polymorphism and worth exploring. Polymorphism in types encodes the minimum we can expect or require, *and does not let us know anything else*.

More concretely, such polymorphic code can't mess around with the data it is working on, because it knows very little about it. This is great for enforcing modularity and such-like.

So how do we get to compare 'b' values? Well, we have to give the code a way to do it, and to keep it simple, I'll pass in another function. Hence we now have `group_on :: (b -> b -> Bool) -> (a -> b) -> [a] -> [(b, [a])]`. Notice how the type now shows the minimum we have to supply in order to use it, and how it succinctly expresses the main aspect of what it's going to do—convert our list into a hash. (Of course, it does not say everything—we'll discuss that later.) Note how it's also acting like part of a specification as well. What we have makes sense, and so next we can develop some code to match the type.

In an ideal world, our Haskell code editor could use the type information to help us build the definition from top to bottom, with us doing the creative parts and it managing the small, inferrable details. In our example here, we know we want to use the `groupBy` library function wrapped up in some way, so we could start by writing `? (groupBy ? (? input))`, where `?` marks the bits we don't know yet, and see what the editor infers for the unknowns—after which we can attack each unknown in turn. (This is a divide and conquer approach, of course.) The above is basically saying we want to do something before and something after the `groupBy`, but we're not sure yet what it is.

We do have this technology now for dependent type languages like Agda, but as far as I know, it hasn't quite made it down to Haskell yet (which is a shame...).

Back to the old-fashioned ASCII text file mode!

However, there is a nice trick to get some targeted support from your typechecker: instead of your unknowns, just use value `()` (i.e., empty parentheses) where you don't know what to put. This is Haskell's dummy or placeholder value, the one we use when some value is needed but we don't want to use anything meaningful. It's also the zero case for the range of notations for pairs (`('a', True)`), triples (`('a', True, 3)`), quads (`('a', True, 3, [Nothing])`), etc., i.e. `()` is a tuple with nothing in. When you use it for unknowns in your code, the type checker will produce a type error indicating that `()`'s type (which is also written as `()`) does not match the type expected at that part of the code. So we could write the following in our Haskell file and compile it:

```
group_on :: (b -> b -> Bool) -> (a -> b) -> [a] -> [(b,[a])]
group_on same eval inp = () (groupBy () (() inp))
```

And instantly get a type error complaining that `() :: ()` (the right-most one) does not have a type `a -> b` because it's being used as a function in the context of `() inp` (i.e. applied to the "inp" argument), but it doesn't match the general `a -> b` type pattern for functions. We can then start trying to refine the `()` values with more concrete versions, and work through the consequences. For example, we might guess `()` should be a map of something, so replace with `map ()`. Then, realize that the new `()` should be a function, hence `map (\x -> ())`. At this point, the checker is satisfied here and it will now get us to work on the argument to `groupBy`. This technique might help some of you.

There are other techniques. My first article said "think data!," so we can think about the input (a list of `'a'` things), and the fact that we want to group them on the basis of some `'b'` result, then consider various ways we can do this while using the library's `groupBy` function.

If you're not happy with the abstract view, it's perfectly OK to think of concrete data instead and work through some examples on paper, maybe even draw pictures of the various stages of the operation. Also useful is to work with a concrete example in the interpreter's REPL and see what you can build up. This technique really does work in FP, since many of your definitions will turn out to be short-ish anyway! It also works in Ruby, of course, so do try it if you haven't already. So let's do that. We know we want `groupBy` in there somewhere, and we can still use the () trick, so we can write the following at the prompt and see what it does:

```
groupBy () [1..10]
```

A type error of course, but it suggests that () should be a function of two arguments. Next iteration:

```
groupBy (\a b -> ()) [1..10]
```

Another type error, because () needs to be a Boolean, so let's think about some test here. By luck, Haskell has `even` in its standard library. We want to group elements if they are both even or both not even, so try this.

```
groupBy (\a b -> even a == even b) [1..10]
```

Finally we get something, but [[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]] is not what we want. But, `groupBy` groups adjacent equal elements, so we need to do a bit more work, maybe trying to sort the list first.

The rest of this example can be downloaded. Check the forum [U1] for the link.

Time passes.

Eventually, we find that the following works for our example

```
map (\xs -> (even $ head xs, xs))
   $ groupBy (\a b -> even a == even b)
   $ sortBy (\a b -> even a `compare` even b) [1..10]
```

And then we can adapt it to code by pulling out the concrete details, hence

```
group_on same eval inp
     = map (\es -> (eval $ head es, es))
     $ groupBy (\x y -> eval x `same` eval y)
     $ sortBy  (\x y -> eval x `compare` eval y) inp
```

We can simplify this in various ways, e.g.. avoiding repeat runs of the `eval` function, or using more overloading (as introduced below), but this will do for now. (One extra detail has slipped in, that we need to know how to order `'b'` values, and I'm quietly allowing Haskell's overloading to handle this.) It passes the type checker, and we can see pretty clearly how it works—a combination of sorting into a suitable order, grouping adjacents, then post-processing the result. One for you to consider: what testing should we do now?

## Comfort Break

Here's my second favourite joke. An eminent professor in computer science was once asked which was the best programming language. He (or maybe a she) pondered for a while then answered:

"Graduate student."

Serious point though: we're certainly not there yet, and we—everyone—have plenty more to explore first. Always reflect on what you'd like in a language rather than sticking to what existing languages give you and soldiering on. Thinking about type systems is part of this.

Here's a relevant and poignant example from Java. If you've used Java, you might have needed the .clone() method a few times. Recall how it gets used though—typically like this.

```
Foo orig = Foo.new();
Foo copy = (Foo) orig.clone();
```

Why is that cast needed on the second line? Is it just because that's the Java way and you've got used to it? Or does it point to a weakness in the language? It is rather annoying to have to write down information that is very obvious to the programmer, which also adds to the "noise" in the program and obscures the "signal."

Sadly, it is a language weakness: Java's type system imposes certain constraints in order to get its version of inheritance to work "safely" and this means that inherited methods get restricted types, that an overridden method must return the same type as the method in its ancestors.

Like most interesting aspects of life, designing a type system is a bit of a balancing act or tradeoff, of juggling flexibility and articulacy with being able to retain useful properties and still be usable by programmers.

Some languages have a good balance, others not, and I'd place Java in the latter camp: the benefits are quite modest, and the costs too high. I would like to put Haskell in the first camp.

However, Java's issues don't mean that type checking in OOP is inherently problematic, or that the only alternative is dynamic checking—even if some people try to sell you this argument. There is much good work in the area that develops alternative approaches. I recommend Kim Bruce's *The Foundation of Object-oriented Languages* (2002) as a useful starting point: it has a highly readable first section which surveys the main languages, including some illuminating discussion of Java's design and implementation, then it proposes several attractive alternatives to Java. A key aspect of Bruce's approach is keeping data and functionality separate, making serious use of interfaces for the latter.

More recently, Bruce has been a key mover in the design of Grace [U2], which is intended as a solid teaching language for OO—and some of his ideas appear there too.

## Interfaces and Type Classes

Interfaces, in the sense used here, work on several levels. Here's a standard example from Haskell that will be familiar to Java programmers—the "comparable" interface. I'll explain it in two stages, first an underlying equality test then the wider less-than-or-equal test.

```
class Eq a where
  (==) :: a -> a -> Bool
class Eq a => Ord a where
  (<=)    :: a -> a -> Bool
  compare :: a -> a -> Ordering     -- LT or EQ or GT
```

The technical term for entities like the above is "type class", intuitively identifying a set of types that provide the listed functionality. I prefer the more informal term "interfaces", since what we're doing is describing some functionality that can be assumed for some type, or a promise of what a value in that type can provide. When you have that interface or promise, you can then write code that uses that interface.

The first definition describes an interface called `Eq` that promises a binary operator == that takes two values and returns a boolean. We can use this operator freely in other code, e.g. counting how many items in a list are equal to a given target, by scanning the list and selecting (i.e. filtering) which items are "the same" as some target, i.e. return True when compared to it.

```
count_same target list = length [ x | x <- list, x == target ]
```

There are other ways to write this definition, e.g. `length . filter (\x -> x == target)`, but that's a bit off topic. Now, how flexible is this code? Which types can we use it with? Well, the only thing we need to know or assume is that the underlying type has == defined for it. For the whole definition, we can say informally: we can count how many values in a list are the same as some target, as long as the target and list elements are the same type, and assuming we can compare values of that type. More formally, expressing this as a type in Haskell we have:

```
count_same :: Eq a => a -> [a] -> Int
```

... which literally says, assuming `Eq` is defined for some `'a'`, and given a value of type `'a'` and a list whose elements are in the same base type, then we can produce an `Int`. The "fat arrow" is used to separate the "preconditions" part from the main type.

Like with the parametric polymorphism above, this type signature can be *inferred automatically* by the Haskell compiler—we don't have to work it out.

*This is a huge difference from Java*, where the compiler is just checking the type information and so the programmer needs to write in all of the type information explicitly. Haskell instead is doing some *type inference* first, to work out what the type should be, then does some type checking to ensure that your code is being used appropriately.

I'll say it again, because it is very important: for a lot of code in Haskell, you can just write down the code and the compiler will infer what the types should be, then check that the types are being used consistently. You do not need to write down lots of type information. It's a bit like programming in Ruby (little or no type baggage) and enjoys some of the benefits of dynamic languages—but with a good safety net in the background.

There's one more detail to explain from the definition of `Eq`, and that's the type signature of `Eq a => a -> a -> Bool` required for (or that can be assumed for) the equality test. That is, assuming `Eq` is defined for some type `'a'`, then the equality test can compare two `'a'` values and return a `Bool`. So `True == False` is

fine typewise, since replacing 'a' with Bool gives Bool -> Bool -> Bool and this matches how we're using it. But, we can't do True == (True,True) since Bool and (Bool, Bool) are different types. It doesn't make sense (in this context) to compare different things for equality. Note also that this restriction is being checked at compile-time, not at run-time, and we certainly don't have to write the code to check this for this ourselves as we do in Java. Neither do we need to add in type casts either—which is another limitation of Java's type system. Incidentally, Kim Bruce's OO-style LOOM and LOOJ languages lift this restriction by adopting similar techniques to Haskell's type classes. (For more info, see my article on DCI and interfaces [U3]).

So, that's equality. The next interface provides ordering tests, and it's introduced as a sub-interface of equality. That is, we're extending the equality interface with a <= test, with the effect that we can assume == is defined if <= is defined. Alternatively, if we want to define <= then we have to define == too. It's a reasonable convention, but you can define a different version if you want, i.e., it's not permanently baked in. (The type class also prescribes an often-useful general comparison test compare which distinguishes between less-than, equals and greater-than cases. This is the detail that snuck into my group_on example—as a result of wanting to sort on the 'b' values.)

There are several ways to equip types with suitable definitions for the interface functions, similar to Java's "implements" mechanism. Firstly, we can do it explicitly by providing code for the functions. Below, we just give a direct definition of equality testing for Booleans. For certain standard type classes, the compiler can automatically *derive* conventional of relevant functions (including the code below), thus saving us work (and avoiding mistakes), though we won't use it here.

```
instance Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Here's a slightly more complex instance definition, concerning equality of lists.

```
instance Eq a => Eq [a] where             -- 1
  []     == []     = True                 -- 2
  (x:xs) == (y:ys) = x == y && xs == ys   -- 3
  _      == _      = False                -- 4
```

Line 1 indicates that this code provides a definition of equality for a list with element type 'a', *assuming* that we already have equality defined for 'a' itself, which is quite reasonable. The definition by cases says (2) empty lists are equal, (3) that when both lists are non-empty then they are equal only if the two head (or first) elements match and the list tails (the rest) match as well, and finally (4) any other case is not equal.

Notice how == is being used with different types on line 3—the first is comparing elements of the list, and the second is comparing lists. This is overloading, where the same name (==) is being used safely with different types, and in particular, the language mechanism is selecting appropriate code to run for each case depending on what the types are. Haskell's class mechanism

was originally designed to handle such overloading in a convenient and sound way, though we've since found it has other great uses too.

Consider what happens when we compare [[[True], [False]]] and [[[True],[True]]]? I.e., a 3-deep nested list of boolean values, or [[[Bool]]]. Behind the scenes, Haskell will construct a "dictionary" that contains code for doing the comparison, and effectively uses the list equality code three times plus equality on Bool to build code that runs the precise code we need. Put another way, it uses the type information to build the code it needs, without us having to spell it out. This adds a lot of power and flexibility.

What do these interfaces provide?

I think there are several related benefits for programming:

- Firstly, it's a great documentation facility—it allows us to clearly say what we're assuming for certain concepts, and to say what assumptions are behind the code using the interfaces. This alone is worth the price of admission(!)
- Rationalization by programming to an interface: we say exactly what we can assume, and our code is only able to use what we've said it can. In particular, it limits the scope for exploiting other details of the data—which greatly helps with modularity and security.
- There are benefits for abstraction, reuse and maintainability too—by assuming only as much as we need, then the code is less likely to contain unnecessary details, and hence be more suitable for use elsewhere; plus, it being simpler then it's going to be easier to change (in fact, if suitably abstract then maybe no changes will be needed, since it has isolated various aspects from each other).
- Safety—e.g. compare some uses of modules in Ruby, where code is included and has to rely on the free variables (or external references) in code being available in the class it's included into. The interface says precisely what is assumed, and the type checker ensures only that is used. (This kind of finger-crossing in Ruby is a detail I find particularly concerning—it is quite a bad code smell in my view. I don't believe we have to accept this weakness.)
- Convenience & overloading: quite often we want to use the same concept, like equality, with various types, but we don't want to pollute our namespace with type-specific names. One name should do for all. Haskell's type system retrieves the particular piece of code to use with a type, just by examining the type of the values involved.
- Ad hoc polymorphism: Parametric polymorphism is about using the same code for everything, and can be contrasted against a kind of polymorphism where everything "behaves" the same by virtue of responding to the same interface, usually via type-specific code. This is known as "ad hoc" polymorphism, in the sense of "determined by case."

Ruby can simulate some of the behaviour, but do notice the value of being explicit and clear about interface use—and not relying on certain implicit features of Ruby. So, is it worth adding some kind of explicit interface feature

to Ruby? Is the tradeoff favourable, particularly if DCI or similar patterns get more popular? Answers on a postcard please.

It's useful to consider what kind of language feature this is, and how it works.

Firstly, it's not magic, in the sense that we can represent the underlying mechanisms inside Haskell itself and do some of the leg-work ourselves. Interfaces are basically records (or dictionaries) whose fields are (mostly) functions, and dependence on interfaces equates to requiring relevant records to be passed in to provide the needed implementations. We can build such records by constant records, or using functions to convert records for small types into records for larger types (i.e. the list equality code above). The type-checker contains an inference component that ensures that relevant dictionaries get created. Finally, there is some syntactic sugar to allow these dictionaries to be used smoothly and consistently (with some efficiency too).

In summary, this is a mechanism that can mostly be programmed within the language, and just uses some extra support to make it more usable and convenient. I thought it worth highlighting this, just to help you visualize how such ideas might apply in other languages, e.g. it's not too big a job to simulate it in Ruby.

## Some Real-world Examples with Interfaces

There has been a lot of interest recently in DCI (Data, Context, Interaction). DCI is an architectural idea developed by Reenskaug—who also proposed MVC—for simplifying and decoupling components in OO programming, basically by encouraging separation of an object's data from the various roles that it plays, and only equipping the object with the role functionality it needs in a particular context. For example, a person could have a role as an employee, and a role as a parent, and the employee part of the application will rarely need to know about the parent role.

This DCI approach echoes some of the ideas from Kim Bruce's work in its separation of data and functionality. It's reassuring to see the same core idea arise from different approaches; from Reenskaug trying to make OO programs more intelligible, and from Bruce trying to rationalise and improve OO type systems.

Now, we can write Ruby code in a DCI style with some extra legwork. Roles can be encoded as modules, then imported into particular instances to equip that instance with the appropriate functionality. We reduce the size of model classes, and gain reusable components too. It works, but, language-wise, this seems a bit "Heath Robinson."

Can we do better? Can we add type information in a useful way that also aids checking of the code and doesn't eat into the flexibility? I believe yes to both. We can set up Haskell-style interfaces to make clear what we are assuming about the source data, and type-checking ensures that we use these interfaces appropriately. Furthermore, the dictionary building process *automatically* assembles the relevant combination of roles needed for any piece of code. (You can find more detail here [U4].)

The next example was inspired by a talk on Hexagonal Rails at the 2012 Scottish Ruby conference. Delivered by Matt Wynne (of Cucumber Book [U5]

and Cucumber Recipes [U6] fame), Steve Tooke and Kevin Rutherford, this talk discussed the need for avoiding deep hierarchical architectures and encouraging a flatter style that is conceptually simpler and also leads to less coupling in code. More modularity and less coupling tends to make code easier to maintain and to understand. They also presented some experiments in restructuring Rails code along these lines, and the experimentation continued into a popular follow-on session where groups of us tried out our own ideas.

It struck me how such experiments and related thinking were being limited by the constructs available in Ruby. The attempts at abstracting controller functionality had to be represented directly in code, were partly being obscured by code details, and there was no easy way to take a more abstract view. Put another way, a good type system is a powerful language for design too, and in the discussions the lack of such a system was telling.

My response was to start sketching out some code that used interfaces and polymorphism to enforce a clean separation of concerns. The persistence role can be managed via an interface that just allows saving and loading of data. Validity of data can be separated off to a layer independent of persistence. Mapping to URLs can also be abstracted out.

Finally, the update action at the controller level can just pull together the functionality from the interfaces (aka roles in DCI terms) and provide a simple connection between them. Such code would only know that (a) it can build some domain-specific data from the form, (b) can try to persist it or get an error message back, and (c) determine the URLS for the next step. Such code will never be able to access domain details or business logic.

By virtue of the overloading, we can also use the same piece of code for most or all controllers in the app. Here's a brief and informal excerpt.

```
-- provides ability to save etc
class Valid a => Persist m a where
  save :: a -> m (Either ErrorMsg a)
-- allows mapping to a URL
class Url r where
  show :: r -> Url
  [...]
create_ constr args
 = do res <- save (constr args)                    -- 1
      case res of                                  -- 2
        Right r    -> redirect (show r)            -- 3
        Left  msg  -> render (template (Just msg)) -- 4
```

Line (1) builds an object from the incoming params, given a suitable constructor; line (2) checks the outcome of persisting the new object; then if OK (line 3) jump to the show page, else (line 4) send the user back to the form with some error message.

Notice how the details of persistence are very separate from details of Url routes, and there's no chance of interference. The controller layer is restricted to mediating between the core app (for building data and saving it) and its web interface. Quite naturally, the interactions are flatter, and indeed only use the minimum info allowed through the interfaces. Notice the DCI feel to this too, where the data's roles are separate and only combined when required. It will be interesting to convert this approach back into Ruby code.

## Pros and Cons—the 80-20 Rule

So, that covers the main ideas from Haskell-98's type system. Let's evaluate it.

I described the design of type systems (and programming languages) as a balancing act: you want the articulacy (freedom to say interesting things) but want to retain some useful automation, ease of use, and conceptual consistency. It is very rare to get both.

Haskell manages a pretty good balance though, I believe. The type system is pretty versatile, and only requires explicit hints when you're using complex overloading. What can't Haskell do, or do nicely?

Well, most of the things I'll cover soon under dependent types! But there are some aspects of traditional OOP that don't translate nicely, primarily the ability to build lists of objects of unrestricted mixed types ("heterogeneous lists"), i.e. we can't write `[2, 'a', False]` in standard Haskell. There are some extensions that allow various cases of such lists, each with their own advantages and disadvantages. If we know in advance which types to expect, then we can code it up as a tagged union and do the decoding ourselves. There are other techniques like reducing the values to a commmon interface, so they all look the same and respond to the same methods, even if they are different underneath.

It is appropriate to invoke an 80-20 rule here.

In my experience, comparing Haskell to its mainstream peers, (much) more than 80% of my code fits adequately inside Haskell and the remaining fraction isn't too painful, certainly no deal-breakers—so I'd rather have the convenience and benefits for the 80% part and live with a few awkward bits for a small section of the code. It's certainly not worth discarding the benefits for the 80% and moving to a dynamic language just because of some issues with a minority part of the program. The 80-20 rule also applies to the programming side of Haskell too—more than 80% of my code works nicely in a pure functional setting, and the awkward stuff takes up less than 20%, and often is much less.

So, what does this say about dynamic typing?

My claim is that Haskell & Co are approaching a level of flexibility and unobtrusiveness that they are becoming a credible alternative, and so many of the traditional objections to static types and traditional justifications of dynamic types are starting to lose their force and so need to be re-evaluated.

Some of these objections are a bit wooly, e.g. drawing conclusions about Haskell from conclusions about Java. One recent Ruby book spends a few pages arguing in favor of dynamic typing because it doesn't need extensive type annotations. It's a good advertisement for Haskell too(!)

There are still some phenomena that even the very advanced type systems can't handle well, but these tend to be rare or contrived, and on balance, I personally would prefer help with the 80% or more of my program that can be covered, and do the best I can on the rest. The alternative, of abandoning help on the 80% just to accommodate the minority aspects, is not too appealing from an engineering standpoint.

Now, what do you think?

## Beyond Haskell: Dependent Types

Haskell is something of a "sweet spot" in language design, balancing a reasonably flexible type system with useful inference properties, but it is by no means the best we can do. So what else is there?

Language designers continually experiment with modest extensions to Haskell's basic system, to see if they can eke out any extra functionality without making the language too hard to use, and mileage varies here.

Instead, let's look at the next big step in language sophistication, where we allow types to become full first-class citizens of our language. The Haskell we've seen so far still keeps the world of data separate from the world of types, i.e. types are a description of happenings in data-land and they do not meet, and this puts some limits on what we can articulate.

Dependent type theory lifts many of the limits. Data and code can appear in types, and types can appear inside code as data. We end up with a single uniform language, not two (data vs types), and can start to do some very interesting things in this uniform language. Quick, let's see some practical examples!

```
reverse :: (A:Type)(n:Nat) Vec A n -> Vec A n
```

Haskell's reverse just promised to give a list of the same type back. With dependent types, we can include information about list length too, hence encode that reverse preserves type *and* input size.

The signature literally says, given some type A (for elements) and a length n, then reverse takes a vector (our name for a sized list) of size n and returns a vector of the same size. So it's a guarantee that we don't add or lose elements.

How about appending one vector to another?

The following says that appending vectors of size m and n gives a vector of their combined size, m + n!

```
vappend :: (A:type)(m,n:Nat) Vec A m -> Vec A n -> Vec A (m + n)
```

So, notice how we've taken a step up in the kinds of information we can represent inside our types, with consequences for what the compiler can do with it! We get more articulacy in data definitions too, e.g. to build a sized vectors we need only the following, a vnil constructor that produces a vector of size 0, and a vcons constructor that adds an element onto a vector of size n to get one of size (n + 1).

```
vnil :: (A:Type) Vec A 0
vcons :: (A:Type)(n:Nat) A -> Vec A n -> Vec A (n+1)
```

It's now impossible to get the size of the vector wrong, and we can never use vnil where a vector of size 1 or more is expected (since 0 != n + 1).

This has implications for pattern matching too, e.g., if we define the head function for a non-empty vector (size n + 1) then the vnil pattern is impossible and never needs to be handled.

This next bit may seriously blow your mind.

## Propositions are Types, and Proofs are Programs

We can use any values of arbitrary complexity inside type expressions, not just simple values like numbers. We can even encode logical propositions too, and use these to express pre and post conditions, and more. For example, the following represents a property of equality, that when two things are equal then what we can do with one thing we can do with the other.

```
foo2 :: (A:Type)(P:A -> Type)(a,b:A)Eq A a b -> P(a) -> P(b)
```

We can do more than just encode logical propositions: we can work with *proofs* of them *entirely inside the language* as well. If you've done any logic before, you may recognise the following as a statement of transitivity of implication, i.e. "if B implies C, and A implies B, then A implies C."

```
(B -> C) -> (A -> B) -> (A -> C)
```

How do we "prove" this? The usual way is to find the combination of inference rules that yield the above statement as the final conclusion. Here's a better proof:

```
f . g = \x -> f (g x)
```

No, this isn't a typo. It *is* the definition of function composition. Recall that its polymorphic type in Haskell is `(b -> c) -> (a -> b) -> (a -> c)`. Look familiar?

This is no accident. There is a striking similarity between how functions work and how logical implications work, and we can use this as a way to encode proofs of propositions as programs. It has a name too, in fact several names because the idea appeared in several places independently around 60 years ago. I'll use the conventional "Curry-Howard Correspondence" for now.

Pause to consider the huge significance of this idea!

Checking validity of a proof is then a case of checking the type of the program. Finding a proof equates to developing a program with the required type (go tease some mathematicians!). We can also think of the program as a way to generate proofs of the conclusion from proofs of the hypotheses: e.g., if we have a proof of `B -> C` and one of `A -> B` then we can use the definition of function composition to build a proof of `A -> C`! So function composition is a bit like a theorem we can apply to get from one stage of reasoning to another.

How about mathematical induction? Ever noticed that it is simple recursion? We can write down the principle as a type that says a lot of what we understand about induction: we have some property `P` to establish for all numbers; then a proof that `P` holds for `0` (the base case); and a proof that if `P` holds for some `m` then it holds for `m + 1` (the step case); finally, if given some `n` then the overall result is a proof of `P` for that `n`.

```
nat_induction :: (P : Nat -> Prop)
                 (base : P 0)
                 (step : (m:Nat)P m -> P (m+1))
                 (n : Nat) P n
```

What does the proof of this look like? It's a recursive function that works through `n` to generate the proof it requires! I.e., `nat_induction P base step 3` yields `step 2 (step 1 (step 0 base))`.

So, the more powerful language allows us to encode complex logical propositions as (dependent) types and the programming language inside it allows us to develop and use proofs, all inside the same language, no external tools needed. This really does represent a language that combines—in a clean, elegant, and powerful way—both computation *and* reasoning.

## Another Look at Sorting

We started off looking at the type of `sort`, and now know enough to see what we can really do with dependent types.

```
     A:Type           n:Nat          v:Vec A n
     lteq:Sigma[f: A -> A -> Bool]is_ordering(f)
------------------------------------------------------------
sort A n lteq v : Sigma[w:Vec A n] is_permutation_of(v,w)
                                  && is_sorted(w,lteq)
```

Don't panic! Let's take it piece by piece. As before, sorting is polymorphic, so we can use it on vectors of any element type. Input length can be anything, and the result contains a vector of the same length, i.e. length is preserved. We also have an argument `lteq` that supplies the information about comparing A values—and as you may have guessed, this will be our less-than-or-equal test, where a result of True means "don't swap" and False means "do swap".

The extra pieces are to do with properties we require from the input, and properties that will hold for the result. For the `lteq` function, not just any 2-ary predicate will do, e.g. using a function that always returns True will not produce a sorted result. Briefly, the construct `Sigma[x:T]P(x)` is pairing some data `x` with some more data `P(x)` that depends on `x`. Above, we're pairing the `lteq` function with a proof that it has the properties required for less-than-or-equal. And in the result, we get the result of sorting plus proofs that the output is a permutation of the input (i.e., we're rearranging data) and that the result really is in the sorted order implied by our choice of `lteq`.

So yes, this is more detailed than the types we saw at the start, but if these are the properties of sorting that we care about, then no further testing is required. The type checker has it covered. And if we change the code then either the code still type checks, or we have to justify why the change is safe by modifying the code so it does type check.

Are dependent types more verbose? Yes, because there is more going on. However, there are ways to hide away some of the obvious information and push the core ideas to the fore. In the above, `A` and `n` are easily inferred from the context, and thus rarely shown, so leaving the important inputs of the comparison function and the vector to be sorted. One thread of research in this area looks at other kinds of abbreviation mechanism to help with practical programming, such as making it easier to bridge between simple data and more complex types (which was one of my specialities).

We're also making great progress with tool support. It's often suggested that type information can be used by code editors to make informed decisions about editing support, and this is particularly true with dependent types. There is much more information available in a processable format (i.e. not just in comments or in the programmer's head), and this can be used to great effect: really powerful type-directed development. Such tools begin with some

statement of what we want to develop, represented as a type, then we use a mix of backwards and forwards reasoning to develop the program. We can leave some details as unknown too, kind of what we simulated in Haskell with the () value, but here the tools provide much more powerful handling of unknowns, inferring immediately obvious values and providing kinds of automatic inference for the more complex steps.

Here's a quick example: developing a proof of (B -> C) -> (A -> B) -> (A -> C). This type is set as our initial goal, and the corresponding proof (i.e. program) as the first unknown. We can shift the arguments into the context to get f : B -> C, g : A -> B, x : A with a new goal of ? : C. That is, we need some combination of f,g,x that generates a C value. We can fill in the full answer if we want, or we can build it up in pieces. For example, we can suggest f ? as the solution to the goal, and get a second goal ? : B, i.e. the argument we need to pass to f. That goal is solved by g x. The editor will then show the final result, that our proof/program is \ f g x -> f (g x). Some editors can also do advanced analysis on the structure of goals and fill in some of the code automatically, such as presenting us with only the data patterns that can occur and leaving the right-hand sides to us.

These tools are very powerful, and have been used to help fully and completely build formal proofs of real results like the Four-color Theorem. They also represent a very different and powerful way to program, and have much potential.

If you want to know more about dependent types right away, I suggest you start looking at the Idris language [U7] for a language in the Haskell style. There are more advanced languages (and tools), such as Agda or Epigram or Coq or Cayenne, though they can be a bit steep as an introduction.

## Back to Earth

I hope you appreciate now that advances in type systems are starting to converge with ideas in TDD. More work is needed, of course, but the exciting possibilities are clear.

What can TDD learn from tests? What can types learn from TDD? Personally, I'm in a strange position, of using TDD for Rails work but also aware of the possibilities in types. It's, um, interesting. And annoying, at times. In the long term, I envisage a framework where the distinction between types and tests isn't so significant, and we are all able to work flexibly and confidently with good support from our tools. I have no idea what we should call this though!

In the meantime, I think types work can benefit from TDD in several ways:

- be encouraged by the great ideas and range of work in this area, particularly real-world code
- alternative perspectives on code verification and development practices
- lots of case studies to think about, analyze, and maybe adapt/translate

And how about this for TDD?

- Stronger DSL use to encode the properties we care about, and making that DSL more flexible. Specs encoded as types tend to be more succinct than TDD specs, and I believe it's worth considering how the DSL for specs can be streamlined in a similar way.

- Encouraging more decomposition of code to promote testability. Types get unwieldy if we try to do too much in a piece of code, so functional programmers often break such code into pieces and develop them independently. It's well known in TDD that some code is more testable than other code, typically code that is smaller, more modular, less reliant on side effects. Can we promote this in TDD by orienting the DSL for specs toward such code, making the more complex cases less attractive?
- Approach to side effects. Side-effects can be a pain in type checking, and it seems people are realizing this in tests too, as they make it harder to isolate the things that need to be tested. Again, can the DSL be designed to promote this aspect of testability?
- Other approaches to mock objects? Mocks seem to cause a lot of confusion. The purpose is to set up fake inputs to some test with predictable behaviour such that certain paths in the tested code are exercised. Put another way, it's setting up some preconditions or assumptions under which the test should be run. We can do this in a dependent types setting in a more abstract way by passing in hypotheses of how such objects behave, e.g. passing in a value of type `(x:A) Eq (f x) (x + 3)` indicates that some `f` adds 3 onto its input, and we can use this hypothesis inside the proof of the test to get values from uses of `f`. So, instead of building objects that have some run-time behavior, we instead frame that behaviour via hypotheses. This view may also help simplify the DSL for mock-like behavior.
- Why do tests have to be in a separate file? Is this a practical detail or a theoretical detail? In the types world, we believe the types and code belong together. Is this possible for tests?
- Using tests to drive code generation. Given some type signature, the tools can sometimes predict what structure the corresponding code should have, and this is something dependent types people use often to cut down on work. Can the same be done for TDD specs? It seems feasible, e.g. to parse a TDD spec and generate as much of the corresponding class as possible, maybe including code fragments, error messages, etc., helpful comments to simplify the development work and side-step certain transcription errors. Changes in the spec could automatically flag diff conflicts in the code. (I wish I had time to try this.)

Indeed, considering ideas such as these may pave the way to better inter-operability in future.

Final thought: how far can existing tests be translated to types? Or existing types be translated to tests? I think the overlap is significant. That says something.



**About the Author**

Dr Paul Callaghan rides a big, fast motorbike, and suggests that this informs his programming style too. Some people say bike riding is too risky, but you can manage the risk in various ways. Defensive riding means looking ahead, anticipating problems, and actively minimizing risk by avoiding trouble. One can program like this too.

Paul also flies big traction kites and can often be seen being dragged around inelegantly on the beaches of North-east England, much to the amusement of his kids. He blogs at free-variable.org [U8].

Send the authors your feedback [U9] or discuss the article in the magazine forum [U10].

**External resources referenced in this article:**

[U1] http://forums.pragprog.com/forums/134

[U2] http://gracelang.org

[U3] http://www.free-variable.org/2012/06/dci-and-programming-with-interfaces/

[U4] http://www.free-variable.org/2012/06/dci-and-programming-with-interfaces/

[U5] http://pragprog.com/refer/pragpub39/titles/hwcuc/the-cucumber-book

[U6] http://pragprog.com/refer/pragpub39/titles/dhwcr/cucumber-recipes

[U7] http://idris-lang.org/

[U8] http://free-variable.org

[U9] mailto:michael@pragprog.com?subject=haskell

[U10] http://forums.pragprog.com/forums/134

# Tales from the Ops Side

## The 5 A.M. Problem

*by Michael Nygard*

Mike Nygard tells another story of his adventures trying to keep large websites up and running. This time it's a site that went down every morning at 5 A.M.



*Several years ago, I worked for a company that offered 24x7 operations for websites that we didn't create. As crazy as it sounds, we would not only support applications and systems that our clients built but also offer uptime and performance guarantees. During that time, I got what you could call a "crash course" in operations and in what it meant to build resilient systems. These systems taught me how well our typical applications were prepared to survive the harsh rigors of production. The answer was, "Not well at all."*

*In this series, I relate some incidents from that time. Names have been changes to protect the parties involved, but the essential details and interactions are all accurate. You may find these vignettes entertaining or enlightening, or you may shake your head at how unprepared we all were. As you read, please bear Kerth's Prime Directive in mind. "Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand." Also, it's no fun to tell stories about times when everything went right!*

One of the sites I launched developed this very nasty pattern of hanging completely at almost exactly 5 A.M. every day. This was running on around thirty different instances, so something was happening to make all thirty different application server instances hang within a five-minute window (the resolution of our URL pinger). Restarting the application servers always cleared it up, so there was some transient effect that tipped the site over at that time. Unfortunately, that was just when traffic started to ramp up for the day. From midnight to 5 A.M., there were only about 100 transactions per hour of interest, but the numbers ramped up quickly once the East Coast started to come online (one hour ahead of us Central Time folks). Restarting all the application servers just as people started to hit the site in earnest was what you'd call a suboptimal approach.

On the third day this occurred, I took thread dumps from one of the afflicted application servers. The instance was up and running, but all request-handling threads were blocked inside the Oracle JDBC library, specifically inside of OCI calls. (We were using the thick-client driver for its superior failover features.) In fact, once I eliminated the threads that were just blocked trying to enter a synchronized method, it looked as if the active threads were all in low-level socket read or write calls.

The next step was tcpdump and ethereal. (Ethereal has since been renamed Wireshark.) The odd thing was how little that showed. A handful of packets were being sent from the application servers to the database servers, but with no replies. Also nothing was coming from the database to the application servers. Yet monitoring showed that the database was alive and healthy. There

were no blocking locks, the run queue was at zero, and the I/O rates were trivial.

By this time, we had to restart the application servers. Our first priority is restoring service. We do data collection when we can, but not at the risk of breaking an SLA. (Service-level agreement: a contractual obligation to provide a service to a measurable, quantitative level. Financial penalties accompany the violation of an SLA.) Any deeper investigation would have to wait until it happened again. None of us doubted that it would happen again.

Sure enough, the pattern repeated itself the next morning. Application servers locked up tight as a drum, with the threads inside the JDBC driver. This time, I was able to look at traffic on the databases' network. Zilch. Nothing at all. The utter absence of traffic on that side of the firewall was like Sherlock Holmes' dog that *didn't* bark in the night—the absence of activity was the biggest clue. I had a hypothesis. Quick decompilation of the application server's resource pool class confirmed that my hypothesis was plausible.

Socket connections are an abstraction. They exist only as objects in the memory of the computers at the endpoints. Once established, a TCP connection can exist for days *without a single packet being sent by either side*. (Assuming you set suitably perverse timeouts in the kernel.) As long as both computers have that socket state in memory, the "connection" is still valid. Routes can change, and physical links can be severed and reconnected. It doesn't matter; the "connection" persists as long as the two computers at the endpoints think it does.

There was a time when that all worked beautifully well. These days, a bunch of paranoid little bastions have broken the philosophy and implementation of the whole Net. I'm talking about firewalls, of course.

A firewall is nothing but a specialized router. It routes packets from one set of physical ports to another. Inside each firewall, a set of access control lists define the rules about which connections it will allow. The rules say such things as "connections originating from 192.0.2.0/24 to 192.168.1.199 port 80 are allowed." A TCP connection attempt starts when one host sends a SYN packet to another. When there's a firewall in the way, it gets to make a decision. When the firewall sees an incoming SYN packet, it checks it against its rule base. The packet might be allowed (routed to the destination network), rejected (TCP reset packet sent back to origin), or ignored (dropped on the floor with no response at all). If the connection is allowed, then the firewall makes an entry in its own internal table that says something like "192.0.2.98:32770 is connected to 192.168.1.199:80." Then all future packets, in either direction, that match the endpoints of the connection are routed between the firewall's networks.

So far, so good. How is this related to my 5 A.M. wake-up calls?

The key is that table of established connections inside the firewall. It's finite. Therefore, it does not allow infinite duration connections, even though TCP itself does allow them. Along with the endpoints of the connection, the firewall also keeps a "last packet" time. If too much time elapses without a packet on a connection, the firewall assumes that the endpoints are dead or gone. It just drops the connection from its table, as shown in the figure. But TCP was never

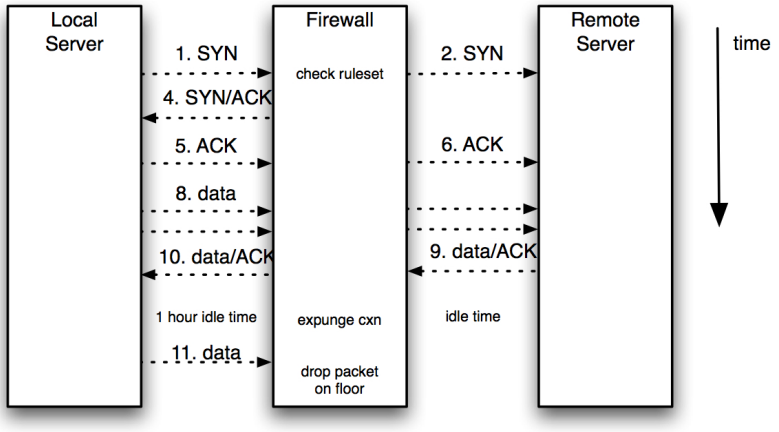
designed for that kind of intelligent device in the middle of a connection. There's no way for a third party to tell the endpoints that their connection is being torn down. The endpoints assume their connection is valid for an indefinite length of time, even if no packets are crossing the wire.

After that point, any attempt to read or write from the socket on either end does *not* result in a TCP reset or an error due to a half-open socket. Instead, the TCP/IP stack sends the packet, waits for an ACK, doesn't get one, and retransmits. The faithful stack tries and tries to reestablish contact, and that firewall just keeps dropping the packets on the floor, without so much as an "ICMP destination unreachable" message. (That could let bad guys probe for active connections by spoofing source addresses.) A Linux box, running on a 2.6 series kernel, has its tcp_retries2 set to the default value of 15, which results in a *twenty-minute* timeout before the TCP/IP stack informs the socket library that the connection is broken. The HP-UX servers we were using at the time had a thirty-minute timeout. That application's one-line call to write to a socket could block for thirty minutes! The situation for reading from the socket is even worse. It could block forever.

So the stack trace claimed that everyone was busy with database calls, but the network trace showed that no database work was happening. What next? Time to decompile something.

When I decompiled the resource pool class, I saw that it used a last-in, first-out strategy. During the slow overnight times, traffic volume was light enough that one single database connection would get checked out of the pool, used, and checked back in. Then the next request would get the same connection, leaving the thirty-nine others to sit idle until traffic started to ramp up. They were idle well over the one-hour idle connection timeout configured into the firewall.

Once traffic started to ramp up, those thirty-nine connections per application server would get locked up immediately. Even if the one connection was still being used to serve pages, sooner or later it would be checked out by a thread that ended up blocked on a connection from one of the other pools. Then the one good connection would be held by a blocked thread. Total site hang.




Idle Connection Dropped by Firewall

Once we understood all the links in that chain of failure, we had to find a solution. The resource pool has the ability to test JDBC connections for validity

before checking them out. It checked validity by executing a SQL query like SELECT SYSDATE FROM DUAL. Well, that would just make the request-handling thread hang anyway. We could also have the pool keep track of the idle time of the JDBC connection and discard any that were older than one hour. Unfortunately, that involves sending a packet to the database server to tell it that the session is being torn down. Hang.

We were starting to look at some really hairy complexities, such as creating a "reaper" thread to find connections that were *close* to getting too old and tearing them down before they timed out. Fortunately, a sharp DBA recalled just the thing. Oracle has a feature called *dead connection detection* that you can enable to discover when clients have crashed. When enabled, the database server sends a ping packet to the client at some periodic interval. If the client responds, then the database knows it is still alive. If the client fails to respond after a few retries, the database server assumes the client has crashed and frees up all the resources held by that connection.

We weren't that worried about the client crashing, but the ping packet itself would be enough to reset the firewall's "last packet" time for the connection, keeping the connection alive. Dead connection detection kept the connection alive, which let me sleep through the night.



**About the Author**

Michael is a veteran software developer and architect. His background is either "well-rounded" or "checkered" depending on how charitable you'd like to be. He has worked in Operations (including for a "Top Ten" internet retailer), sales engineering, and as a technology manager and executive.

In 2007, Michael wrote Release It![U1] to bring awareness of operational concerns to the software development community. This early influence in the DevOps movement showed developers how to write systems that survive the real world after QA.

Send the authors your feedback[U2] or discuss the article in the magazine forum[U3].

**External resources referenced in this article:**

[U1] http://pragprog.com/refer/pragpub39/titles/mnee/release-it

[U2] mailto:michael@pragprog.com?subject=case-study

[U3] http://forums.pragprog.com/forums/134

# US Visa: My First iPhone App

## An Adventure in Objective-C and Cocoa

*by Alexander Demin*

Alexander shares his experience in developing an iPhone app—and shares the code, too.




Being a programmer and having an iPhone and a Mac but not trying to write a mobile app was a little embarrassing for me. Eventually I decided to fix it and dived into Objective-C and Cocoa.

Luckily I had come across a problem that nicely fitted the requirements for a first iOS app. The problem was real, useful, but also not very complicated.

Disclaimer: this is not only my first iOS application. It is even worse. This is my first application in Objective-C ever. I don't claim in any way that the implementation is efficient or elegant. But I hope that at least it is a quite simple, clean, and complete example of what a programmer writing a first iApp will run into. If you find any issues with the code, please let me know.

## Houston, We Have a Problem!

Last year I applied a few times for a US visa in London. Unfortunately, each time my application required what they called "administrative processing." After your application is formally accepted, you are given a batch number. At the embassy website there is a PDF file with statuses of visa applications currently in "administrative processing." The file is updated regularly, and you have to periodically check your status there to know the next step (provide more details, send a passport, etc.) You click on a link [U1], the browser brings up the PDF file, and you search for your batch number to find the update.

I came up with the idea to automate this routine. I realized it could be an iPhone app where I can put in my batch number once and then retrieve my current status by only pressing a refresh button. Or maybe the app could check the status in the background and notify me when it gets changed.

I should clarify that this is not just a Macintosh coding example. If you're stuck with Windows, there is some relief. Objective-C is available on Windows with Cygwin or MinGW. Moreover, there is a GNUstep [U2] project bringing AppKit and Foundation libraries to Windows. With GNUstep you develop a fullblown UI application for Windows. But I'm not going to dig that deeply, and we will only create a command-line application capable of downloading the PDF file and parsing it. The application will work on both OSX and Windows (via GNUstep). But the rest of the application, its UI, of course, will be only relevant for Mac users.

## GNUstep Prerequisites for Windows Users

I found a couple of extremely useful blog posts:

- *Learn Objective-C on Windows* [U3], explaining how to install GNUstep and try a trivial application.

- *Clang and Objective-C on Windows* [U4], explaining how to build the latest Clang compiler for your GNUstep environment. Unfortunately, the GCC compiler bundled within the GNUstep packages does not support Objective-C features required by Apple. Also, there is no official Clang installer for Windows. That is why you have to build it by yourself. I followed the recommendations from this blog post, and it all worked for me.

## Getting Familiar with Objective-C and the iOS API

I had no Objective-C and Cocoa experience whatsoever. I also heard that Objective-C is quite the strange beast compared to C++ and Java, so I started at the very beginning and thumbed through a few books:

1. *iOS Programming: The Big Nerd Ranch Guide, 3/e* [U5]
2. *Objective-C Programming: The Big Nerd Ranch Guide* [U6]
3. *Programming in Objective-C (4th Edition)* [U7]

Plus one more neat document called "From C++ to Objective-C" [U8].

I split the application into three main parts:

- PDF parser
- PDF downloader (without sticking to a particular UI implementation)
- UI

After getting into Objective-C, I would say that if you are an experienced C/C++ developer and familiar with basic concepts of creating UI (I personally spent years programming in Delphi and C++Builder), Objective-C and Cocoa are not really difficult to understand. I would only recommend focusing on Objective-C memory management, because after C++ with RAII or Java with the garbage collector, you may find memory management in Objective-C a bit awkward. You have to manually control allocating and freeing memory as in C, and the only mechanism provided by the runtime for you is reference counting. It is better that a purely manual approach as in C, but it's still tough to do it right. You must precisely follow Objective-C memory management guidelines and naming conventions for constructors. Otherwise you will inevitably end up with memory leaks. This is exactly what happened to me in the very first version of the application. The good news is that the Xcode code profiler does an amazing job helping to identify memory leaks.

I will describe a few of my observations about Objective-C and Cocoa. I have found them quite interesting from the C/C++ developer's point of view. A few pieces of code from the project will help demonstrate them. I'm not going to explain every single line though. Instead, I'd like you to get the taste of Objective-C, if you are not familiar with this language at all.

To begin with, let's have a look at how Objective-C names class methods. It looks almost like a natural language. For example, if I say "please, find a needle in a portion of some data and add the result to a list implemented as a mutable array," in Objective-C it will be:

```
+ (bool)findInPortion:(NSMutableData *)someData
               needle:(NSString*)aNeedle
            andAddTo:(NSMutableArray*)aList {
   ...
}
```

If you read this code from top to bottom, left to right, it sounds almost like the original English sentence. Formally, the signature of this method is findInPortion:needle:andAddTo:. The parameters have names, and the names are part of the method signature. By choosing good names for the parameter variables (someData, aNeedle and aList) you can form almost natural sentences. Agreed, it is wordy, but the incredible code prediction system in Xcode helps you type such long statements very easily. Also, it is worthwhile to point out a somewhat unusual way to break long statements down into multiple lines. In Objective-C they are aligned by the : symbol separating the argument name and the variable.

One more observation. Objective-C uses unusual syntax to call class methods. For example, instead of the quite common dot notation:

```
NSMutableArray* list = NSMutableArray.alloc.init;
```

you write:

```
NSMutableArray* list = [[NSMutableArray alloc] init];
```

It looks strange, but it is a question of habit. Again, the Xcode code prediction system helps put square brackets sometimes without even your typing them.

Objective-C and Cocoa actively use some design patterns. One of them is a delegate pattern. Delegates are everywhere in Cocoa. For example, I used the NSURLConnection class to download the PDF file. When you instantiate this class, you must provide a delegate, NSURLConnectionDelegate, whose methods allow hooking up into the process of downloading.

Eventually, after a couple of weeks of reading at evenings and experiments, and I was ready to sketch a skeleton of my iOS application. The next part of the Ballet de la Merlaison was about parsing the PDF format.

## PDF Parsing

The document containing visa application statuses is a PDF file. Luckily, PDF manuals are available online [U9] on the Adobe website. I used a document called "PDF Reference third edition, Version 1.4" [U10].

I came up with a quite straightforward implementation of the PDF parser. The parser routine is called each time when a chunk of data is received from the downloader. The parser tries to locate blocks of data marked by stream and endstream markers. Then such blocks are unzipped by zlib/inflate, and the parser checks whether your batch record appears in the chunk. If yes, it extracts the visa status update and adds it to a list (a batch number may have several updates in that PDF file). Finally, the parser cuts out the processed piece of data (between stream and endstream markers) from the buffer.

The algorithm:

1. Find each block in the PDF data surrounded by the stream\r\n and endstream\r\n markers. The content of such blocks is packed using the zlib deflate algorithm. We inflate it.

2. Each unzipped (inflated) block of data is a plain text with PDF markup tags. We need to locate blocks of the text embraced by "BT\r\n" (Begin Text) and "ET\r\n" (End Text) tags. All these blocks will be collected in a list of strings.

3. Inside each block of text identified on the step 2 (elements of the collected list of strings) we need to find all substrings embraced by parentheses (( and ) characters). Then we join all these substrings, and the result text will be a plain text representation of the user information we are looking for. So on this step we will be processing each string found on the step 2 by removing all substrings that are not surrounded by ( and ).

4. Finally we have to identify the structure of the plain text. The batch status information in our PDF is formatted as a three-column table. The columns are: batch number, status, and date. Unfortunately, the PDF also contains some extra text (headers, footers, etc.) which will also appear in our parsed plain text along with the useful information. We will identify the table column fields by iterating sequentially along the list of strings. If the current string looks like a batch number (11 decimal digits), we take it as the batch number, and the following two strings will be interpreted as the status update and the date accordingly. We take these three string and jump over them further to find the next potential batch number.

Of course, we pretty much hard code the format, and if clerks at the US embassy change the format, it may not work anymore. But at the moment these four steps work fine for parsing the existing PDF.

## The command-line application

Now we have everything in place to build up a command-line application for downloading the PDF file, parsing it, and looking for a particular batch number. As I said, this application can be built natively on Mac using the latest Xcode or on Windows via GNUstep and the latest Clang compiler (3.2+). Importantly, the source files of this command line program will be used unchanged in the fully functional UI iOS application.

There are several files involved:

- BatchPDFParser.m (and .h)—The PDF parser.
- NSURLConnectionDirectDownload.m (and .h)—The downloader. This file contains plumbing around the NSURLConnection class (initialization, delegates, waiting loop).
- DirectDownloadDelegate.m (and .h)—The delegate attached to NSURLConnection. This delegate class provides callbacks invoked by NSURLConnection notifying us about different events during downloading.
- ViewController.m—The ViewController prototype. This file contains glue between the downloader and the UI. OSX and iOS use the MVC

(Model-View-Controller) paradigm. So, the controller provides the link between visible UI components (buttons, text fields, check and radio boxes, etc.) and invisible ones driving the business logic. Because our batch parser and downloader will be the parts of the full-blown UI application, we prepare the prototype of the UI controller. For the command-line application it will contain a few stubs only. This is the only file that is slightly simplified compared to its full-blown version.

- main-cli.m—the main application entry point.

I will list files one by one, putting commentaries along the code.

*BatchPDFParser.h*

This file contains declarations of the Batch object representing a single batch update and the BatchPDFParser providing the findInPortion:needle:andAddTo: method. Note, this method is static (see the + at the beginning of the line) and can be used without class instantiation.

```
@interface Batch: NSObject {
  NSString *batchNumber, *status, *date;
}
@property (atomic, copy) NSString* batchNumber, *status, *date;
@end
@interface BatchPDFParser: NSObject
+ (bool)findInPortion:(NSMutableData *)data needle:(NSString* const)needle
                                              andAddTo:(NSMutableArray*)list;
@end
```

*BatchPDFParser.m*

This file contains the implementation of the PDF parser.

```
#import <Foundation/Foundation.h>
#import "BatchPDFParser.h"
#import "zlib.h"
@implementation Batch
@synthesize batchNumber, status, date;
- (void) dealloc {
    [batchNumber release];
    [status release];
    [date release];
    [super dealloc];
}
@end
@implementation BatchPDFParser
```

The findInData:fromOffset:needle: private method searches for a given string in a block of data (similar to strstr()).

```
+ (int) findInData:(NSMutableData *)data fromOffset:(size_t)offset needle:
    (char const * const)needle {
    int const needleSize = strlen(needle);
    char const* const bytes = [data mutableBytes];
    int const bytesLength = [data length] - needleSize;
    for (int i = 0; i < bytesLength;) {
        char const* const current = memchr(bytes + i, needle[0],
          bytesLength - i);
        if (current == NULL) return -1;
        if (memcmp(current, needle, needleSize) == 0)
           return current - bytes;
        i = current - bytes + 1;
    }
    return -1;
}
```

The isBatchNumber:numberL methods checks whether a string represents a proper batch number:

```
+ (bool) isBatchNumber:(NSString*)number {
    long long const value = [number longLongValue];
    return value >= 20000000000L && value < 29000000000L;
}
```

The findBatchNumberInChunk:needle:andAddToL method looks for a given string in a portion of the PDF file. It parses the BT and ET tags and contents between them embraced by parentheses. I don't use regular expressions or any kind of grammars. It can be implemented better, but I leave it for readers as an exercise. My implementation is based on two simple state machines: the first iterates over BT and ET tags (the state variable) and cuts the contents surrounded by parentheses, and the second identifies situations when the batch number is extracted and takes the following two strings as the status and date.

```
+ (bool) findBatchNumberInChunk:(char const*)chunk needle:(NSString*)needle
                 andAddTo:(NSMutableArray*)list {
    enum {
        waitBT, waitText, insideText
    } state = waitBT;
    enum {
        waitBatchNumber, waitStatus, waitDate
    } batchParserState = waitBatchNumber;
    NSMutableString* line = [[NSMutableString alloc] init];
    Batch* batch = nil;
    bool found = NO;
    while (*chunk) {
        if (state == waitBT) {
            if (chunk[0] == 'B' && chunk[1] == 'T') {
                state = waitText;
                [line deleteCharactersInRange:NSMakeRange(0,
                      [line length])];
            }
        } else if (state == waitText) {
            if (chunk[0] == '(') {
                state = insideText;
            } else if (chunk[0] == 'E' && chunk[1] == 'T') {
                if (batchParserState == waitBatchNumber) {
                    if ([self isBatchNumber:line]) {
                        [batch autorelease];
                        batch = [[Batch alloc] init];
                        batch.batchNumber = line;
                        batchParserState = waitStatus;
                    }
                } else if (batchParserState == waitStatus) {
                    batch.status = line;
                    batchParserState = waitDate;
                } else if (batchParserState == waitDate) {
                    batch.date = line;
                    batchParserState = waitBatchNumber;
                    if ([batch.batchNumber isEqualToString:needle]) {
                        NSString* pair =
                          [NSString stringWithFormat:@"%@\n%@",
                          batch.status, batch.date];
                        [list addObject:pair];
                        NSLog(@"Found match: '%@' '%@' '%@'",
                          batch.batchNumber, batch.status, batch.date);
                        found = YES;
                    }
                }
                [line autorelease];
                line = [[NSMutableString alloc] init];
                state = waitBT;
            }
        } else if (state == insideText) {
            if (chunk[0] == ')') {
                state = waitText;
            } else {
                char const c[2] = { chunk[0], 0 };
                [line appendString:[NSString stringWithUTF8String:&c[0]]];
            }
        }
        chunk += 1;
    }
    [line release];
    [batch release];
    return found;
}
```

Now the main method, findInPortionMethod:needle:andAddTo:. This method tries to find a block surrounded by the stream\r\n and endstream\r\n markers in the data received so far. If the block has been found, it tries to inflate it and pass it to the findBatchNumberInChunk: method. After the block is processed, it gets removed from the buffer.

```
+ (bool)findInPortion:(NSMutableData *)portion needle:(NSString*)needle
          andAddTo:(NSMutableArray*)list {
    static char const* const streamStartMarker = "stream\x0d\x0a";
    static char const* const streamStopMarker = "endstream\x0d\x0a";
    bool found = false;
    while (true) {
        int const beginPosition = [self findInData:portion fromOffset:0
            needle:streamStartMarker];
        if (beginPosition == -1) break;
        int const endPosition = [self findInData:portion
          fromOffset:beginPosition needle:streamStopMarker];
        if (endPosition == -1) break;
        int const blockLength = endPosition + strlen(streamStopMarker) -
            beginPosition;
        char const* const zipped = [portion mutableBytes] + beginPosition +
                strlen(streamStartMarker);
        z_stream zstream;
        memset(&zstream, 0, sizeof(zstream));
        int const zippedLength = blockLength - strlen(streamStartMarker) -
               strlen(streamStopMarker);
        zstream.avail_in = zippedLength;
        zstream.avail_out = zstream.avail_in * 10;
        zstream.next_in = (Bytef*)zipped;
        char* const unzipped = malloc(zstream.avail_out);
        zstream.next_out = (Bytef*)unzipped;
        int const zstatus = inflateInit(&zstream);
        if (zstatus == Z_OK) {
            int const inflateStatus = inflate(&zstream, Z_FINISH);
            if (inflateStatus >= 0) {
                found = found || [BatchPDFParser
                       findBatchNumberInChunk:unzipped needle:needle
                       andAddTo:list];
            } else {
                NSLog(@"inflate() failed, error %d", inflateStatus);
            }
        } else {
            NSLog(@"Unable to initialize zlib, error %d", zstatus);
        }
        free(unzipped);
        inflateEnd(&zstream);
        int const cutLength = endPosition + strlen(streamStopMarker);
        [portion replaceBytesInRange:NSMakeRange(0, cutLength)
             withBytes:NULL length:0];
    }
    return found;
}
@end
```

*DirectDownloadViewDelegate.h*

This is a header file declaring callbacks from NSURLConnection:

```
@protocol DirectDownloadViewDelegate<NSObject>
- (void)setProgress: (float)progress;
- (void)appendStatus: (NSString*)status;
- (void)setCompleteDate: (NSString*)date;
@end
```

*DirectDownloadDelegate.h*

The NSURLConnectionDelegate delegate.

```
#import "DirectDownloadViewDelegate.h"
@interface DirectDownloadDelegate : NSObject {
    NSError *error;
    BOOL done;
    BOOL found;
    NSMutableData *receivedData;
    float expectedBytes, receivedBytes;
    id<DirectDownloadViewDelegate> viewDelegate;
    NSString* needle;
}
- (id) initWithNeedle:(NSString*)aNeedle andViewDelegate:
    (id<DirectDownloadViewDelegate>)aViewDelegate;
@property (atomic, readonly, getter=isDone) BOOL done;
@property (atomic, readonly, getter=isFound) BOOL found;
@property (atomic, readonly) NSError *error;
@end
```

*DirectDownloadDelegate.m*

This file contains the NSURLConnectionDelegate implementation.

```
#import <Foundation/Foundation.h>
#import "DirectDownloadDelegate.h"
#import "BatchPDFParser.h"
@implementation DirectDownloadDelegate
@synthesize error, done, found;
```

The initWithNeedle:andViewDelegate: constructor creates a delegate. Interestingly, the NSURLConnection delegate is parameterized by another delegate, DirectDownloadViewDelegate. The last one allows the downloader and PDF parser to publish the results to the screen.

```
- (id) initWithNeedle:(NSString*)aNeedle andViewDelegate:
    (id<DirectDownloadViewDelegate>)aViewDelegate {
    viewDelegate = aViewDelegate;
    [viewDelegate retain];
    needle = [[NSString alloc] initWithString:aNeedle];
    receivedData = [[NSMutableData alloc] init];
    expectedBytes = receivedBytes = 0.0;
    found = NO;
    return self;
}
- (void) dealloc {
    [error release];
    [receivedData release];
    [needle release];
    [viewDelegate release];
    [super dealloc];
}
```

The connectionDidFinishLoading: method is invoked when the connection is finished.

```
- (void) connectionDidFinishLoading:(NSURLConnection *)connection {
        done = YES;
        NSLog(@"Connection finished");
    }
```

The connection:didFailWithError: method is invoked when an error occurs.

```
- (void) connection:(NSURLConnection *)connection
         didFailWithError:(NSError *)anError {
        error = [anError retain];
        [self connectionDidFinishLoading:connection];
    }
```

The connection:didReceiveData: method is called when a portion of data is received from the socket. We try to extract the payload from this portion and print it out. Also we update the progress bar.

```
- (void) connection:(NSURLConnection *)connection
             didReceiveData:(NSData *)someData {
        receivedBytes += [someData length];
        [viewDelegate setProgress:(receivedBytes / expectedBytes)];
        [receivedData appendData:someData];
        NSMutableArray* list = [[NSMutableArray alloc] init];
        bool foundInCurrentPortion = [BatchPDFParser
          findInPortion:receivedData needle:needle andAddTo:list];
        for (id batch in list) {
          NSLog(@"[%@]",
     [batch stringByReplacingOccurrencesOfString:@"\n" withString:@"\\n"]);
          [viewDelegate appendStatus:batch];
        }
        [list release];
        found = found || foundInCurrentPortion;
    }
```

The last callback, connection:didReceiveReponse:, is called when HTTP headers are received. We extract the Content-Length header to update the progress bar correctly later on.

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:
        (NSHTTPURLResponse *)someResponse {
    NSDictionary *headers = [someResponse allHeaderFields];
    NSLog(@"[didReceiveResponse] response headers: %@", headers);
    if (headers) {
        if ([headers objectForKey: @"Content-Length"]) {
            NSLog(@"Content-Length: %@", [headers objectForKey:
                   @"Content-Length"]);
            expectedBytes = [[headers objectForKey:
                   @"Content-Length"] floatValue];
        } else {
            NSLog(@"No Content-Length header found");
        }
    }
}
@end
```

*NSURLConnectionDirectDownload.h*

This file contains a declaration of an extra method we are adding to the NSURLConnection class. Note, there is something interesting going on here. We don't need to have the sources of the NSURLConnection class. Objective-C has the concept of categories, allowing mixing in new methods to the existing class. I call our new category DirectDownload.

```
@interface NSURLConnection (DirectDownload)
+ (BOOL) downloadAtURL:(NSURL *)url searching:(NSString*)batchNumber
       viewingOn:(id)viewDelegate;
@end
```

*NSURLConnectionDirectDownload.m*

Now the final bit of the downloader. The function donwloadAtURL:searching:viewingOn: below creates a connection and kicks off downloading. Then it waits in a loop (NSRunLoop) until the file is received. This loop allows the rest of the application to process events and remain responsive. Interestingly, this function is abstracted from the particular UI. It uses the viewDelegate delegate to interact with the UI (update the progress bar and print out parsed pieces of the PDF file).

```
#import <Foundation/Foundation.h>
#import "DirectDownloadDelegate.h"
@implementation NSURLConnection (DirectDownload)
+ (BOOL) downloadAtURL:(NSURL *)url searching:(NSString*)batchNumber
      viewingOn:(id)viewDelegate {
    NSMutableURLRequest *request = [[NSMutableURLRequest alloc]
      initWithURL:url];
    DirectDownloadDelegate *delegate = [[[DirectDownloadDelegate alloc]
     initWithNeedle:batchNumber andViewDelegate:viewDelegate] autorelease];
    NSURLConnection *connection = [[NSURLConnection alloc]
     initWithRequest:request delegate:delegate];
    [request release];
    while ([delegate isDone] == NO) {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
           dateWithTimeIntervalSinceNow:1.0]];
    }
    if ([delegate isFound] != YES) {
        [viewDelegate appendStatus:@"This batch number is not found."];
        NSLog(@"This batch number is not found.");
    }
    NSLog(@"PDF is processed");
    [connection release];
    NSDateFormatter* dateFormatter = [[NSDateFormatter alloc] init];
    dateFormatter.dateFormat = @"yyyy/MM/dd HH:mm:ss";
    NSString* lastUpdateDate = [dateFormatter stringFromDate:[NSDate
     date]];
    NSLog(@"Last update at: %@", lastUpdateDate);
    [viewDelegate setCompleteDate:lastUpdateDate];
    [dateFormatter release];
    NSError *error = [delegate error];
    if (error != nil) {
        NSLog(@"Download error: %@", error);
        return NO;
    }
    return YES;
}
@end
```

*ViewController.m*

This file contains an implementation of the view controller. This file is not the same as in the full-blown UI application, because, obviously, we have no UI here. So, this significantly simplified version of the controller only has a few stub functions allowing checking whether the callbacks are actually invoked by the downloader.

```
#import <Foundation/Foundation.h>
#import "DirectDownloadViewDelegate.h"
#define IBAction void
```

Then we declare an empty stub object mocking the view controller.

```
@interface ViewController : NSObject <DirectDownloadViewDelegate>
@end
#import "NSURLConnectionDirectDownload.h"
```

There is a URL of our PDF file.

```
static char const* const pdf =
  "http://photos.state.gov/libraries/unitedkingdom/164203/cons-visa/
  admin_processing_dates.pdf";
```

Now the view controller mock implementation.

```
@implementation ViewController
```

The appendStatus: callback is invoked when we have discovered one more update related to the given batch number. The real view controller will append the information to a UI widget, but here we only log it.

```
- (void) appendStatus:(NSString*)status {
    NSLog(@"appendStatus(): '%@'", [status
      stringByReplacingOccurrencesOfString:@"\n" withString:@"\\n"]);
    // Some code is skipped here
    // because not required for the command line mode.
```

The setProgress: callback is called when the downloader wants to update the progress bar. Technically, we can print out the value that is passed in if we're in doubt, but the values will be running from 0% to 100% with some step, so it pollutes the log with not-quite-useful stuff. As an exercise, readers could try logging the parameter via NSLog() and checking the result.

```
- (void) setProgress:(float)progress {
    // Some code is skipped here
    // because not required for the command line mode.
```

The setCompleteDate: callback is called when the PDF file is fully downloaded. In this case our UI will update a label widget representing the last update time. But here, again, we simply log it.

```
- (void) setCompleteDate:(NSString*)date {
    NSLog(@"setCompleteDate(): '%@'", date);
    // Some code is skipped here
    // because not required for the command line mode.
```

The last method in this file is updateBatchStatus:. This is a starting point that we use to kick off downloading. It accepts the batch number as a parameter and then calls NSURLConnection.

```
- (bool) updateBatchStatus:(NSString*)batchNumber {
    NSURL *url = [[[NSURL alloc] initWithString:[NSString
      stringWithCString:pdf encoding:NSASCIIStringEncoding]] autorelease];
    return [NSURLConnection downloadAtURL:url searching:batchNumber
      viewingOn:self];
}
```

*main-cli.m*

This is an entry point of our command-line application. We create a mock of the view controller, then after extracting the batch number from the command we kick off downloading.

```
#import <Foundation/Foundation.h>
#import "DirectDownloadDelegate.h"
@interface ViewController : NSObject <DirectDownloadViewDelegate>
- (bool) updateBatchStatus:(NSString*)batchNumber;
@end
int main(int argc, char *argv[]) {
    @autoreleasepool {
        ViewController* viewController = [ViewController alloc];
        [viewController updateBatchStatus:[NSString
          stringWithCString:argv[1] encoding:NSASCIIStringEncoding]];
        [viewController release];
    }
    return 0;
}
```

*Want to give it a whirl?*

Okay, the Makefile. For Mac it'll be very simple:

```
files = \
    ViewController.m \
    BatchPDFParser.m \
    NSURLConnectionDirectDownload.m \
    DirectDownloadDelegate.m
    main-cli.m
all: build run
build:
    clang -o USVisaTest -DTESTING -framework Foundation -lz $(files)
run:
    ./USVisaTest 20121456171
```

For GNUstep running on Windows it'll be a bit clunky (this file needs to be named GNUmakefile, not Makefile):

```
include $(GNUSTEP_MAKEFILES)/common.make
TOOL_NAME = USVisa
USVisa_OBJC_FILES = \
    ../ViewController.m \
    ../BatchPDFParser.m \
    ../NSURLConnectionDirectDownload.m \
    ../DirectDownloadDelegate.m \
    ../main-cli.m
USVisa_TOOL_LIBS = -lz
ADDITIONAL_OBJCFLAGS = -DTESTING
CC = clang
include $(GNUSTEP_MAKEFILES)/tool.make
run:
    ./obj/USVisa 20121456171
```

Now we can build simply by running make. For example, on Windows, it should come up with something like:

```
This is gnustep-make 2.6.2.
  Type 'mmake print-gnustep-make-help' for help.
Making all for tool USVisa...
 Creating obj/USVisa.obj/../...
 Compiling file ViewController.m ...
 Compiling file BatchPDFParser.m ...
 Compiling file NSURLConnectionDirectDownload.m ...
 Compiling file DirectDownloadDelegate.m ...
 Compiling file main-cli.m ...
 Linking tool USVisa ...
```

And finally we can try running it using some test batch number:

```
make run
```

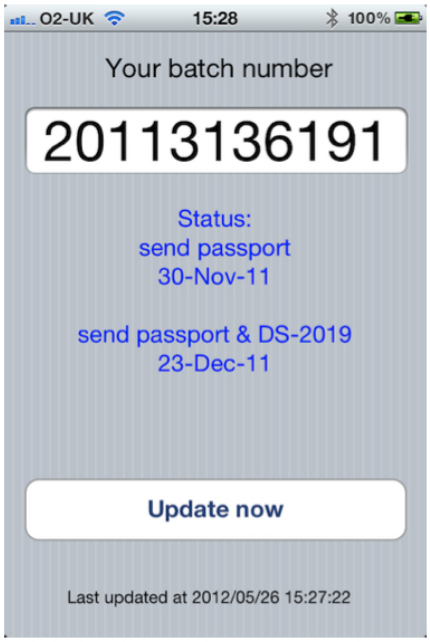
In my case it printed the following:

```
This is gnustep-make 2.6.2. Type 'mmake print-gnustep-make-help' for help.
    ./obj/USVisa 20121456171
    2012-06-19 17:27:11.472 USVisa[3420] [didReceiveResponse] response headers:
      {"Accept-Ranges" = bytes; "Cache-Control" = "max-age=600";
      Connection = "keep-alive"; "Content-Length" = 2237242;
      "Content-Type" = "application/pdf"; Date = "Tue, 19 Jun 2012 16:27:11 GMT
      ETag = "\"4b2ca3e41de5ba4ae45670e776edfc3b:1339778351\"";
      "Last-Modified" = "Fri, 15 Jun 2012 16:06:15 GMT"; Server = Apache; }
    2012-06-19 17:27:11.604 USVisa[3420] Content-Length: 2237242
    2012-06-19 17:27:12.093 USVisa[3420] Found match: '20121456171'
      'send passport & new travel itinerary' '14-Jun-12'
    2012-06-19 17:27:12.104 USVisa[3420] [send passport &
      new travel itinerary\n14-Jun-12]
    2012-06-19 17:27:12.111 USVisa[3420] appendStatus():
      'send passport & new travel itinerary\n14-Jun-12'
    2012-06-19 17:27:13.769 USVisa[3420] Connection finished
    2012-06-19 17:27:13.774 USVisa[3420] PDF is processed
    2012-06-19 17:27:13.961 USVisa[3420] Last update at: 2012/06/19 16:27:13
    2012-06-19 17:27:13.972 USVisa[3420] setCompleteDate(): '2012/06/19 16:27:1
```

So, we have proven that our PDF parser and downloader work as expected, and we can proceed to the proper iOS application. Unfortunately, Windows users cannot try any of the following code, but I hope they will enjoy watching it.

## The Application UI Design

I wanted my application to be very simple: only one screen having just a text entry field for the batch number, a refresh button and the area to print out the status. For example:




Also to indicate the progress of downloading it may temporarily display a progress bar.

*ViewController.h*

This is a full implementation of the ViewContoller.h. I will be skipping details explained previously within the reduced, command-line version of the controller.

```
#import <Foundation/Foundation.h>
#import "DirectDownloadViewDelegate.h"
#ifdef TESTING
#define IBAction void
@interface ViewController : NSObject <DirectDownloadViewDelegate>
@end
#else
#import "ViewController.h"
#endif
#import "NSURLConnectionDirectDownload.h"
static char const* const pdf = "http://photos.state.gov/libraries/
  unitedkingdom/164203/cons-visa/admin_processing_dates.pdf";
@implementation ViewController
#ifndef TESTING
@synthesize updateProgressView, batchNumberTextField, statusTextView,
  lastUpdatedLabel, updateButton;
#endif
NSString* const PropertiesFilename = @"Properties";
NSString *pathInDocumentDirectory(NSString *fileName) {
    NSArray *documentDirectories = NSSearchPathForDirectoriesInDomains
      (NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentDirectory = [documentDirectories objectAtIndex:0];
    return [documentDirectory stringByAppendingPathComponent:fileName];
}
```

appendStatus: is a callback from the PDF parse saying that the status update regarding our batch number has been found. We check that it is not empty and stick it to the screen.

```
- (void) appendStatus:(NSString*)status {
    NSLog(@"appendStatus(): '%@'", [status
      stringByReplacingOccurrencesOfString:@"\n" withString:@"\\n"]);
#ifndef TESTING
    if ([[statusTextView text] length] == 0)
        [statusTextView setText:@"Status:\n"];
    [statusTextView setText:[[statusTextView text]
      stringByAppendingString:status]];
    [statusTextView setText:[[statusTextView text]
      stringByAppendingString:@"\n"]];
#endif
```

setProcess: is another callback invoking when a chunk of data is received and we need to update the progress bar.

```
- (void) setProgress:(float)progress {
#ifndef TESTING
    updateProgressView.progress = progress;
#endif
}
```

setCompleteDate: is yet another callback updating the time of the last success update.

```
- (void) setCompleteDate:(NSString*)date {
    NSLog(@"setCompleteDate(): '%@'", date);
#ifndef TESTING
    [lastUpdatedLabel setText:date];
#endif
}
```

Finally, the updateBatchStatus: is a method we call to kick off downloading of the PDF.

```
- (bool) updateBatchStatus:(NSString*)batchNumber {
    NSURL *url = [[[NSURL alloc] initWithString:[NSString
      stringWithCString:pdf encoding:NSASCIIStringEncoding]] autorelease];
    return [NSURLConnection downloadAtURL:url searching:batchNumber
      viewingOn:self];
}
```

Now there are a few iOS-specific methods. viewDidLoad: is called by the application loader when the view is activated and ready to use. In this method we create a spinner indicator on the fly. We will be displaying it when downloading is in progress. Also we tweak heights of two buttons because for some reason the Xcode Interface Builder doesn't allow changing it in the UI.

```
#ifndef TESTING
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view,
    // typically from a nib.
    spinnerActivityIndicatorView = [[UIActivityIndicatorView alloc]
    initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhiteLarge];
    [spinnerActivityIndicatorView setColor:[UIColor blueColor]];
    CGSize size = [[self view] frame].size;
    [spinnerActivityIndicatorView setCenter:CGPointMake(size.width / 2,
      size.height / 2 + 60)];
    [self.view addSubview:spinnerActivityIndicatorView];
    CGRect rect = [self.updateButton bounds];
    rect.size.height += 10;
    [self.updateButton setBounds:rect];
    rect = [self.batchNumberTextField bounds];
    rect.size.height += 20;
    [self.batchNumberTextField setBounds:rect];
#ifdef DEBUG
    NSLog(@"DEBUG mode");
#endif
}
```

viewDidUnload: is called when the view becomes inactive.

```
- (void)viewDidUnload
{
    [super viewDidUnload];
    // Release any retained subviews of the main view.
}
```

shouldAutorotateToInterfaceOrientation: allows to control the screen rotation in the iOS application. Here we say that we only allow the portrait mode.

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation
{
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
#endif
```

launchUpdate: is an action attached to an Update button on the screen. We disable the button itself, activate the progress bar and the spinner indicator, and invoke the updateBatchStatus: method. After it is finished we restore the things in the reverse order.

```
- (IBAction)launchUpdate:(id)sender {
    [self setProgress:0.0];
#ifndef TESTING
    [updateButton setEnabled: NO];
    [updateProgressView setHidden:NO];
    NSString* previousStatus = [statusTextView text];
    [statusTextView setText:@""];
    NSString* batchNumber = [batchNumberTextField text];
    [spinnerActivityIndicatorView startAnimating];
    BOOL const ok = [self updateBatchStatus:batchNumber];
    [spinnerActivityIndicatorView stopAnimating];
    if (!ok) {
        UIAlertView *alert =
            [[UIAlertView alloc] initWithTitle:@"Error"
                                       message:@"Internet connectivity
                                                problem"
                                      delegate:self cancelButtonTitle:nil
                             otherButtonTitles:@"OK", nil];
        [alert show];
        [alert release];
        [statusTextView setText:previousStatus];
    }
    [updateProgressView setHidden:YES];
    [updateButton setEnabled: YES];
#endif
}
```

The savePropertiesmethod stores the content of the screen element to a file.

```
- (void) saveProperties {
    NSDictionary *props = [[NSDictionary alloc] initWithObjectsAndKeys:
#ifndef TESTING
                           batchNumberTextField.text,
                             @"batchNumberTextField",
                           statusTextView.text, @"statusTextView",
                           lastUpdatedLabel.text, @"lastUpdatedLabel",
#endif
                            nil];
    for (NSString* key in props) {
        NSLog(@"%@ - %@", key, [props objectForKey:key]);
    }
    NSString* filename = pathInDocumentDirectory(PropertiesFilename);
    if ([props writeToFile:filename atomically:YES] == NO)
        NSLog(@"Unable to save properties into file [%@]", filename);
    [props release];
}
```

The loadPropertiesmethod is a counterpart of saveProperties:.

```
- (void) loadProperties {
    NSDictionary *props = [[NSDictionary alloc]
    initWithContentsOfFile:pathInDocumentDirectory(PropertiesFilename)];
    for (NSString* key in props) {
        NSLog(@"%@ - %@", key, [props objectForKey:key]);
    }
#ifndef TESTING
    [batchNumberTextField setText:[props
      objectForKey:@"batchNumberTextField"]];
    [statusTextView setText:[props objectForKey:@"statusTextView"]];
    [lastUpdatedLabel setText:[props objectForKey:@"lastUpdatedLabel"]];
#endif
    [props release];
}
- (IBAction)textFieldReturn:(id)sender {
#ifndef TESTING
    [sender resignFirstResponder];
#endif
}
-(IBAction)backgroundTouched:(id)sender {
#ifndef TESTING
    [batchNumberTextField resignFirstResponder];
#endif
}
@end
```

This is basically it! We have all the parts to build the application.

I've uploaded the project to GitHub—usvisa[U11], so please free to mess around. Any feedback is more than appreciated.

## And One More Thing!

The catchy icon is the most important thing if you're out to sell zillions of copies of your app. In our example, we'll simply take it from the most respectful source—Wikipedia. It will be The Great Seal of the United States[U12].



## P.S.

Funnily enough I decided to open-source the application and dissect it in this article after it had been rejected by Apple from the AppStore. They referred to a statement in the official guidelines for iOS developers saying that applications with very limited functionality, and which can be re-implemented in HTML5, will be rejected. They want to avoid more farting or displaying single static pictures applications. I would probably argue with the censor about "limited functionality" or "can be implemented as an HTML5 app," but I decided not do so. First, I appreciate Apple's effort to reduce the amount of crap in the AppStore, and second, I had to dive quite deep into Objective-C and Cocoa, and I don't regret it. I'm currently working on two more apps.



**About the Author**

Alexander Demin is a software engineer and Ph.D. in Computer Science. Constantly exploring new technologies, he believes that something amazing is always out there. He can be contacted at alexander@demin.ws [U13] or through his homepage and blog [U14].

Send the author your feedback [U15] or discuss the article in the magazine forum [U16].

**External resources referenced in this article:**

[U1] http://photos.state.gov/libraries/unitedkingdom/164203/cons-visa/admin_processing_dates.pdf
[U2] http://www.gnustep.org/
[U3] http://solarianprogrammer.com/2011/09/14/learn-objective-c-on-windows/
[U4] http://solarianprogrammer.com/2012/03/21/clang-objective-c-windows/
[U5] http://www.amazon.co.uk/gp/product/B007OWBAB0/
[U6] http://www.amazon.co.uk/gp/product/0321706285/
[U7] http://www.amazon.co.uk/gp/product/B006GFZ288/
[U8] http://pierre.chachatelier.fr/programmation/fichiers/cpp-objc-en.pdf
[U9] http://www.adobe.com/devnet/pdf/pdf_reference_archive.html
[U10] http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/pdf_reference_archives/PDFReference.pdf
[U11] https://github.com/begoon/usvisa-app
[U12] http://en.wikipedia.org/wiki/Great_Seal_of_the_United_States
[U13] mailto:alexander@demin.ws
[U14] http://demin.ws
[U15] mailto:michael@pragprog.com?subject=go
[U16] http://forums.pragprog.com/forums/134

# Threads

## Three Ages of Chris Espinosa

*by Michael Swaine*

Chris Espinosa was just a precocious kid when he started working at Apple. He has kids of his own now, he's still at Apple, and he has some great memories.



Paul Freiberger and I are currently writing the third edition of *Fire in the Valley*, our history of the personal computer—to be published by The Pragmatic Bookshelf, naturally. As we go back over the material, we keep finding threads that run through the whole narrative. I am sharing some of these as brief history articles here in *PragPub*.

This time I want to share the story of someone who has had a close-up view of Apple throughout its entire amazing history: Chris Espinosa.

### The Veteran: 1996-the present

In the run-up to the iPhone release, Chris Espinosa had been tasked with bringing the Xcode development environment up for this new platform. Except that nobody told him that that was what he was doing. "They just said, 'Here's the processor architecture. Here's what you need to do, and we'll tell you when you've done it right.'"

It had been different before. Once Steve returned to Apple, Chris recalls, the company became "stovepiped and compartmentalized." He was used to working very broadly at Apple, talking regularly with people in marketing and sales and support and software. Now he "would go for months without talking with somebody outside my local organization." It wasn't bad, but it was different.

He'd had more access to Steve back then, too. Now he'd run into him at lunch once in a while. He introduced his son to Steve. Steve was always congenial, but there just wasn't much interaction. Which was fine with Chris. Things had been pretty intense in the early days. Now he was enjoying his relative anonymity, working in developer tools.

Then one day he got a call.

It was the day before the iPhone shipped, and Steve had scheduled a big employee communication meeting to celebrate the event. When the phone rang in Chris's office, he picked it up and the voice on the other end said, "Please hold for Steve."

Chris held.

It turned out that Steve was calling to get Chris's thoughts on the iPhone. "This iPhone," Steve said, "What do you think of it?"

Chris was flabbergasted. He had to admit that he hadn't touched one. The release had been extremely secretive, and he'd watched the announcement along with the general public. But he realized that he had an opinion. "From everything I know about it," he told Steve, "it's the best thing we've ever done."

Steve thanked him and they chatted some more, and the call ended and Chris went back to work. And when the company-wide memo from Steve came out announcing the meeting, it said that Steve thought the iPhone was "the best thing we've ever done."

## The Professional: 1984-1986

It meant a lot to Chris that Steve had turned to him for confirmation that he wasn't blowing smoke about the iPhone. Chris remembered how it had been two decades earlier. As a brash kid in a company where anybody could talk to anybody, he had played the role of court jester to Steve. "After we had shipped the Mac and were trying to do the Macintosh Office—the server, the laser printer," he remembers, "[Steve] really wanted to believe rosy scenarios, and he pushed people to [promise] things that were unachievable.... It was basically what he lost his job over." And Chris was calling him on all of it.

Somebody needed to. For the first two years of the Macintosh's life, it failed to deliver the sales Steve had projected. The aging Apple II was keeping the company alive.

When Steve got pushed out, Chris hung in there. Apple had always been more than a job to him. He now had relatives also employed at Apple, in the Apple II division. He stuck it out through a succession of CEOs and ups and downs. He watched John Sculley put out the immediate fires, and saw the desktop publishing revolution gave Apple a huge boost.

"We were making 55 percent gross margins," Chris recalled, "on our way to becoming a 10 billion dollar company. We were in fat city."

But the deeper problems at Apple couldn't be papered over. The company lost market share. Its product line lost focus. The operating system was showing its age. And there was no powerful vision driving decisions. By the mid-1990s it was widely assumed that Apple would be bought by another company. Sun Microsystems made an offer, but it was rejected. Before long, Apple was bound to get an offer it couldn't refuse.

Chris had never had a job outside Apple, and he wasn't looking for one now, no matter how bad things looked. He would hang on for the endgame. He told himself he'd "stick around to turn out the lights."

## The Kid: 1975-1984

Of course things didn't work out that way. But Chris couldn't have predicted that back then. He couldn't have predicted any of Apple's remarkable story, and he arguably had more perspective on it than anybody else.

It had begun when he was just 14 years old. That was when he and Randy Wigginton would catch a ride to the Homebrew Computer Club meetings with Woz. Soon he was hanging out at Woz's place with Randy, where Woz let then write programs for his prototype Apple II. Or he'd spend time in that new store that sold computer stuff, the Byte Shop.

That was where he'd first seen Steve Jobs. It was Steve who had hired him to work at Apple, impressed by the demo program running on the Apple II and even more impressed that it had been written by this little kid. Steve was

different from Woz, though. Chris always had the feeling that Steve was shaping him.

He never really left Apple, even when he went to college. While there, he spent his free time writing the official Apple II manual.

It's been a long time since he was that kid riding his moped to work to write software or demo the Apple II to people who walked into Apple's office. Now he has kids of his own, and one of them just got back from summer camp where he was writing code using Xcode, the development environment Chris and his team built.

# Calendar

Author sightings, partner events, and other notable happenings.



Gotta lead with this item because it so obviously belongs *up front*. Front-end developers, take note:



Sept 20–21 **The Treasure of the Frontend Island: "On Friday 21st join us on a mind-blowing journey with some of the biggest names in our industry. The conference will take place at the inspiring Teatro Duse, a wonderful XVII century building recently renovated." Also workshops on the 20th. Not to mention the pirate theme and the Italian craft beers at the after party!**
The Treasure of the Frontend Island Frontend Conference [U1], Bologna, IT

## Author Appearances

Who's where, and what for.

Sept 5 **"Understanding the iOS Graphics System Workshop"**
Bill Dudney, co-author of iOS SDK Development [U2]
Swipe Conference [U3], Sydney, Australia

Sept 5–7 **"Intro to iOS Programming"**
Bill Dudney, co-author of iOS SDK Development [U4]
Swipe Conference [U5], Sydney, Australia

Sept 6 **"Working with Core Data"**
Bill Dudney, co-author of iOS SDK Development [U6]
Swipe Conference [U7], Sydney, Australia

Sept 6–7 **"Agile Portfolio Planning: Managing Your Project Portfolio" and "Workshop: Hiring For Your Agile Team" and "Workshop: Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects"**
Johanna Rothman, author of Behind Closed Doors: Secrets of Great Management [U8], Manage It! Your Guide to Modern, Pragmatic Project Management [U9], and Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects [U10]
Agile Brazil [U11], Sao Paulo, Brazil

Sept 6–7 **"Workshop: Pulling together as a Team"**
Rachel Davies, co-author of Agile Coaching [U12]
Agile on the Beach [U13], Falmouth, UK

Sept 7 **The RSpec Toolbox**
David Chelimsky, co-author of The RSpec Book [U14]
Windy City Rails [U15], Chicago, IL

Sept 7-9 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U16], the author of Programming Groovy, [U17] Programming Scala [U18] and Programming Concurrency on the JVM [U19]
NFJS [U20], Seattle, WA

Sept 8–9 **"The Three Laws of Bulls*it"**
Paolo "Nusco" Perrotta, author of Metaprogramming Ruby [U21]
Barcelona, Spain

Sept 10 **Scala for the Intrigued**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U22], the author of Programming Groovy, [U23] Programming Scala [U24] and Programming Concurrency on the JVM [U25]
Twin Cities Java Users Group [U26], Eagan, MN

Sept 11 **"The Rails View"**
John Athayde, co-author of The Rails View [U27]
CVReg [U28]Glen Allen, VA

Sept 11 **"Live Document Syncing with iCloud"**
Jim Dovey
CocoaHeads [U29], Toronto, CA

Sept 11 **"Uncomfortable with Agile"**
Andy Hunt
Agile RTP [U30] Raleigh, NC

Sept 12 **Design Pattern on Modern JVM Languages**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U31], the author of Programming Groovy, [U32] Programming Scala [U33] and Programming Concurrency on the JVM [U34]
Dallas Java Metroplex Users Group [U35], Dallas, TX

Sept 14–15 **Services, Scale, Backgrounding and WTF is going on here?**
David Copeland, Author of Build Awesome Command-Line Applications in Ruby [U36]
Golden Gate Ruby Conference [U37], San Francisco, CA

Sept 14–15 **"Deploy, Scale and Sleep Better with JRuby"**
Joe Kutner, author of Deploying with JRuby [U38]
GoGaRuCo [U39]San Francisco, CA

Sept 14-16 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U40], the author of Programming Groovy, [U41] Programming Scala [U42] and Programming Concurrency on the JVM [U43]
NFJS [U44], Boston, MA

Sept 17 **"CoffeeScript and Testing JavaScript"**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U45], the author of Programming Groovy, [U46] Programming Scala [U47] and Programming Concurrency on the JVM [U48]
HTML5 Denver [U49], Denver, CO

Sept 19 **Vim Masterclass**
Drew Neil, author of Practical Vim [U50]
Rocky Mountain Ruby Conference [U51], Boulder, CO

Sept 19–20 **"Deploy, Scale and Sleep Better with JRuby"**
Joe Kutner, author of Deploying with JRuby [U52]
Frozen Rails [U53]Helsinki, Finland

Sept 21-23 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U54], the author of Programming Groovy, [U55] Programming Scala [U56] and Programming Concurrency on the JVM [U57]
NFJS [U58], Atlanta, GA

Sept 23 **Vim Masterclass**
Drew Neil, author of Practical Vim [U59]
Strange Loop [U60], St. Louis, MO

Sept 25 **"Uncomfortable with Agile"**
Andy Hunt
Agile Carolinas [U61] Charlotte, NC

Sept 27–28 **TBA**
Michael Bevilacqua-Linn
Surge [U62]Baltimore, MD

Sept 27–28 **"Workshop: Improve your Coaching Skills"**
Rachel Davies, co-author of Agile Coaching [U63]
Agile Cambridge [U64], UK

Sept 30–Oct 4 **Multiple topics on Java, Groovy, and Scala**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U65], the author of Programming Groovy, [U66] Programming Scala [U67] and Programming Concurrency on the JVM [U68]
JavaOne [U69], San Francisco, CA

Oct 1 **Behavior-Driven Objects**
David Chelimsky, co-author of The RSpec Book [U70]
Agile Testing & BDD Exchange NYC [U71], New York, NY

Oct 1–2 **"Advanced Agile" (co-teaching with Alistair Cockburn)**
Henrik Kniberg, author of Lean from the Trenches [U72]
Crisp [U73], Stockholm, Sweden

Oct 2 **"Continuous Delivery"**
Michael Nygard, author of Release It! [U74]
GOTO [U75], Aarhus, Denmark

Oct 3 **Becoming a Kick-*** Test Manager**
Johanna Rothman, author of Behind Closed Doors: Secrets of Great Management [U76]
Manage It! Your Guide to Modern, Pragmatic Project Management [U77]
Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects [U78]
StarWest [U79], Anaheim, CA

Oct 5-7 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U80], the author of Programming Groovy, [U81] Programming Scala [U82] and Programming Concurrency on the JVM [U83]
NFJS [U84], Minneapolis, MN

Oct 6 **Keynote**
Henrik Kniberg, author of Lean from the Trenches [U85]
AgileEE [U86], Kiev, Ukraine

Oct 10 **Wearable Android Computing**
Mike Riley, author of Programming Your Home [U87]
ChicagoAndroid.com and Google Technology User Group Meeting [U88], Chicago, IL

Oct 10-11 **TBA**
Michael Bevilacqua-Linn
Ricon [U89]San Francisco, CA

Oct 15-18 **Talks on Groovy and JVM Languages Integration**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U90], the author of Programming Groovy, [U91] Programming Scala [U92] and Programming Concurrency on the JVM [U93]
Spring2GX [U94], Washington, D.C.

Oct 22 **"Talk: Coaching Teams through Change"**
Rachel Davies, co-author of Agile Coaching [U95]
Heart of England Scrum User Group [U96], Birmingham, UK

Oct 23 **"Test Abstraction: Keeping Your Test-Driven Tests Healthy"**
Jeff Langr, co-author of Agile in a Flash [U97]
AgileDC [U98]Washington, DC

Nov 4–8 **Has Your Project Outgrown Its Training Wheels?, Replacing Management Myths, What's Your Number One Project?, Improve Your Social Networking Skills, Reinventing Yourself, Congruent Coaching, a full day workshop**
Johanna Rothman, author of Behind Closed Doors: Secrets of Great Management [U99] Manage It! Your Guide to Modern, Pragmatic Project Management [U100] Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects [U101] Amplifying Your Effectiveness Conference [U102], Albuquerque, NM

Nov 2-4 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U103], the author of Programming Groovy, [U104] Programming Scala [U105] and Programming Concurrency on the JVM [U106]
NFJS [U107], Reston, VA

Nov 6 **Vim Masterclass**
Drew Neil, author of Practical Vim [U108]
Øredev [U109], Malmo, Sweden

Nov 7 **"Vim—precision editing at the speed of thought"**
Drew Neil, author of Practical Vim [U110]
Øredev [U111], Malmo, Sweden

Nov 9-11 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U112], the author of Programming Groovy, [U113] Programming Scala [U114] and Programming Concurrency on the JVM [U115]
NFJS [U116], Chicago, IL

Nov 13 **"Talk: Coaching teams through Change**
Rachel Davies, co-author of Agile Coaching [U117]
Bath Scrum User Group [U118], Bath, UK

Nov 16-18 **Talks on JVM languages, Concurrency, HTML5, JavaScript, and Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U119], the author of Programming Groovy, [U120] Programming Scala [U121] and Programming Concurrency on the JVM [U122]
NFJS [U123], Denver, CO

Nov 27-30 **Talks and workshops on JavaScript, CoffeeScript, HTML5, Testing**
Venkat Subramaniam, the author of *.NET Gotchas,* the coauthor of 2007 Jolt Productivity Award winning Practices of an Agile Developer [U124], the author of Programming Groovy, [U125] Programming Scala [U126] and Programming Concurrency on the JVM [U127]
RichWeb Experience [U128], Fort Lauderdale, FL

## O'Reilly Events

Upcoming events from our friends at O'Reilly.

Oct 1–2 **Strata Conference: "The O'Reilly Strata Conference in London will bring together the best minds in data to explore the opportunities made possible by the latest state-of-the-art big data tools and analytic approaches."**
Strata Conference [U129], London, England

Oct 2–4 **Velocity Conference: "This October, hundreds of web ops and performance professionals will gather in London to share their ideas and expertise—and to discover the technical skills, tools, tips, tricks, and best practices they need to exceed the continually increasing expectations for a faster, stronger web."**
Velocity Conference [U130], London, England

Oct 9 **TOC Frankfurt: "Tools of Change Frankfurt returns for a fourth year on Tuesday, 9 October 2012—the eve of the Frankfurt Book Fair (10–14 October)—gathering the best and brightest in our global publishing and technology community for a full day of intriguing keynotes, sessions, and networking."**
TOC Frankfurt [U131], Frankfurt, Germany

Oct 16–17 **Strata Rx Conference: "Strata Rx is the first conference to bring data science to the intractable and urgent issues confronting healthcare."**
Strata Rx Conference [U132], San Francisco, CA

Oct 20 **Mini TOC Vancouver: "While the focus at Mini TOC is on the intersection of technology and publishing—just like at the 'big' TOCs—at Mini TOC, the crowd is small, the atmosphere is intimate and informal, and all the emphasis is on attendees and presenters sharing the conversations."**
Mini TOC Vancouver [U133], Vancouver, B.C.

Oct 23–25 **O'Reilly Strata Conference and Hadoop World: "Now in its second year in New York, the O'Reilly Strata Conference explores the changes brought to technology and business by big data, data science, and pervasive computing. This year, Strata has joined forces with Hadoop World to create the largest gathering of the Apache Hadoop community in the world."**
O'Reilly Strata Conference and Hadoop World [U134], New York, NY

## USENIX Events

What's coming from our USENIX friends.

Oct 8–10 **10th USENIX Symposium on Operating Systems Design and Implementation: "The 10th OSDI seeks to present innovative, exciting research in computer systems. OSDI brings together professionals from academic and industrial backgrounds in what has become the premier forum for discussing the design, implementation, and implications of systems software."**
OSDI '12 [U135], Hollywood, CA

## Other Happenings

Sept 1 **This month the GNU Project turns 29.**

Sept 2 **First message sent over Internet, 1969.**

Sept 2 **sendmail creator Eric Allman is 57.**

Sept 10 **Space tourist Charles Simonyi is 64.**

Sept 27 **Perl creator Larry Wall is 58.**

Oct 2 **The Great Quux is 58.**

Oct 4 **The Free Software Foundation is 37.**

Oct 4 **Birthday of John Vincent Atanasoff, one of the inventors of the computer.**

Oct 5 **Linux is 21!**

Oct 6 **Adobe co-founder John Warnock is 72.**

Oct 12 **Birthday of Ole-Johan Dahl, co-creator of the Simula language and thus of object-oriented programming.**

Oct 16 **Mike Muuss, who invented Ping, was born on this date in 1958.**

Oct 25 **Peter Naur, the N in BNF, is 84.**

Oct 28 **Former software executive Bill Gates is 57.**

Oct 29 **On this date in 1969, ARPAnet first linked Doug Engelbart's lab at SRI with Leonard Kleinrock's at UCLA.**

Oct 31 **The traditional date for the leaking of confidential internal Microsoft documents outlining strategies for undermining open source software.**

Nov 1 **Mitch Kapor is 62.**

Nov 8 **Bill Joy is 58.**

Nov 6 **Jerry Yang is 44.**

Nov 10 **Bert Bos, co-creator of Cascading Style Sheets, is 49.**

Nov 10 **Richard Brodie, creator of Microsoft Word, is 52.**

Nov 18 **Pamela Kyle Crossley, historian and creator of SIMPLE, is 57.**

Nov 19 **Dries Buytaert, Drupal founder and lead developer, is 34.**

Nov 22 **Randall Schwartz is 51.**

Nov 22 **PHP creator Rasmus Lerdorf is 44.**

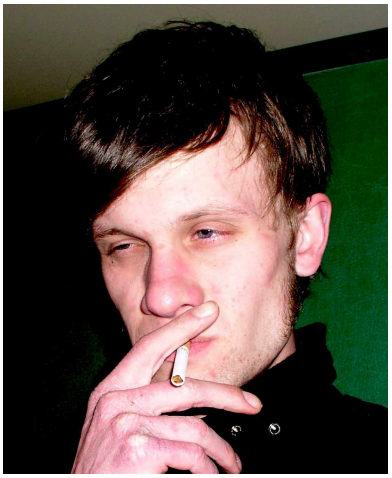
Nov 22 **And it's Charles Simonyi's wedding anniversary.**

# Shady Illuminations

## Apple and Samsung Square Off

*by John Shade*

John reflects on copying, and how Samsung lost a billion dollars and still won.



*If you've seen one rectangle, you've seen them all.* –John Markoff

Tim Cook's most dramatic accomplishment since becoming CEO has been to squeeze a billion out of Samsung for stealing the intellectual property inherent in rounded rectangles. And on his anniversary, even! What a nice present the company lawyers gave him.

I am not a lawyer, but I play one in my imagination. For some reason mny imagination sounds like Carl Sagan.

Here's me, presenting Apple's case against Samsung:

"When Curiosity rolled across the barren Martian landscape to gaze at the side of a 3.4-mile high mountain, it saw vast features that were both incredibly obvious and at the same time never before seen by Man. Great design is like that: it is the discovery of awesome features that noone ever saw before and yet, once they've been shown to you, are incredibly obvious."

And here's me presenting Samsung's case:

"*Rounded rectangles?* Are you freaking kidding me?"

## Apparently Not

The jury forman holds several patents, and told his fellow members of iTheJury that they didn't need to concern themselves with distractions like prior art. I guess if the PTO granted Apple a patent on the wheel or sex or the World Wide Web, the jury figured that Apple then owns the wheel or sex or the World Wide Web. (Thanks, John Perry Barlow, for the examples of things we can all be thankful Apple doesn't hold a patent on.) Perfectly reasonable. The PTO wasn't on trial. Although....

Anyway, the main feature that Apple won on is interesting: a rounded rectangle. I'd show you a picture from the patent application but I'm not stupid. Samsung just paid a billion dollars for showing what that image looks like. I can give you a link to the source code, though: Bill Atkinson's QuickDraw code for roundrect [U1]. See? They *totally* invented that thing.

Now that the trial's over, we can all move on to other things. Like what company Apple will sue next. I think they should take it to the next level. Think of Pinterest, Youtube's mood wall, and Microsoft's UI formerly known as Metro: in each case, a rectangle of rectangles. Is that not a clear case of infringement squared? I can already imagine the exhibits of prior art, though: old episodes of Hollywood Squares and The Brady Bunch.

But really I have nothing but scorn for both sides. Samsung slavishishly copies, Apple sues over rounded rectangles. A pox on both their houses. Of course

both are prospering from all this. Apple gets a billion dollars plus enhanced suing rights. Samsung gets to keep one less of the billions it made from copying Apple and now only has to tweak its designs to satisfy a San Jose jury. Which I guess is a version of design-by-committee.

Before Apple's design, it wasn't obvious that this was the only sensible way to design a smartphone. After, it was hard to make a case for doing anything else. And yet some companies avoided the copying route. Kudos to RIM/Nokia for the courage to stay in their condo when the tsunami rolled in. Kudos to Palm for once again showing that real innovation can be a losing strategy. And Microsoft—I don't know what you're doing. Don't you have patent immunity with Apple? Couldn't you just totally rip off their IP? I know it goes against everything you stand for, but look how well it's worked for Samsung. Instead, you go and do something innovative. #ShakesHead.

**About the Author**

John Shade was born under a cloud in Montreux, Switzerland, in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. His ringtone is sosumi. Send him your feedback [U2] or discuss the article in the magazine forum [U3].

**External resources referenced in this article:**

[U1] http://s3data.computerhistory.org/102658076_quickdraw_acc.zip

[U2] mailto:michael@pragprog.com?subject=shade

[U3] http://forums.pragprog.com/forums/134